WebSphere® software

IBM

WebSphere Business Integration Server Foundation V5.1 Handbook

- Process Choreography with WebSphere Studio Integration Edition V5.1
- Runtime and development of BPEL4WS with sample code
- J2EE Programming Model Extensions

Peter Kovari
Lisa Boardman
Giles Dring
Richard Johnson
Hirotake Kitabayashi
Salman Moghal
Raphael Mueller
Sudhakar Nagarajan
Yu Zeng

Redbooks

ibm.com/redbooks

International Technical Support Organization

WebSphere Business Integrator Server Foundation V5.1 Handbook

November 2004

SG24-6318-00

Note: Before using this information and the product it supports, read the information in “Notices” on page xiii.

First Edition (November 2004)

This edition applies to WebSphere Business Integration Server Foundation V5.1 on Microsoft Windows 2000 Server, IBM AIX 5.2, Red Hat Linux; WebSphere Studio Application Developer Integration Edition V5.1 on Microsoft Windows 2000 Professional, Microsoft Windows XP Pro.

Contents

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

@server®	Cloudscape™	Notes®
@server®	CICS®	PartnerLink®
Redbooks (logo) ™	DB2®	Redbooks™
developerWorks®	Extended Services®	Tivoli®
e-business on demand™	ETE™	TME®
ibm.com®	HACMP™	VisualAge®
pSeries®	Informix®	WebSphere®
xSeries®	IBM®	
AIX®	Lotus®	

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Preface

This IBM® Redbook describes the technical details of WebSphere Business Integration Server Foundation and discusses using WebSphere Studio Application Developer Integration Edition for application development. It provides valuable information for system administrators, developers and architects about the products covered. The book specifically focuses on WebSphere® Process Choreographer and on solutions implementing it.

Part 1, “Architecting a WebSphere Enterprise solution” on page 1 contains high-level details about WebSphere solutions using WebSphere Business Integration Server Foundation.

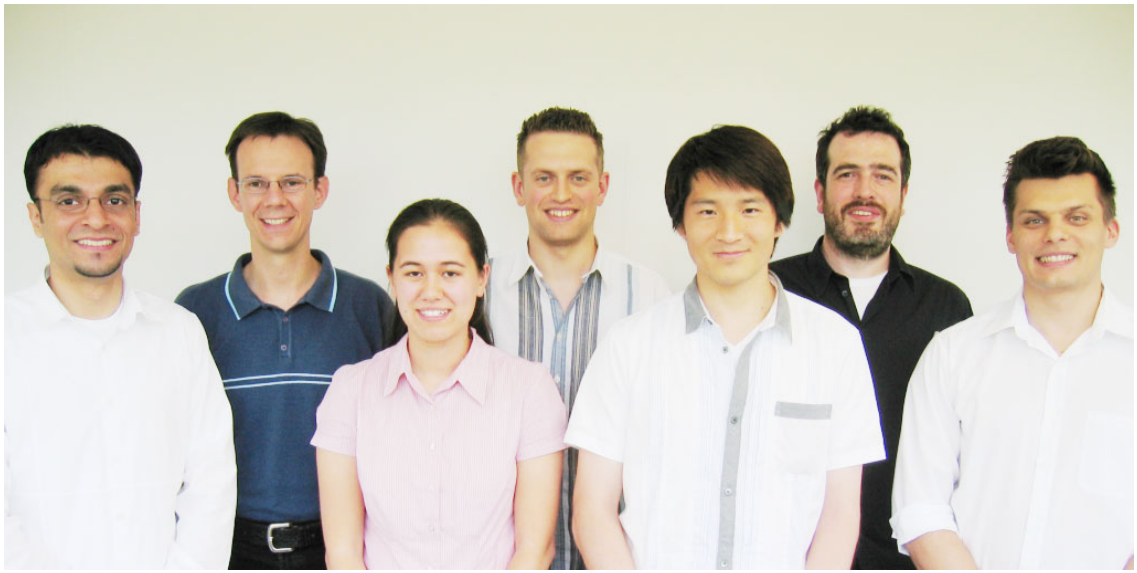
Part 2, “Setting up the environment” on page 29 provides step-by-step details for installing the runtime and development environments.

Part 3, “Implementing WebSphere Enterprise solutions” on page 119 provides details about the J2EE Programming Model Extensions and functions in WebSphere Business Integration Server Foundation. You can learn how to design, develop, assemble, deploy and administer applications in the WebSphere Business Integration Server Foundation environment.

The “Appendixes” on page 559 provide additional information about the sample application.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.



The Team (left to right): Salman Moghal, Raphael Mueller, Lisa Boardman, Richard Johnson, Hirotake Kitabayashi, Giles Dring, Peter Kovari

Peter Kovari is a WebSphere Specialist at the International Technical Support Organization, Raleigh Center. He writes extensively about all areas of WebSphere. His areas of expertise include e-business, e-commerce, security, Internet technologies and mobile computing. Before joining the ITSO, he worked as an IT Specialist for IBM in Hungary.

Lisa Boardman is an IT Specialist at IBM Global Services Australia. Her areas of expertise include Web application development, Web Services and Internet technologies. She has two years of experience at IBM and holds a Bachelor of Computer and Information Science and a Bachelor of Arts(Multimedia Studies) from the University of South Australia.

Giles Dring is an IT Architect in IBM Global Services based in the UK. He works with large UK-based companies to create solutions to their business problems. Lately, he has been spending much of his time helping a UK government department realize its vision of an integration infrastructure. He is also a teacher within the IBM Architectural Methods curriculum and has a keen interest in solution performance. He earned a Masters degree in Electronic and Computer Engineering from the University of Leeds.

Richard Johnson is an Advisory IT Specialist for the IBM Software Services for WebSphere consultancy, based at IBM Hursley Park, UK. In this role, he works as an expert on WebSphere and J2EE, designing and implementing solutions for IBM clients and business partners across EMEA and worldwide. Richard's areas of specialization include Process Choreographer, Web Services and WebSphere to CICS® integration. He has three years of consulting experience and two previous years within CICS Transaction Server development. He holds a Masters degree in Chemistry from the University of Oxford.

Hirotake Kitabayashi is a System Engineer for Hitachi Software Engineering in Tokyo, Japan. Currently, he is working at the IBM WebSphere Performance Lab in Rochester as a trainee. He has four years of experience in Web and J2EE technologies. His areas of expertise include Java™ programming, J2EE performance, Web Services and e-commerce solutions.

Salman Moghal is a Senior IT Specialist for the IBM Software Services for WebSphere group based in IBM Toronto Lab, Canada. He is a technical lead and application architect for WebSphere products, with over eleven years of combined software development experience in Web technologies since 1993. His areas of expertise include WebSphere and J2EE design, WebSphere performance and migration, and e-business technologies. Salman's key focus is on WebSphere Process Choreographer, Web Services, Business Integration, and emerging open-source technologies. He holds a Bachelor of Engineering degree in Computer Science from Mississippi State University.

Raphael Mueller works as an IT Specialist within the WebSphere Lab-based Services department in the development laboratory of Boeblingen, Germany. He has been working with IBM for more than four years as a Software Engineer in the development team for DB2® Performance Expert. His areas of expertise include DB2, WebSphere Application Server, WebSphere Process Choreographer, J2EE technologies, and computer graphics. He holds a Diplom-Ingenieur degree in Computer Science from the University of Applied Sciences Bingen, Germany.

Sudhakar Nagarajan is an IBM WebSphere Certified Specialist, presently team leader for the WebSphere Globalization testing team under the Software group at RTP. Prior to joining the Software group, he was an IT specialist under Global Services, working with various clients. His background includes over ten years of application design, development and project management on both mainframe-based and distributed systems across a wide variety of industries and platforms. He holds a Master's degree in Manufacturing Engineering from REC Tiruchy, India.

Yu Zeng is a Senior I/T Specialist and certified administrator of WebSphere Application Server in the AP South Product Introduction Center. Yu Zeng has over six years of J2EE experience in the IT industry. In addition, he has in-depth

industry experience in Utility involving architecture design and J2EE implementation. He has worked in many WebSphere-related (WebSphere Application Server/WebSphere Portal Server/WebSphere Everyplace Access) beta projects in the Product Introduction Center and gained the know-how of sophisticated technology implementation, especially in J2EE and Java programming. He has coached many ISV/BPs in China to enable the development of many solutions based on IBM platforms.

Thanks to the following people for their contributions to this project:

Margaret Ticknor
Jeanne Tucker
Carla Sadtler
Martin Keen
Linda Robinson
Cecilia Bardy

International Technical Support Organization, Raleigh Center

Kent Below
Thomas Bernhardt
Jürgen Bönsch
Bernd Breier
Russell Butek
Mandy Chessell
Logan Colby
Alexander Dietzsch
Eric Herness
Joshy Joseph
Thomas Kasemir
Matthias Kloppmann
Alexander Koutsoumbos
Wolfgang Kulhanek
Kurt Lind
Fintan McElroy
Frank Neumann
Dr. Hans-Joachim Novak
Gerhard Pfau
Laurent Rieu
Stefan Ruettinger
Ruth Schilling
Joseph Sharpe
Jeff Stratford
Gerd Watmann
Gunnar Wilmsmann

Special thanks for their invaluable contribution to the redbook go to:

Hermann Akermann
Gunnar Wilmsmann

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an Internet note to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HZ8 Building 662
  P.O. Box 12195
  Research Triangle Park, NC 27709-2195

Part 1

Architecting a WebSphere Enterprise solution

1

Chapter 1

Positioning WebSphere Enterprise

This chapter is a high-level overview of the WebSphere Business Integration Server Foundation V5.1 product. It also positions the product in the IBM software palette.

1.1 Business challenges

Computer systems made their business debut in a supporting role, running administrative systems such as payroll. Over time, IT moved center-stage to underpin auxiliary processes such as stock control or production planning. This gradual movement experienced a big shift upon the emergence of the e-business boom of the mid-nineties, when many organizations went to market with a business model which placed IT systems firmly in the spotlight. Some of these pure-play businesses are still with us today, and many traditional businesses have transformed themselves in part or in whole into e-businesses.

Today, businesses of all sizes continue to increase their reliance on complex IT systems for their mainstream business functions. This trend is set to continue as the on demand world places more emphasis on rapid reconfiguration of the business processes to meet the changing demands of customers and increasing interdependencies between partner organizations.

In this business environment, those responsible for planning, developing and managing IT systems face a number of challenges.

- **IT systems and processes must enable change**

  The rapid pace of change is required by the changes in customer expectations and the business environment. IT systems should be responsive to these changes, not form a bottleneck in the process.

- **New applications must exploit existing assets**

  Organizations have invested significant amounts in their existing applications. This leaves a legacy of heterogeneous systems providing islands of function and data. The challenge is to integrate these islands to provide high-value business processes greater than the sum of their parts.

- **IT projects must generate an increasing return on investment**

  IT can influence the bottom line by providing significant benefits to core business processes. Coupling this value with a reduction in development and support costs dramatically increases the return on any IT investment. The challenge is to increase the efficiency of development projects without compromising quality.

WebSphere Business Integration Server Foundation V5.1 and WebSphere Studio Application Developer Integration Edition V5.1 provide an enterprise strength build and runtime environment. These products build on the industry leading WebSphere Application Server and WebSphere Studio Application Developer products, respectively, adding a number of features which enhance the capability of IT support organizations to meet the challenges.

In the following section, each of the challenges is discussed in turn, outlining the ways in which the WebSphere Business Integration Server Foundation and WebSphere Studio Application Developer Integration Edition products can enhance IT system delivery.

1.1.1 IT systems and processes must enable change

The on demand world requires that businesses react rapidly to a changing environment. This change may mean re-engineering business processes to take advantage of increased capabilities provided by partners or to improve the service provided to customers. The key theme is that IT organizations need to understand and mirror the requirements of the business.

This cannot be achieved in environments where dependencies between IT systems are so tightly coupled and business logic exists in so many places that changes in a component supporting one part of the enterprise cause knock-on effects in other components. In this situation, developing and testing a seemingly trivial change snowballs into a considerable effort, and IT processes frustrate the necessary business change.

Service oriented architecture

Service oriented architectures (SOA) enable flexible, modular applications to be constructed from heterogeneous systems. The key to this flexibility is the creation of coarse-grained building blocks known as *services*. These self-describing groupings of data and/or function can be combined and recombined in multiple configurations as business requirements change. The standard abstractions used within an SOA mean that the underlying programming language, operating system, geographical location and organizational ownership are not important. Web services are an example of an SOA.

The key interaction in a SOA is the binding of a service requester to a service provider. This may be supported by publication of the service description to a directory by the service provider, and a discovery of the service by the service requester.

WebSphere Application Server provides excellent support for Web services. WebSphere Business Integration Server Foundation adds to this capability by providing the powerful service composition and dynamic invocation capabilities of WebSphere Process Choreographer. This can be used as a service integrator to combine a series of diverse existing services as a single service which is of use in the modernized systems.

WebSphere Studio Application Developer Integration Edition provides visual tools to create business processes. This allows business users and developers

to collaborate when creating business processes, enabling rapid prototyping, development and deployment of code that fits the needs of the business.

More details can be found in:

- Chapter 6, “Process choreographer: introduction” on page 121
- Chapter 7, “Process choreographer: developing a simple process” on page 135
- Chapter 8, “Process choreographer: developing a complex process” on page 203

Simplifying code maintenance

One of the most common reasons for changing code is to update the business logic that it encapsulates, rather than the supporting business flow. Poorly structured code which does not distinguish between logic and control can significantly increase the time required to develop and test application changes. This can also lead to inconsistent application of business logic, where updates are made to only one part of the codebase.

WebSphere Enterprise provides a number of features which support management of business logic in a single place within the infrastructure. These not only impose a structure on the application but mean that maintenance becomes a configuration rather than a coding activity. Further, this configuration can be dynamically imported into a live application, reducing the need for extended outages to implement the code.

More details can be found in:

- Chapter 11, “Business Rule Beans” on page 331
- Chapter 19, “Dynamic Query” on page 495

1.1.2 New applications must exploit existing assets

A key trend in the last few years has been the continuing integration of systems within organizations. There is also a move to increase integration between partner organizations. In an ideal world, all of these platforms would be planned and designed with interoperability in mind. Unfortunately, systems have grown by adding functionality organically. The environment to be integrated may include functions provided by mainframe transaction handlers, packaged EPR systems, Web servers and even text-based consoles. To further complicate matters, each of these platforms may be owned and managed by different groups within organization or its partners.

An integration platform is required which handles the complexity of this task, offering a standard mechanism for connecting to and integrating the heterogeneous environment.

WebSphere Business Integration

WebSphere Business Integration is IBM's software platform for delivering integration solutions. The product suite currently includes the following process engines:

- WebSphere MQ Workflow

  Provides development, runtime and monitoring capabilities to manage long-running workflows which interact with systems and people.

- WebSphere InterChange Server

  Provides capabilities to construct business processes which combine existing function and data from other applications, including common ERP, CRM and financial packages.

- WebSphere Business Integration Message Broker

  Provides transformation and enrichment for in-flight messages, acting as an intermediary between applications which use different message structures and formats.

Each of these engines has its own particular strength. The engines can be combined with one another to cover the full range of business integration functionality.

WebSphere Business Integration Server Foundation V5.1 is the first release of the WebSphere Application Server which is named as part of the WebSphere Business Integration product suite. This reflects the advanced integration capabilities provided by such capabilities as WebSphere Process Choreographer.

WebSphere Process Choreographer adds a fourth engine to the product suite. At first glance, it appears to share capabilities with each of the three other engines:

- It supports long-running processes with staff interaction.
- It enables composite processes to be created and executed.
- It can operate on the content of messages.
- It can interoperate with the other three engines through the Web services interface.

Why then would it be selected as part of an architecture? A few arguments for using WebSphere Process Choreographer are given next.

- Built-in support for service orientation

  As discussed previously, the WebSphere Application Server base of WebSphere Process Choreographer provides native support for Web services. Any processes created in the tool can be exposed as services for consumption by service requesters. In addition, the dynamic invocation capabilities of the product mean that services can be invoked after locating the service during runtime.

- Fits naturally with J2EE environment

  Given that the process engine is based on a J2EE platform, it is an excellent fit for organizations that have invested in creating J2EE applications and now wish to deploy visually modeled business processes. WebSphere Process Choreographer adds a series of workflow capabilities which can dramatically improve developer productivity and application quality. The visual process editor means that business users can easily see the processes that are being created.

- Aligns to WebSphere Business Integration strategy

  WebSphere Process Choreographer is developed by the same team that creates the WebSphere MQ Workflow product. As such, it capitalizes on the strengths of both this product and WebSphere Application Server. It will also form a key element of the suite when the WebSphere Business Integration products move to a common architecture in the near future.

For more details, see also:

- Chapter 6, “Process choreographer: introduction” on page 121
- Chapter 7, “Process choreographer: developing a simple process” on page 135
- Chapter 8, “Process choreographer: developing a complex process” on page 203

You can also refer to the following Web site for more information:

`http://www.ibm.com/websphere/integration`

Application connectivity

As mentioned earlier, the heterogeneous nature of the environments means that significant time and effort could be required to connect them to a single integration server. Integration products use the concept of adapters to interface to applications and technologies. WebSphere Business Integration Adapters support a wide range of ERP, HR, CRM and supply chain systems, as well as technology adapters to many popular RDBMS, transaction handlers and operating systems.

WebSphere Business Integration Server Foundation provides support for these adapters. It also includes advances transactional capabilities to overcome shortcomings in the interfaces exposed by these services.

For more details, see also:

http://www.ibm.com/websphere/integration/wbiadapters/

1.1.3 IT products must generate an increasing return on investment

The challenge of providing higher-value applications is compounded by the drive to deliver them more cheaply and faster without compromising quality. To achieve this, the skills of highly trained developers should be applied to real business problems, rather than to creating common reusable components.

WebSphere Business Integration Server Foundation and WebSphere Studio Application Developer Integration Edition provide a number of advanced capabilities to enable IT organizations to be more efficient while continuing to deliver benefits.

Increasing development productivity

Delivery time is a prime concern for many IT organizations. One way to improve developer productivity is to capitalize on the advanced capabilities of modern middleware platforms. This is a double gain, as not only does the development team spend more time on the business problem but the costly and time-consuming ongoing maintenance of handcrafted solutions is also eliminated.

WebSphere Business Integration Server Foundation and WebSphere Studio Application Developer Integration Edition were designed to enhance the capabilities of development teams. These extensions to the underlying technology provide powerful yet simple capabilities to add value to applications.

Refer to the following chapters for further details of these capabilities:

- Chapter 10, “Common Event Infrastructure” on page 309
- Chapter 12, “Extended messaging” on page 353
- Chapter 18, “Shared Work Area service” on page 479
- Chapter 21, “Internationalization (i18n)” on page 539

Advanced middleware

While most applications can be delivered within the constraints of standards-based development frameworks, there are always situations that call for an extra degree of control over the runtime behavior of the system. The developer of such systems is faced with the decision of compromising the design

by adhering to the standard or diverging from the standard to create bespoke solutions which are not guaranteed to work under future versions.

The products provide support for advanced capabilities that build on concepts within the current version of the framework and are actively being promoted within the standards community. These provide an exceptional degree of flexibility to the developer, with the knowledge that they will be supported for future versions of the platform.

For further details, please refer to:

- Chapter 13, “Startup beans” on page 375
- Chapter 14, “Scheduler service” on page 391
- Chapter 15, “Asynchronous beans” on page 415
- Chapter 16, “Container Managed Persistence over Anything” on page 435

Performance optimization

Application efficiency can have a real impact on the cost of the runtime environment. While there is no replacement for good design and coding practices, WebSphere Business Integration Server Foundation does provide a number of capabilities to enhance the performance of the runtime environment.

For further details, refer to:

- Chapter 17, “Application profiling” on page 449
- Chapter 20, “Object pools” on page 523

2

Chapter 2.

Product overview

WebSphere Business Integration Server Foundation V5.1 and WebSphere Studio Application Developer Integration Edition V5.1 help companies to reduce IT complexity, reuse existing resources, and automate business processes through a powerful but simplified build-to-integrate framework.

This chapter will focus on these two products and provide definitions of the key technologies these products support. This chapter will discuss:

- Web services
- J2EE concepts
- Programming Model Extensions (PMEs)
- Business Process Execution Language for Web Services (BPEL4WS)
- Process Choreographer

2.1 Products

When IBM WebSphere Business Integration Server Foundation V5.1 is used in conjunction with WebSphere Studio Application Developer Integration Edition V5.1 for development, it can deliver a next-generation integration platform optimized for building and deploying composite applications that extend and integrate your existing IT assets.

This section will discuss the features of WebSphere Business Integration Foundation Server V5.1and WebSphere Studio Application Developer Integration Edition V5.1.

2.1.1 WebSphere Business Integration Server Foundation V5.1

WebSphere Business Integration Server Foundation V5.1 builds on the WebSphere Application Server to provide a premier Java 2 Enterprise Edition (J2EE) and Web Services technology-based application platform for deploying enterprise Web Services solutions for dynamic e-business on demand™.

It represents IBM's approach to building and deploying SOA-based applications that can adapt quickly and easily to change. It is designed to support the creation of reusable services (either new ones or those based on existing services, back-end systems, Java assets, and packaged applications). Services can then be combined to form both composite applications and business processes, which can further leverage business rules to make these applications and business processes adaptable.

WebSphere Business Integration Server Foundation V5.1 includes all of the features available in WebSphere Application Server Network Deployment V5.1, including:

- J2EE 1.3 support (support for some features planned for J2EE 1.4)
- Full XML support
- Full Web services support
- Support for private UDDI registries
- Web Services Gateway
- Database Connectivity
- Embedded HTTP server
- Web server plug-ins
- Authentication and authorization for secure access to Web resources
- Single sign-on and support for LDAP
- Java Message Service (JMS) support
- Dynamic caching
- IBM Tivoli® Performance Viewer
- Integration with third-party performance management tools

- Browser-based administration and workload management
- Intelligent workload distribution across a cluster
- Failure bypass
- Clustering support
- Migration support

Platforms

- Windows® 2000, 2003
- Solaris
- Linux® (Red Hat, Suse, United, etc.)
- HP-UX
- AIX®

For complete information on system requirements, see:

http://www-306.ibm.com/software/integration/wbisf/requirements/

2.1.2 IBM WebSphere Studio Application Developer Integration Edition V5.1

WebSphere Studio Application Developer Integration Edition V5.1 provides the tools you need to create, develop, test, and manage all of the resources involved in building Web and enterprise-scale J2EE and Web services applications. WebSphere Studio Application Developer offers creation tools, editors, wizards, templates, and code generators that help you rapidly develop J2EE resources such as HTML files, JSP pages, Java classes and servlets, EJB beans, and XML deployment descriptors. You can organize these resources into projects that correspond to modules defined in the J2EE specification. Once the resources have been created, you can easily test and debug them within the development environment, or export and test them on a remote server.

A major focus of this product is improving developer efficiency, as reflected in its existing functionality as well as new features introduced as part of V5.1. The product allows users to visually develop business processes and V5.1 updates this capability with a new business process designer and debugger that support the creation of process flows to conform to the BPEL 1.1 standard. V5.1 also includes a new editor for the Web Services Description Language (WSDL) that simplifies user interaction with the product and adds visual clarity to how the various components interact.

In summary, the major new features in WebSphere Studio Application Developer Integration Edition V5.1 are as follows:

- Business process designer for creating Business Process Execution Language for Web Services 1.1 (BPEL4WS) process flows
- Integrated visual BPEL debugger
- Enhanced performance for installing and debugging, including support for J9 Hot Swap
- New visual condition builder to direct the execution of BPEL processes
- Automated migration of process flows from Flow Definition Markup Language (FDML) to BPEL4WS

WebSphere Studio Application Developer Integration Edition also includes the following Programming Model Extensions (PMEs) that build on Java™ 2 Enterprise Edition (J2EE) standards to accelerate large-scale application development:

- Asynchronous beans
- Startup beans
- Last participant support
- Internationalization service
- Work areas
- Scheduler service
- Activity session services
- Dynamic query service
- WSGW Filters
- Object pools
- Container Managed Messaging
- Distributed Map
- Container Managed Persistence over anything
- Application profiling
- Back-up Cluster Support

Platforms

- Windows XP
- Windows 2000
- Windows NT®
- Linux (RedHat, Suse, etc.)

For complete information on system requirements, see:

http://www-306.ibm.com/software/integration/wsadie/requirements/

2.2 Key technologies

The following technologies are supported by WebSphere Business Integration Server Foundation and WebSphere Studio Application Developer Integration Edition V5.1 and are related to service oriented architectures.

2.2.1 Web services

Web services are a relatively new technology that has received wide acceptance as an important implementation of service oriented architecture. This is because Web services provide a distributed computing approach for integrating extremely heterogeneous applications over the Internet. The Web service specifications are completely independent of programming language, operating system, and hardware to promote loose coupling between the service consumer and provider. The technology is based on open technologies such as:

- eXtensible Markup Language (XML)
- Simple Object Access Protocol (SOAP)
- Universal Description, Discovery and Integration (UDDI)
- Web Services Description Language (WSDL)

Web services are described by WSDL files which contain:

- **Interface information:** the interface is described by a combination of port type, operations supported by the port type and input, output and error message structures.
- **Binding information:** the interface may be mapped to one or more concrete implementations in technologies such as EJB, Java, SOAP or JMS. The binding information describes the instantiation of the interface.
- **Service information:** each binding must be associated with a physical location where the implementation described in the binding executes. Examples include the location of an EJB, a Java class, an RPC router or JMS destination.

Note that there are more bindings possible than the combination of SOAP over HTTP that is usually associated with Web services. The flexibility of WSDL can be fully exploited in an SOA.

However, the different combinations of bindings and implementations could prove a significant challenge if the desired result is a flexible application. The *Java API for XML-based Remote Procedure Call* (JAX-RPC) and *Web Services Invocation Framework* (WSIF) standards were created to address this challenge. Each of these provides a standard API for invoking Web services from Java, although JAX-RPC does not specify how services can be invoked over

anything other than SOAP over HTTP. *Multiprotocol JAX-RPC* adds this support from WebSphere Application Server V5.1.1.

Both Multiprotocol JAX-RPC and WSIF are supported by the WebSphere Application Server base. WSIF is deprecated from V5.1.1, so Multiprotocol JAX-RPC should be used for new applications.

A detailed discussion of service oriented architectures and Web services standards is outside the scope of this book.

2.2.2 J2EE concepts

J2EE defines the standard for developing multi-tier enterprise applications in Java. J2EE simplifies enterprise applications by basing them on standardized modular components, by providing a complete set of services to those components, and by handling many details of application behavior automatically without complex programming.

WebSphere implements the J2EE platform and provides additional functionalities. WebSphere also adds new APIs and implements additional services for the J2EE platform in order to provide broader functionality or to bring future extensions into the product earlier.

The base WebSphere architecture gives flexibility to the system architecture level, not addressed in the J2EE specification, by separating the managed processes and implementing an overall administration system.

The J2EE application model divides enterprise applications into three fundamental parts: components, containers and connectors.

Components are the key functional elements developed for certain applications, *containers* are provided by system vendors, while *connectors* conceal complexity and promote portability.

2.2.3 PMEs

WebSphere Enterprise brings several Programming Model Extensions (PMEs) to the application server. The extensions are delivered in different forms, including services, APIs, wizards for development, and deployment extensions.

This book discusses how Programming Model Extensions can be used and provides sample scenarios to demonstrate how PMEs can be designed, implemented and maintained in an application. A brief description of each PME is shown in Table 2-1 on page 17.

Table 2-1 WebSphere Programming Model Extensions

Enterprise enablements	Description
Activity Session	Offers long-running transactions semantics without requiring XA support for all the resources involved in the unit of work, but without data integrity. You can keep EJBs active through multiple transactions and have them passivated at the end of the activity session.
Application Profiling, Access Intent	Allows you to define different Access Intents depending on the EJB client that is accessing a certain Entity EJB, allowing much more flexibility in the way you can tune data access. See Chapter 17, “Application profiling” on page 449 for more information.
Asynchronous beans	Allows J2EE applications (EJBs) to start other threads and transfer their J2EE context to those threads. See Chapter 15, “Asynchronous beans” on page 415 for more information.
Business Rule Beans (BRB)	Extends the scope of the WebSphere Application Server to support business applications that externalize their business rules. See Chapter 11, “Business Rule Beans” on page 331 for more information.
Common Event Infrastructure (CEI)	The Common Event Infrastructure provides the runtime environment to persistently store and retrieve events from many different programming environments. See Chapter 10, “Common Event Infrastructure” on page 309 for more information.
Container Managed Persistence over Anything (CMP/A)	Expands container-managed persistence to include the capability of persisting data to any back-end. CMP allows mapping CMP EJBs to back-ends beyond the traditional relational database. See Chapter 16, “Container Managed Persistence over Anything” on page 435 for more information.
Dynamic query	Extends J2EE EJB QL. You can formulate queries at runtime, select multiple EJB attributes out of a CMP EJB in a single SELECT clause, support for GROUP BY. See Chapter 19, “Dynamic Query” on page 495 for more information.
Extended Messaging (EMS)	Enhances standard J2EE Messaging by providing support for all types of messaging patterns, container support for these patterns, and code simplification. See Chapter 12, “Extended messaging” on page 353 for more information.

Enterprise enablements	Description
Internationalization (I18N)	Allows you to automatically recognize the calling client's time zone and location information so your application can act appropriately. This technology allows you to deliver to each user, around the world, the right date and time information, the appropriate currencies and languages, and the correct date and decimal formats. See Chapter 21, “Internationalization (i18n)” on page 539 for more information.
Last Participant Support	Allows J2EE application to have a single-phase resource (only one per transaction) in a transaction.
Object pools	Enables an application to avoid creating new Java objects repeatedly. See Chapter 20, “Object pools” on page 523 for more information.
Scheduler service	Allows a J2EE application to schedule the execution of tasks in the future. See Chapter 14, “Scheduler service” on page 391 for more information.
Shared work area (SWA)	Shared Work Areas provide a solution to pass and propagate contextual information between application components. For detailed information, refer to Chapter 18, “Shared Work Area service” on page 479.
Startup beans	A special kind of EJBs that are executed automatically when an application starts up or shuts down. See Chapter 13, “Startup beans” on page 375 for more information.

2.2.4 BPEL4WS

Business Process Execution Language for Web Services (BPEL4WS) is used to implement all business processes in WebSphere Business Integration Server Foundation V5.1. BPEL was originally proposed by IBM (in conjunction with BEA and Microsoft®) in July 2002 and combined ideas from IBM’s Web Services Flow Language (WSFL) and Microsoft’s XLANG. A technical committee within OASIS was formed in April 2003, giving BPEL a stronger standards “backbone.” The language (which may be more accurately described as a “meta language”) is used to define business process models by enabling the description of Web services operations, their relationships, and order of execution. It is intended to support the activities of system architects and software developers who are increasingly concentrating on taking the Web services created during an earlier phase of their SOA adoption and linking them together to construct workflows.

2.2.5 WebSphere Process Choreographer

WebSphere Process Choreographer provides support for business-process applications within the WebSphere Application Server. Process Choreographer supports service composition as specified by Business Process Execution Language for Web Services (BPEL4WS or abbreviated to BPEL) and enables developers to define the structure and behavior of a set of Web Services that jointly implement a business process.

Process Choreographer, also known as the business process container, is the process engine of WebSphere Application Server Enterprise. It allows developers to create processes using a visual tool which in turn helps to speed up application development.

The business processes that are implemented in an enterprise typically require a mixture of human and IT resources and these processes are supported by Process Choreographer. A process is a directed graph that starts with an Input node and ends with an Output node. A process itself is described in WSDL. Its input and output are described as WSDL messages.

A process can contain many activities. An activity can be the invocation of an EJB, a Java class, a service or another process. A process can also be event-driven. For example, it can be paused, waiting for an event, and then resumed when a message arrives.

3

Chapter 3.

Scenarios

This section outlines a number of ways that WebSphere Business Integration Server Foundation can be used within an IT solution. In each case, problems, solutions and benefits are outlined. Of course, the fictitious scenarios used here are far simpler than the potential reqirements seen in the real world.

For further detailed information of potential patterns of usage, please refer to the redbook *Patterns: Serial and Parallel Processes for Process Choreography and Workflow,* SG24-6306.

3.1 Scenario 1: Service composition

An insurance company has a significant legacy estate comprising stovepipe applications, each of which supports a different insurance product. This IT environment presents a number of issues.

- **Highly skilled staff is required**

  Company staff uses separate applications to work with each product and requires training in each of them. Some of the applications have highly specialized interfaces which require experienced staff to be allocated solely to their support.

  The company wishes to begin to consolidate these into a single application offering a simple user interface. This will free up the experienced staff for business development.

- **Product development is a slow process**

  Development of new insurance products composed from elements of existing products takes a long time. This affects the company's responsiveness to changes in the market. The company's market share position is threatened because its competitors are able to release products much faster.

  The company wishes to improve the ability of its business development team to create innovative products and release them to the market more quickly.

- **Customers have limited choices to purchase the products**

  At present, the only way of accessing the full range of the company's products is through the call center or insurance resellers. The reseller staff needs to attend a series of courses and install specific software to access the company's systems. A limited range of products are available on the Web.

  The company wishes to improve the customer experience by offering a full range of products through the Web and simplified integration options services to partners.

Solution

To achieve these aims, the company has invested in a service oriented architecture, which is shown in Figure 3-1 on page 23. The key points of the architecture are as follows.

- **Back-end applications are wrapped as Web services**

  These product-related services use WebSphere Business Integration Adapters and the CICS transaction gateway to expose functions and data from the existing systems.

- **Business services are constructed using these services**

  These business services are short-lived, supporting simple processes such as getting quotes, viewing customer insurance products and submitting insurance claims. The processes defined in WebSphere Business Integration Server Foundation contain the logic to route the data to the relevant systems.

- **New applications and channels use the business services**

  The company invests in its call center and Web infrastructures to capitalize on the new services. It also offers these services to insurance resellers for inclusion in their systems.

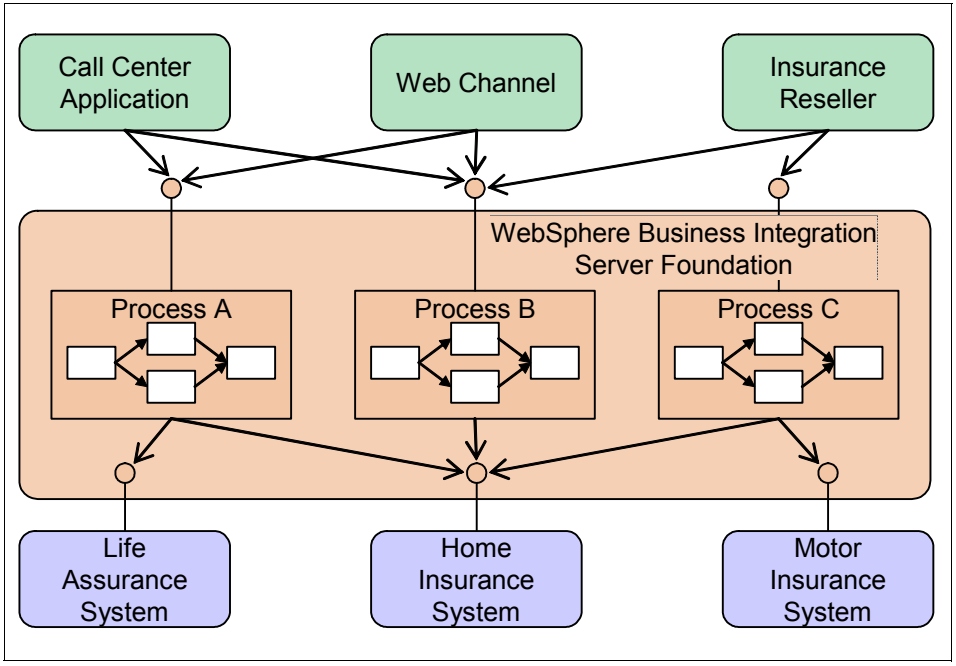


Figure 3-1 Service composition

Benefits

This approach offers a number of benefits:

- **Common interface to multiple back-end services**

  The technical complexities of the back-end implementations are hidden from the front-end applications.

  Functions and data are offered as services based on Web standards, and so should be supported by a wide range of packaged solutions.

- **Isolation of layers in the architecture**

  The insurance applications can be replaced with new applications which offer similar functionality without affecting the applications which consume the services.

  Similarly, new channels can be added to the front end without reworking the services.

- **Rapid prototyping of new services**

  New products can be offered by using the back-end services in different combinations.

  Visual editing of processes means that business users can assist with development of the services.

3.2 Scenario 2: Process state management

A telecommunications company is implementing a new order process for customers taking out a cellular phone contract. The company's requirements are outlined below.

- **The process must be flexible**

  The business process takes place using three distinct steps:

  a. The customer requests a quote and is provided with a quote reference number.

  b. The customer decides to proceed with the order on the basis of the quote. Contract documentation is printed and sent to customer.

  c. The customer returns a signed contract and payment details.

  The customer must be able to pause the process at any of these stages. There are, however, time limits beyond which the order is invalidated and the customer must request a new quote.

- **The process must be accessible from multiple channels**

  Part of the company's desire to implement the new process is to give the customer more ordering options. The company's channels include stores, a Web site and call centers. Customers should be able to start the order process from any of these channels and continue with the same order on another channel with little delay.

- **The system must enforce statutory requirements**

  The company is required to have a signed copy of the contract and payment details before proceeding to the setup of the customer's account. The

process must wait for this information to be received before completing the order.

Solution

The company has implemented a process engine as shown in Figure 3-2. The key points of the architecture are as follows.

- **The process is defined in BPEL4WS**

  A WebSphere Process Choreographer interruptible process is used to manage the state of the order. This allows the order state to be stored and retreived at a later date.

- **Front-end applications all access the same process**

  Once an order process has been started by a customer, the process instance can be accessed by the different channels. The process offers message-based interfaces to retrieve the process status and continue the process.

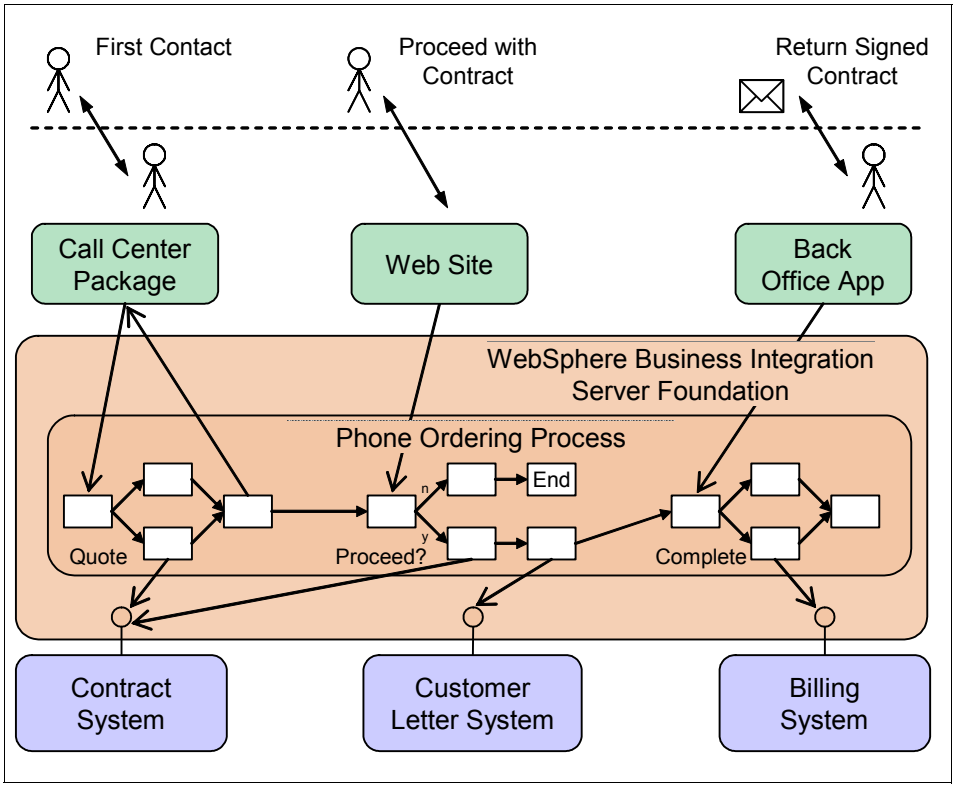


Figure 3-2 Process state management

Benefits

A number of additional benefits are provided with this solution.

- **The process is defined once**

  The channel applications do not contain any details of the implementation of the process, only the interfaces through which they need to pass data. This separation of process and user interface greatly aids future flexibility by allowing the process to change and new channels to be added.

- **Multiple channels can access the same process instance**

  The active process instances all reside in the same component. This means that any channel can retrieve and update the state of the process. In addition, it becomes possible to monitor the progress of the processes and, potentially, to produce statistics of interest to business users, such as *percentage of quotes converted to orders*.

- **Business-defined time-outs are enforced**

  Certain stages in the process have an associated time-out. As an example, a quote is valid for 30 days form the point that it is raised. WebSphere Business Integration Server Foundation provides capabilities to implement this time-out at the points in the process where an incoming message is awaited.

3.3 Scenario 3: Human interaction

A company operates a requisiton process to enable its staff to obtain a range of office supplies from pens and paper up to printers and photocopiers. This presents a number of challenges.

- **High-value items require management approval**

  Orders over a certain limit require approval from management. Delays in this process can reduce the productivity of the staff.

- **Received goods are often misplaced in the busy postroom**

  The company wishes to improve the quality of its post-receipt processes. Each received item should be associated with an order and dispatched to the correct department immediately. This process should also trigger the payment process.

Solution

The overal flow of the process is similar to that of the scenario outlined in 3.2, “Scenario 2: Process state management” on page 24. In this instance, however, the work is allocated to humans at two points in the process. This contrasts with the previous case, where the process waited for incoming messages.

- **Order approval**

  Management approval is required at this stage in the process. A WebSphere Process Choreographer Staff activity is used to assign this approval to appropriate members of the staff. Interaction is through a portal which links directly to the WebSphere Business Integration Server Foundation application.

- **Order receipt**

  The postroom staff is required to mark orders as received within the process. This is done through another portal, in which all outstanding orders are listed, allowing postroom staff to select the order and update the status.

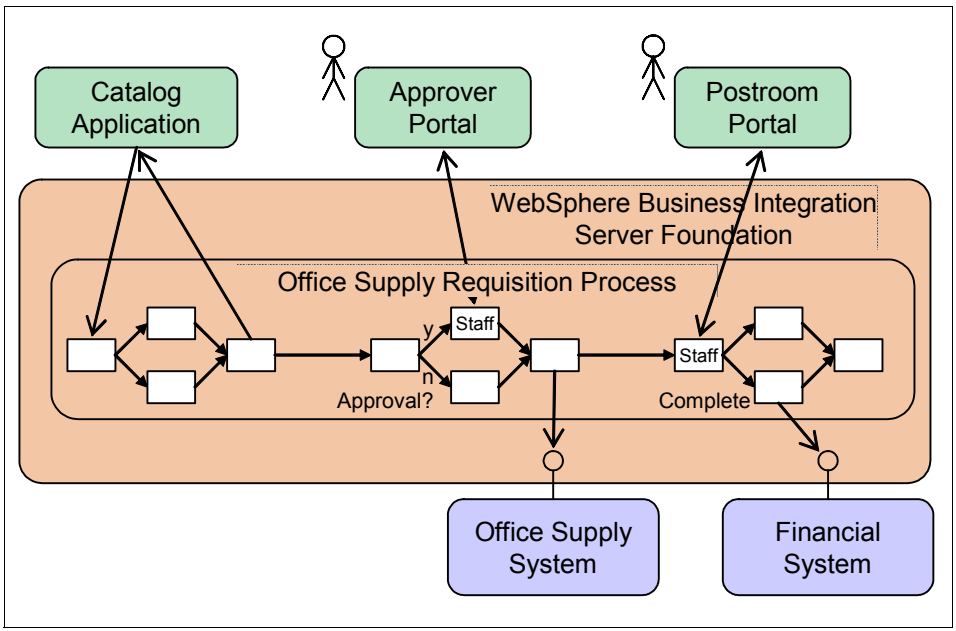


Figure 3-3 Human interaction

Benefits

The benefits in this instance are similar to those of the previous scenario outlined in 3.2, “Scenario 2: Process state management” on page 24, namely, tracking of orders, with potential management information extraction.

- **Humans can contribute to the process**

  The inclusion of staff activities in the process allows more complex decisions to be handled while still keeping overall state under the control of the process engine. The decisions that the humans handle in the process would be extremely difficult to codify, because they rely on management discretion.

Part 2

Setting up the environment

4

Runtime environment

This chapter provides an overview of the runtime architecture of WebSphere Business Integration Server Foundation. The features of the product are described.

This chapter also provides an explanation of the procedures for installing, configuring, and verifying WebSphere Business Integration Server Foundation V5.1.

Additionally, the chapter concentrates on the decisions and tasks associated with WebSphere Business Integration Server Foundation V5.1 from the operational and administration points of view.

4.1 Architecture

Figure 4-1 shows the basic architecture of WebSphere Business Integration Server Foundation.

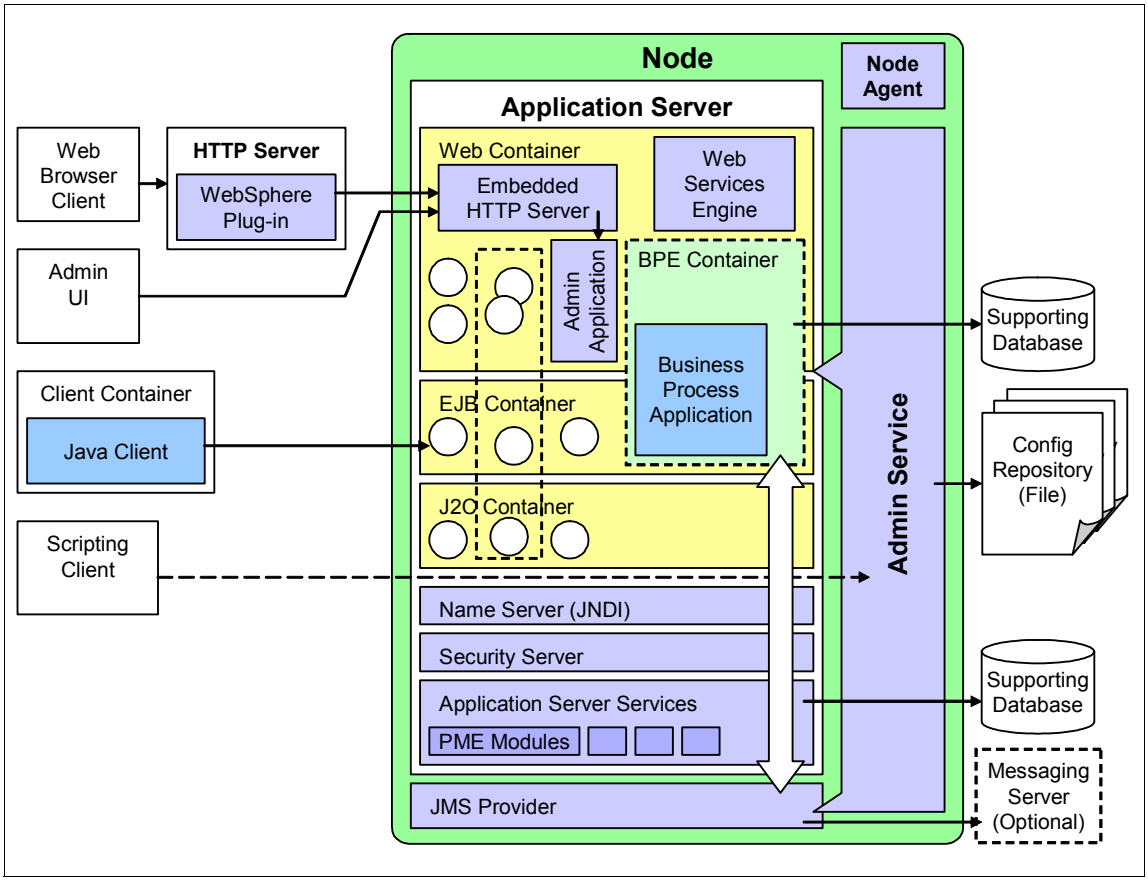


Figure 4-1 WebSphere Business Integration Server Foundation V5.1 architecture

WebSphere Business Integration Server Foundation is based on the capabilities provided by WebSphere Application Server base and Network Deployment editions. These capabilities are summarized in the next section, with subsequent sections covering the specific components provided by WebSphere Business Integration Server Foundation.

4.1.1 WebSphere Application Server base components

This section summarizes the components provided by the WebSphere Application Server base and Network Deployment editions. For a detailed discussion of these topics, please refer to the redbook *IBM WebSphere Application Server V5.1 System Management and Configuration WebSphere Handbook Series*, SG24-6195.

Node and node agent

A *node* is a logical grouping of WebSphere-managed server processes that share common configuration and operational control. A node is generally associated with one physical installation of WebSphere Application Server.

Under the more advanced configurations of WebSphere Application Server, multiple nodes can be managed from a single administration server and the nodes can collaborate to share the workload amongst themselves. In these centralized management configurations, each node has a *node agent* that works with a Deployment Manager to manage administration processes.

Servers

Servers perform the actual code execution. There are several types of servers, depending on the configuration. Each server runs in its own JVM.

- Application servers

  The application server is the primary runtime component in all configurations. This is where the application actually executes. All WebSphere Application Server configurations can have one or more application servers. In the Network Deployment configuration which WebSphere Business Integration Server Foundation can build on, multiple application servers are maintained from a central administration point. In addition, application servers can be clustered for workload distribution.

- JMS servers

  WebSphere Application Server provides an embedded JMS server for messaging support. In the Base configuration, the JMS server functions are integrated into the application server. In the Network Deployment and WebSphere Business Integration Server Foundation configurations, the JMS server runs in a separate JVM. There is one JMS server per node.

  The JMS server can be replaced by another JMS provider, such as WebSphere MQ.

Containers

The J2EE 1.3 specification defines the concept of containers to provide runtime support for applications. There are two containers in the application server implementation:

- Web container

  The Web container processes servlets, JSP files and other types of server-side includes. Each application server runtime has one logical Web container, which can be modified, but not created or removed.

  The Web container provides an embedded HTTP server, although it is unlikely that this would be used to handle incoming requests in a real implementation.

- EJB container

  The EJB container provides all the runtime services needed to deploy and manage enterprise beans. It is a server process that handles requests for both session and entity beans.

  EJBs do not communicate directly with the server, instead using the interfaces provided by the EJB container. The container provides capabilities such as threading, transaction support and data management.

  In addition, there is an application client container which can run on the client machine.

- J2C container

  This container provides services enabling the use of J2EE Connector Architecture. This allows applications executing in WebSphere Application Server to communicate with existing systems such as CICS and popular ERP systems via a simplified API.

HTTP server with WebSphere plug-in

Although the Web container has an embedded HTTP server, a more likely scenario is that an external Web server will be used to receive client requests. The Web server can serve requests that do not require any dynamic content, for example, HTML pages. However, when a request requires dynamic content (JSP/servlet processing), it must be forwarded to WebSphere Application Server for handling.

This is achieved using the Web server plug-in. The plug-in is included with the WebSphere Application Server package for installation on a Web server. The plug-in can use HTTP or HTTPs to transmit the request from the HTTP server to the WebSphere Application Server Web container.

Application server services

WebSphere Business Integration Server Foundation provides a number of application services to assist with creating enterprise-strength applications.

Transaction service

WebSphere applications can use transactions to coordinate multiple updates to resources as one unit of work such that all or none of the updates are made permanent. Transactions are started and ended by applications or the container in which the applications are deployed.

WebSphere Application Server is a transaction manager that supports coordination of resource managers through their XAResource interface and participates in distributed global transactions with other OTS 1.2 compliant transaction managers (for example, J2EE 1.3 application servers).

JNDI

Each application server hosts a name service that provides a Java Naming and Directory Interface (JNDI) name space. The service is used to register resources hosted by the application server.

Security service

Each application server JVM hosts a security service that uses the security settings held in the configuration repository to provide authentication and authorization functionality.

4.1.2 Business Process Execution container

The Business Process Execution container is a specialized J2EE application which executes business processes and flows. It handles the lifecycle of a process, from instantiation of a process template to final deletion of the completed process.

It relies on a database to manage state and on a JMS provider for transitions between activities. This means that to make a fully scalable, available installation of WebSphere Process Choreographer, the characteristics of the underlying infrastructure must also be considered.

4.1.3 Programming Model Extensions

WebSphere Business Integration Server Foundation provides a number of valuable extensions to the J2EE specification. These are delivered in many different forms, including services, APIs and tooling extensions. The remainder of the book is dedicated in a large part to these Programming Model Extensions (PMEs).

4.2 Basic configuration

The following section outlines the method for configuring WebSphere Business Integration Server Foundation on a single Microsoft Windows server.

4.2.1 Planning

The products that will be installed on the single-server implementation are:

- IBM HTTP Server V1.3
- WebSphere Business Integration Server Foundation V5.1
- DB2 Enterprise Server Edition V8.1
- WebSphere MQ V5.3, in a central configuration
- Tivoli Directory Server V5.2

To enable support for WebSphere Process Choreographer, the following WebSphere Business Integration Server Foundation components must be configured:

- WebSphere MQ Adapter for BPE
- DB2 Enterprise Server Edition
- (Optional) Staff plug-in. Required if staff interaction is to be employed.

The WebSphere Application Server Network Deployment components will not be configured in the basic configuration. This component is not required because the node is not part of a distributed configuration.

4.2.2 Software requirements

Review the operating system requirements for further details:

http://www.ibm.com/software/integration/wbisf/reqirements/

Our operating system installation was Microsoft Windows 2000 Server with SP4.

4.2.3 Installation

The order of installation of the WebSphere components is as follows:

1. WebSphere MQ V5.3
2. DB2 Enterprise Server Edition V8.1
3. Tivoli Directory Server V5.2

Supporting components

Installation details are given in Appendix B, “Additional configuration help” on page 577. Refer to the following topics for instructions.

- “WebSphere MQ setup instructions” on page 578
- “DB2 Enterprise Server Edition V8.1 installation” on page 579
- “Tivoli Directory Server V5.2 installation” on page 580

> **Note:** When installing Tivoli Directory Server on the same machine as WebSphere Business Integration Server Foundation V5.1, do not install the Embedded WebSphere Application Server V5.0 Express server.
>
> In the new version of Tivoli Directory Server, the administration is done via a thin client application that must be run from WebSphere Application Server. Since we already have WebSphere Business Integration Server Foundation installed on the single-server machine, we don’t need an extra Express V5.0 installation.
>
> The J2EE application called IDSWebApp.war, which is located in Tivoli Directory Server install_home, must be installed in WebSphere Business Integration Server Foundation.

WebSphere Business Integration Server Foundation

Once the prerequisites have been installed, continue with the installation of WebSphere Business Integration Server Foundation. The WebSphere Business Integration Server Foundation Installer performs the following tasks.

1. Check prerequisites.
2. Search for WebSphere Application Server family products, V4.x and V5.0.x.
3. Install WebSphere Application Server V5.1 base if not already installed, or add required features to an existing WebSphere Application Server base V5.1.0.
4. Install cumulative fix 5.1.0.2 to a base or Network Deployment node.
5. Install 5.1.0 cumulative fix 1 for the Java SDK that V5.1.0.x WebSphere Application Server products use.
6. Install WebSphere Business Integration Server Foundation product extensions.

The steps to install WebSphere Business Integration Server Foundation are given below:

1. Use launchpad.bat to start the installer.

2. Click **Install the product** to start the WebSphere Business Integration Server Foundation V5.1 installation.
3. Accept the license agreement.
4. The tool starts the prerequisite checker. This tool is built into the installer program. Verify that all prerequisites are met.

   Note: The prerequisite checker creates log files in the C:\Document and Settings\Administrator\Local Settings\Temp\1 folder, which can be used to troubleshoot installation prerequisite failures.

5. Once the prerequisite checker has completed, continue by selecting a **Custom** installation.
6. Select all components, including CEI. Deselect Samples for all components. Also uncheck CORBA C++ SDK support, Javadocs, and Embedded messaging.
7. Install the product in the directory C:\WebSphere\AppServer.
8. Accept the default node name and the hostname.
9. Run the application server as a service. For this, you will be required to provide an administrator's user ID and password.
10. Review the summary and start the installation.

Although the Web container provides an embedded HTTP server, robust Web applications generally require a separate Web server component. The default installation of WebSphere Business Integration Server Foundation does not include IBM HTTP Server or the Web server plug-in. Additionally, Tivoli Performance and Analysis tools are useful tools to install. Follow the instructions in "IBM HTTP Server, IBM HTTP Web server plug-in, and Tivoli Performance Viewer installation" on page 582 to install these components.

Verifying the installation

We will run the Installation Verification tool and some other tests to ensure that the application server has been installed successfully.

1. Run the FirstStep.bat utility
2. Click **Verify Installation**. This step should automatically launch the server process. Check that there are no errors in the output, that the message `Installation Verification Complete` is shown and that all status are marked as `PASSED`.
3. Click **Start the Server** and check to see if there are any errors in the log.
4. Click **Stop the Server** and check to see if there are any errors in the log.

5. Click **Administrative Console** to connect to the admin console and log in as `admin` to verify admin console functionality.

Installing Cumulative Fix 3 and interim fixes

At the time of writing, Cumulative Fix 3 was the latest available for WebSphere Business Integration Server Foundation V5.1. It is recommended that this fix and any other interim fixes be applied, since they address several key issues.

Fixes are available from the WebSphere Business Integration Server Foundation support site:

`http://www.ibm.com/software/integration/wbisf/support/`

1. Open the WebSphere Business Integration Server Foundation support site above; navigate to Recommended Updates and open WAS Base/ND 5.1.0.3 Cumulative Fix.
2. Download the Windows - base package. The file is called was510_cf3_win.zip.
3. On the command prompt, set the JAVA_HOME variable to point to the <WBISF_root>/java folder.
4. Extract this file into a separate folder. It will create the was510_cf3_win folder.

> **Note:** Cumulative Fix 3 also includes the updateInstaller listed on the Web site, so this does not need to be downloaded separately.

5. Run updateWizard.bat from the was510_cf3_win folder.
6. Follow the on-screen instructions to install the FixPack.
7. Select **install fixpack**. Accept the default location for the FixPack folder.
8. Follow the remaining instructions to finish installing Cumulative Fix 3.

A number of interim fixes are also available from the WebSphere Business Integration Server Foundation support site. These are named PQ*nnnn*.

1. Open the WebSphere Business Integration Server Foundation support site and navigate to Recommended Updates.
2. Download all interim fixes.
3. Extract the interim fixes into the was510_cf3_win folder. This will create the efixes directory.
4. On the command prompt, set the JAVA_HOME variable to point to the <WBISF_root>\java folder.
5. Ru**n** updateWizard.bat.

6. Accept the default language (English).
7. Follow the on-screen instructions, making sure that **IBM WebSphere Application Server V5.1.0.3** is selected and that the correct application server installation directory is selected.
8. Select **Install Fixes** in the next window. This is an efix install, rather than a FixPack install.
9. Select the directory where efixes were copied above.
10. The wizard will scan for the efixes in the given folder. Select all all listed fixes, checking that the wizard lists all the fixes that were downloaded.
11. Review the summary and click **Finish** to install the fixes.

4.2.4 Configuration

The tasks included in this section entail having different components of WebSphere Business Integration Server Foundation V5.1 configured and working properly. This section focuses on the product configuration and integration. These WebSphere Business Integration Server Foundation components include:

1. IBM HTTP Server
2. BPE components:
   a. WebSphere MQ configuration
   b. DB2 configuration
   c. Tivoli Directory Server Server for Staff plug-in and Global Security
3. BPE container configuration

IBM HTTP Server

In order to configure IBM HTTP Server, verify that the following WebSphere Business Integration Server Foundation plug-in information is available in the IBM HTTP Server configuration file.

1. Edit <IHS_root>/conf/httpd.conf.
2. Search for for the `ServerName` directive and verify that it is set properly.
3. Go to the end of the file and verify that it contains the following lines.

Example 4-1 httpd configuration

```
LoadModule ibm_app_server_http_module
"C:\WebSphere\AppServer\bin/mod_ibm_app_server_http.dll"
WebSpherePluginConfig "C:\WebSphere\AppServer/config/cells/plugin-cfg.xml"
```

BPE supporting components

Configuring BPE container involves several steps:

1. Set up WebSphere MQ Queue Managers and appropriate queues.
2. Set up the DB2 database for BPE container.
3. (Optional) Configure the Staff plug-in to utilize Tivoli Directory Server.

The high-level task plan for configuring the BPE container is given in the following InfoCenter topic; click **WebSphere Business Integration Server Foundation → Administering → Applications → Process choreographer → Configuring the business process container**.

WebSphere MQ

Refer to the InfoCenter for configuration of WebSphere MQ. Navigate to **WebSphere Business Integration Server Foundation → Administering → Applications → Process choreographer → Configuring the business process container → Creating the queue manager and queues for the business process container**. At a high level, the following steps must be completed as documented in the InfoCenter:

1. Create Queue Manager.

   Important: During tests, we used WBISFQM1 and ran into the following MQJMS exception:

   ```
   MQJMS2005: failed to create MQQueueManager for '<queue manager>'
   ```

   Make sure you have a TCP Listener and a Server Connection Channel; if either of these is missing, create it.

2. Create WebSphere MQ Cluster as needed (optional).
3. Create the queues.
4. Add the listener for the Queue Manager (optional).

IBM DB2 setup

The database must be configured prior to using the Business Process container. In the InfoCenter, navigate to **WebSphere Business Integration Server Foundation → Administering → Applications → Process choreographer → Configuring the business process container → Creating the database for the business process container**. A summary of the tasks that must be performed to configure DB2 is as follows:

1. Ensure that DB2 is set up and installed properly as per DB2 Installation Guidelines in the DB2 product documentation
2. In order to avoid deadlocks, make sure `db2set DB2_RR_TO_RS=YES` is set.

3. Verify that a default DB2 instance is created, `db2inst1`.
4. Create a new database called BPEDB for Business Process Container, and the schema for the database. This can be done using the *createDatabaseDb2.ddl* script.
5. Verify there are no errors from the script, then bind certain CLI packages to the database as follows.

Example 4-2 Bindings for the database

```
db2 connect to BPEDB
db2 bind %DB2PATH%\bnd\@db2cli.lst blocking all grant public
```

Note: Note that createDatabaseDb2.ddl creates a database and sample tables as needed. This script should be used only for creating test and development databases. For production BPE setup, it is recommended that you relocate the tables into different table spaces using the createSchemaDb2.ddl script. Also, adjusting DB2 tuning parameters always helps from the performance standpoint.

Tivoli Directory Server setup for Staff plug-in

Tivoli Directory Server must be configured before the WebSphere Process Choreographer Staff plug-in can be used. Review the contents of the Tivoli Directory Server installation guide prior to proceeding with the Staff plug-in configuration.

The Tivoli Directory Server must be installed and configured properly with an appropriate base DN. In our tests, we used `ou=itso, o=ibm, c=us`.

Important: By default, Tivoli Directory Server installs an instance of Embedded WebSphere Application Server Express V5.0.2 to provide administrative support. The Tivoli Directory Server administration application will not run on the WebSphere Business Integration Server Foundation V5.1 application server.

If the Embedded WebSphere Application Server V5.0.2 and WebSphere Business Integration Server Foundation V5.1 must coexist on the same machine, conflicts between TCP/IP ports must be resolved.

Detailed configuration steps can be found in “Tivoli Directory Server V5.2 installation” on page 580.

BPE container

The next task is to install and configure the BPE container in WebSphere Business Integration Server Foundation V5.1.

Installing Business Process Container

The BPE container will be set up to communicate with the WebSphere MQ and DB2 Enterprise Server Edition components. Tivoli Directory Server is optionally configured for Staff plug-in support in the BPE container.

Review the instructions in the appropriate InfoCenter topic by clicking **WebSphere Business Integration Server Foundation** → **Administering** → **Applications** → **Process Choreographer** → **Configuring the business process container** → **Using the Install Wizard to configure the business process container**.

A high-level overview of the guidelines is given below.

1. Log in as Administrator and start WebSphere Business Integration Server Foundation V5.1 application server.
2. Open the Administrative Console.
3. Define environment variables under **Environment** → **Manage WebSphere Variables**.

Table 4-1 Environment variables

Variable name	Variable value
MQ_INSTALL_ROOT	C:\WebSphere\WMQ
DB2_JDBC_DRIVER_PATH	C:\WebSphere\SQLLIB\java
DB2UNIVERSAL_JDBC_DRIVER_PATH	C:\WebSphere\SQLLIB\java

4. For WebSphere MQ JMS Provider, use the following settings as documented in Table 4-2.

Table 4-2 JMS Provider settings

Setting	Value
JMS Providers	WebSphere MQ JMS Provider
Queue Manager	QM1
JMS Security Role	Administrators
JMS API User ID	Administrator
JMS API Password	password

5. JMS Resoruces should be created from scratch as per documented instructions.
6. Select the check box **Check this box to install the Web client**.
7. Review summary information and click **Finish** to install the Business Process container.

Configuring Staff service for Process Choreographer

Process choreographer uses Staff plug-ins to determine who can start a process or claim an activity. Your business processes can also use the Staff plug-in services to resolve staff queries. Each type of directory service requires a different staff plug-in. You can register multiple staff plug-ins. The user registry and system plug-ins are already installed and can be used without any configuration. To configure a Staff Plugin Provider, consult the following InfoCenter topic; click **WebSphere Business Integration Server Foundation** → **Administering** → **Applications** → **Process choreographer** → **Configuring the staff service for process choreographer**.

A J2C Authentication alias must be defined first in WebSphere Business Integration Server Foundation V5.1 Security section. Navigate to **Security** → **JAAS Configuration** → **J2C Authentication Data**. and click **New** to define a new authentication alias, as follows. If this alias is not set, an anonymous logon to the LDAP server is used.

Table 4-3 J2C authentication alias

Setting	Value
Alias	<hostname>/tds
User ID	cn=root
Password	tdsadmin
Description	TDS Authentication Alias

Define a new Staff plug-in configuration as in the InfoCenter item mentioned above.

Configure the Staff plug-in at the node level. Configure the customer properties for the Staff Plugin Provider as given in Table 4-4 on page 45.

Table 4-4 Staff plug-in configuration parameters

Setting	Value
AuthenticationAlias	<hostname>/tds
AuthenticationType	<optional>
BaseDN	cu=itso,o=ibm,c=us
ContextFactory	com.sun.jndi.ldap.LdapCtxFactory
ProviderURL	ldap://<hostname>:389
SearchScope	subtreeScope

4.3 Distributed configuration

The purpose of this section is to show how to configure WebSphere Business Integration Server Foundation components in a multi-tier environment. All components will be installed on separate machines. However, this section will not address any WLM and HA issues. Procedures for testing the configurations using sample applications are also not covered in this chapter. However, in a production environment, it is imperative to test the distributed, high availability workload management architecture with a sample application.

4.3.1 Planning

It is assumed that the following components are already installed on separate physical nodes

- IBM DB2 UDB V8.1 Enterprise Server Edition
- WebSphere MQ V5.3 Server
- Tivoli Directory Server V5.2

The highlighted components will be installed and configured in this section.

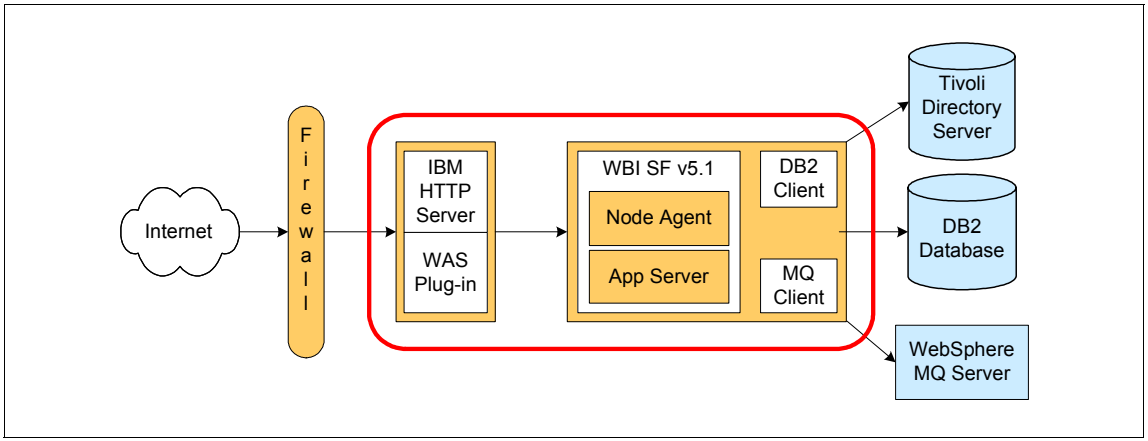


Figure 4-2 The distributed configuration infrastructure

Components installed in the distributed configuration will include:

- WebSphere Business Integration Server Foundation
- IBM HTTP Server
- IBM DB2 Client
- WebSphere MQ Client

4.3.2 Software requirements

The installation depends on the software listed below.

Operating system software requirements

This section outlines the requirements for the operating system.

1. Ensure that the Operating system is at the correct level.

   Table 4-5 outlines the hardware and operating system configuration used in setting up the distributed environment.

Table 4-5 Product - machine mapping

WebSphere Business Integration Server Foundation Components	Machine Type	OS Level
WebSphere Business Integration Server Foundation Node 1	pSeries® 44P 170	AIX 5.2 PTF ML5200-02
WebSphere Business Integration Server Foundation Network Deployment	pSeries 43P 150	AIX 5.2 PTF ML5200-02

WebSphere Business Integration Server Foundation Components	Machine Type	OS Level
IBM DB2 V8.1 ESE	pSeries 44P 170	AIX 5.2 PTF ML5200-02
WebSphere MQ	pSeries 43P 150	AIX 5.2 PTF ML5200-02
Tivoli Directory Server	xSeries® NF5100	Windows 2k Server SP4

In addition to the PTF ML5200-02, the following critical AIX fixes were applied: U488962, U488959, U488958, U488923, U488917. Table 4-6 lists all the updates that must be installed on all physical nodes of the distributed enviornment.

Table 4-6 AIX package updates

AIX Package	Package Level
bos.mp	5.2.0.15
bos.mp64	5.2.0.15
bos.up	5.2.0.15
devices.chrp.base.rte	5.2.0.14
devices.common.IBM.ethernet.rte	5.2.0.14
X11.fnt.ucs.ttf_KR	5.2.0.0
X11.fnt.ucs.ttf_TW	5.2.0.0
X11.fnt.ucs.ttf_CN	5.2.0.0
X11.fnt.ucs.ttf	5.2.0.0

2. Check that sufficient disk space is available in various file systems:
   - /usr: 4 GB
   - /var: 1 GB
   - /tmp: 2 GB

   You may also like to create logical volumes mounted under these locations to contain the WebSphere Business Integration Server Foundation installation.
3. You must install the xlC.rte 6.0 run-time code, which is a prerequisite of GSKit7. This is a prerequisite for WebSphere Business Integration Server Foundation installation.
4. Download the latest AIX fixes for AIX 5.2 from the Fix Central Web site:

   https://techsupport.services.ibm.com/server/aix.fdc

5. Verify that there is DNS name resolution configured on the operating system. All nodes should be reachable from each other using both short and fully qualified hostnames. A good test to perform is to ping both short and FQHN for all nodes.

WebSphere MQ

A WebSphere MQ server installation is required to support the BPE container.

For more information about software requirements for WebSphere MQ V5.3 on AIX, refer to:

http://www.ibm.com/software/integration/mqfamily/platforms/supported/wsmq_for_aix_5_3.html

For other platforms and products, refer to:

http://www.ibm.com/software/integration/websphere/mqplatforms/supported.html

IBM DB2 ESE Server

A database server is required to support the BPE container.

For more information about software requirements for IBM DB2 UDB ESE, refer to:

http://www-306.ibm.com/software/data/db2/udb/sysreqs.html

Tivoli Directory Server

For more information about software requirements for Tivoli Directory Server, refer to:

http://www.ibm.com/software/tivoli/products/directory-server/platforms.html

4.3.3 Installation

This section details the installation instructions for WebSphere Business Integration Server Foundation V5.1 on the AIX platform.

WebSphere MQ Client

Installation instructions for WebSphere MQ are provided in the platform-specific manual at the following Web site:

http://www.ibm.com/software/ts/mqseries/library/manualsa/index.htm

A summary of the instructions for installing the WebSphere MQ Client on AIX is given below:

1. Log in as an administrator user (root).
2. Insert the WebSphere MQ Client software CD.
3. Run the **`smitty install_latest`** command to install the software.
4. Specify the CD mount point where WebSphere MQ Client software is located and follow the on-screen instructions.
5. Select the components **mqm.base**, **mqm.client**, and **mqm.java**.
6. Verify that *Accept new license agreements* is set to `yes` and *Preview new license agreement* is set to `no`.

CSD05

The CSD05 FixPack must be installed for the WebSphere MQ client.

1. Download WebSphere MQ CSD5 from the WebSphere MQ Support site and extract it to a temporary folder.
2. Use **`smitty`** to install CSD05: **`smitty update_all`**. Specify the temporary directory where CD05 was extracted in the previous step.
3. Verify that *Accept new license agreements* is set to `yes` and *Preview new Licensen agreement* is set to `no`.

IBM DB2 UDB Client

For more information about IBM DB2 UDB, refer to:

`http://publib.boulder.ibm.com/infocenter/db2help/index.jsp`

The instructions below are provided as a guideline. You must refer to the DB2 InfoCenter above for detailed instructions.

1. Log in using an administrator user ID (root).
2. Insert the DB2 Software CD.
3. Locate the db2setup script and execute **`./db2setup`**.
4. Select **Install Products**.
5. In the next window, select **DB2 Administration Client**.
6. The DB2 setup wizard will start. Accept the license, then select **Custom installation** method.
7. Verify that **Install DB2 Administration Client on this computer** is selected.
8. In the Features window, accept the detaults.

9. In the Languages window, accept the defaults (if there is a need, install any additional languages).
10. In the DB2 setup instance window, be sure to select **Create a DB2 instance**.
11. Be sure to select **New user**, and set the DB2 instance owner to the same value as the DB2 server. For our tests, the DB2 server instance owner was set to `db2admin`.
12. Review the summary information and click **Finish** to install the DB2 client.
13. At the end of the setup, verify that the report contains all `SUCCESS` status.
14. If there is a problem with the installation, uninstall DB2 client using **`db2_deinstall`** and re-install.

FixPack 5

FixPack 5 must also be installed on this node.

1. Log in as an administrator user (root).
2. Download the FixPack from the DB2 support site, and extract to a temporary folder.
3. Change to the temporary folder where DB2 FixPack was extracted.
4. Set the execute permission on the installFixPack script using the following command:

   ```
   chmod a+x installFixPackexecute
   ```

5. Verify that the script displays a `SUCCESS` status for all installed DB2 client components.

WebSphere Business Integration Server Foundation

This topic describes how to install WebSphere Business Integration Server Foundation as the root user on an AIX operating system platform. Detailed instructions on how to set up WebSphere Business Integration Server Foundation V5.1 are given in the following InfoCenter topic. Click **WebSphere Business Integration Server Foundation** → **Installing** → **Getting started** → **Installing the product** → **Installing the Integration Server on AIX platforms**.

The following guidelines should be used in addition to the above instructions:

1. Mount CD1, which contains the WebSphere Business Integration Server Foundation installation image.
2. Change to the mounted CD-ROM directory.
3. Issue **`launchpad.sh`** to bring up the installation wizard.
4. Click the **Install the product** to start the WebSphere Business Integration Server Foundation V5.1 installation.

5. Accept the license agreement.
6. The tool starts the prerequisite checker. This tool is built in to the installer program. Verify that all prerequisites are met.
7. Choose a **Custom** installation.
8. Select all components, including CEI. Deselect Samples for all compoenents. Also uncheck CORBA C++ SDK support, Javadocs, and Embedded messaging.
9. Install under /usr/WebSphere/AppServer.
10. Accept the default node name and the hostname.
11. Review the summary and start the installation.

> **Note:** The InfoCenter lists four installation methods. Method 1 is recommended; Method 4 proved problematic during tests.

12. If the Install Verification utility starts automatically after the WebSphere Business Integration Server Foundation installation finishes, close the window and exit the Product Installation window as well. We will be addressing the IVT and configuration in 4.3.4, “Configuration” on page 54.

Installing Cumulative Fix 3 and interim fixes

At the time of writing, Cumulative Fix 3 was the latest available for WebSphere Business Integration Server Foundation V5.1. It is recommended that this fix and any other interim fixes be applied since they address several key issues.

Fixes are available from the WebSphere Business Integration Server Foundation support site:

`http://www.ibm.com/software/integration/wbisf/support/`

1. Open the WebSphere Business Integration Server Foundation support site, then navigate to Recommended Updates and click **WAS Base/ND 5.1.0.3 Cumulative Fix**.
2. Download the **AIX - base** package. The file is called was510_cf3_aix.zip.
3. On the command prompt, set the JAVA_HOME variable to point to the <WBISF_root>/java folder.
4. Extract the fix file into the was510_cf3_aix folder.

> **Note:** Extracting this file also creates files necessary to run the *updateInstaller* wizard.

5. Run **`updateWizard.sh`** from the was512_cf3_aix folder.

6. Accept the default language (English).
7. Follow the on-screen instructions, making sure that **IBM WebSphere Application Server V5.1.0.2** is selected and that the correct application server installation directory is also selected.
8. Select **Install FixPack** in the next window. Accept the default location for the FixPack folder.
9. Follow the remaining on-screen instructions to finish installing Cumulative Fix 3.

A number of interim fixes are also available from the WebSphere Business Integration Server Foundation support site. These are named PQ*nnnn*.

1. Open the WebSphere Business Integration Server Foundation support site and click **Recommended Updates**.
2. Download all interim fixes.
3. Copy these interim fixes in the efixes directory of the was512_cf3_aix folder.
4. On the command prompt, set the JAVA_HOME variable to point to the <WBISF_root>/java folder.
5. From the same command prompt, change the updateInstaller directory, and run **`updateWizard.sh`**.
6. Accept the default language (English).
7. Follow the on-screen instructions, making sure that **IBM WebSphere Application Server V5.1.0.2** is selected and that the correct application server installation directory is also selected.
8. Select **Install Fixes** in the next window. We are not doing a FixPack install, but efixes instead.
9. Select the directory where efixes were copied above.
10. The wizard will scan for the efixes in the given folder. Select all listed fixes, checking that the wizard lists all the fixes that were downloaded.
11. Review the summary and click **Finish** to install the fixes.

IBM HTTP Server

There are two versions of IBM HTTP Server that can be used for this setup. In our test scenarios, we used IBM HTTP Server V2.0.

Important: IBM HTTP Server V2.0 should be installed on a separate machine as shown in Figure 4-2 on page 46. Please refer to this figure for the location of the WebSphere Business Integration Server Foundation V5.1 components.

IBM HTTP Server V1.3.x

IBM HTTP Server V1.3.x ships with the WebSphere Business Integration Server Foundation V5.1 CDs.

In order to install IBM HTTP Server V1.3, follow these instructions:

1. Change to the following directory on Disk 1: /cdrom/aix/WAS/ihs.
2. Run **`installIHS.sh`** and follow the on-screen instructions.

IBM HTTP Server V2.0

IBM HTTP Server V2.0 is packaged together with WebSphere Business Integration Server Foundation V5.1 and is available for download from the IBM external Web site.

The following guidelines can be used to install IBM HTTP Server V2.0:

1. Extract the downloaded IBM HTTP Server V2.0 archive into a temporary directory.
2. Locate and run **`installIHS.sh`** script to start the installation.
3. Follow on-screen instructions to finish installing IBM HTTP Server V2.0.

WebSphere Application Server Plug-in

WebSphere Application Server V5.1 Plug-in must be installed on the same machine as IBM HTTP Server. The IBM HTTP Server V2.0 installation described above does not include WebSphere Application Server V5.1 Plug-in. The plug-in must be installed separately from WebSphere Business Integration Server Foundation V5.1 installation disks.

The following guidelines illustrate how to install WebSphere Application Server V5.1 Plug-in:

1. Mount WebSphere Business Integration Server Foundation V5.1 Disk 1.
2. Change to the aix/WAS folder. This represents the WebSphere Application Server V5.1 base folder.
3. Run the install script to bring up the Installation Wizard.
4. Accept the license.
5. Wait for the prerequisite checks to finish. Verify that no AIX system packages or patches are missing. If there is a missing package warning, refer to 4.3.2, “Software requirements” on page 46 where the required AIX system packages are listed.
6. Choose a **Custom** installation.

7. Click **Web Server Plug-in** and select the **Plug-in for IBM HTTP Server V2.0** plug-in. Make sure the remaining options are unchecked.
8. Accept the default installation folder.
9. Follow on-screen instructions to complete the WebSphere Application Server V5.1 Plug-in installation.

Installing Cumulative Fix 3

Cumulative Fix 3 should also be applied to the machine hosting IBM HTTP Server and the WebSphere Application Server V5.1 plug-in. To do this, follow the instructions given in “Installing Cumulative Fix 3 and interim fixes” on page 51. Interim fixes do not need to be installed.

Note: The level of WebSphere Application Server that is selected will differ, since Cumulative Fix 2 will not have been installed on the WebSphere Application Server V5.1 Plug-in. The version will be V5.1.0.

4.3.4 Configuration

This section focuses on integrating various software components of WebSphere Business Integration Server Foundation V5.1. These components include:

- WebSphere Business Integration Server Foundation installation verification
- IBM HTTP Server
- WebSphere MQ
- IBM DB2 UDB
- Tivoli Directory Server (for Staff plug-in and Global Security)
- BPE container configuration

Configuration tasks for the distributed environment are not significantly different from those of the single-server environment above. There are, however, a few subtle differences which are outlined in the following sections.

Installation verification

Before proceeding with the configuration of the above components, WebSphere Business Integration Server Foundation V5.1 installation verification should be performed. This process will ensure that there are no problems starting the server. Sometimes there are port conflicts with existing AIX V5.2 ports. These are highlighted and workarounds are documented as follows.

Disabling AIX port 9090

AIX V5.2 ships with a Web based system manager utility called wsmserver. The **`wsmserver`** command is used to control the server processes used by the Web

based System Manager. The servers are used to enable applet and client-server modes of execution. This server utility runs on port 9090 by default.

In order to successfully run WebSphere Business Integration Server Foundation V5.1, wsmserver port 9090 must be disabled. The following instructions illustrate how to disable the port.

1. Log in as root.
2. Edit the /etc/services file and locate and comment out the following line (by putting a # in front of it):

   ```
   #wsmserver              9090/tcp
   ```

3. Edit the /etc/inetd.conf file. Locate and comment out the following line:

   ```
   #wsmserver stream tcp nowait root /usr/websm/bin/wsmserver wsmserver -start
   ```

4. Restart the inetd process. This will make the inetd process re-read inetd and services configuration files. Issue the following command to restart inetd.

   ```
   kill -HUP `ps -ef | grep inetd | awk '{print $2}`
   ```

Install verification

1. Start the install verification utility using the **ivt.sh** command.

   This starts up the command line install verification test utility. There is no interactive interface.

2. Note that all the statuses are marked `Passed` or `Succeeded`.
3. Start the Administrative Console and log in as admin to verify admin console functionality. The URL for the Administrative Console is `http://<hostname>:9090/admin`.

IBM HTTP Server

In order to configure IBM HTTP Server, do the following.

1. Verify the location of plug-in files. Change to the /usr/WebSphere/AppServer/bin directory and verify that following files exist:
   - mod_app_server_http.so
   - mod_app_server_http_eapi.so
   - mod_ibm_app_server_http.so
   - mod_was_ap20_http.so
2. Edit <IHS_root>/conf/httpd.conf.
3. Search for the `ServerName` directive and verify that it is set properly.
4. Go to the end of the file and verify that it contains the following lines:

Example 4-3 httpd configuration

```
LoadModule ibm_app_server_http_module "/usr/WebSphere/AppServer/bin/mod_was_ap20_http.so
WebSpherePluginConfig "/usr/WebSphere/AppServer/config/cells/plugin-cfg.xml"
```

BPE supporting components

Configuring the BPE container in a distributed environment is similar to configuring in a single-server environment. The key difference is that the products are installed on separate machines.

As in the single-server environment, setting up the BPE container consists of configuring:

- WebSphere MQ Queue Managers and appropriate queues
- The DB2 database for the BPE container
- The Staff plug-in that utilizes Tivoli Directory Server

The high-evel task plan for configuring the BPE container is given in the InfoCenter in the topic brought up by clicking **WebSphere Business Integration Server Foundation** → **Administering** → **Applications** → **Process Choreographer** → **Configuring the business process container**.

WebSphere MQ

Refer to the InfoCenter topic *Creating the queue manager and queues for the business process container* under the page listed above. In summary, the following steps must be completed:

1. Log in as root.
2. Switch user to mqm.
3. Run the **`/usr/mqm/bin/setmqcap 1`** command to set the number of processor to `1`.
4. Run the createQueues.sh script and provide it with a Queue Manager name. In our tests, we used: `QM1`
5. Start the queue manager:

   ```
   /usr/mqm/bin/strmqm QM1
   ```

6. Run the listener for the queue manager:

   ```
   /usr/mqm/bin/runmqlsr -t tcp -p 1414 -m QM1 &
   ```

Tip: During tests, we used WBISFQM1 and ran into the following MQJMS exception:

```
MQJMS2005: failed to create MQQueueManager for '<queue manager>'
```

Creating a new Queue manager called QM1 later on and re-configuring WebSphere MQ queue connection factories to communicate with WebSphere MQ solved the problem.

DB2 Enterprise Server Edition setup

The database must be configured prior to using the Business Process container. Review the material in the InfoCenter topic *Creating the database for the business process container* for configuration steps.

This configuration differs from the procedure followed in the single-server environment since the database is located on a remote server. The remote database server must be cataloged locally on the WebSphere Business Integration Server Foundation V5.1 box, then the BPE database should be created.

The following high-level tasks should be followed to finish DB2 configurations.

Perform the following steps on DB2 Server machine:

1. Ensure that DB2 is installed properly on the DB2 server machine as per the guidelines in DB2 InfoCenter and WebSphere Business Integration Server Foundation V5.1 InfoCenter.
2. In order to avoid deadlocks, issue the following command as the db2admin user on the server machine:

   ```
   db2set DB2_RR_TO_RS=YES
   ```

The WebSphere Business Integration Server Foundation machine does not have the full database installed, only the client. All data sources defined in WebSphere Business Integration Server Foundation access databases through this client.

Perform the following configuration as the root user on the WebSphere Business Integration Server Foundation V5.1 server machine.

1. Add the following lines in the .profile file of the WebSphere Business Integration Server Foundation instance owner.

   ```
   ./home/db2admin/sqllib/db2profile
   ```

Perform the following steps on the WebSphere Business Integration Server Foundation V5.1 server machine as the db2admin user.

1. Log in as db2admin.
2. Catalog the remote DB2 node as follows:

   ```
   db2 catalog tcpip node <remote node> remote <FQHN> server 50000
   ```

3. Attach to the remote node and verify that the catalog operation was successful.

   ```
   db2 attach to <remote node> user db2admin using db2admin
   ```

4. Create a new database called BPEDB for Business Process Container, and the schema for the database.

   ```
   db2 create database BPEDB using codeset UTF-8 territory en-us
   ```

5. Change to the /usr/WebSphere/AppServer/ProcessChoreographer/ directory and edit the createTablespaceDb2.ddl script.

   Note: For distributed environments which will be used for production or system tests, BPE tables should be separated into different tablespaces. BPE tables are grouped as indicated in the createSchemaDB2.sh script. Create the tablespaces first using createTablespaceDb2.ddl and then populate the schema using createSchemaDB2.sh.

   Change the `@location@` tag as per your Database environment. For our test, we used the following script.

Example 4-4 createTablespaceDb2.ddl

```
CREATE TABLESPACE STAFFQRY MANAGED BY SYSTEM USING( 'STAFFQRY' );
CREATE TABLESPACE INSTANCE MANAGED BY SYSTEM USING( 'INSTANCE' );
CREATE TABLESPACE AUDITLOG MANAGED BY SYSTEM USING( 'AUDITLOG' );
CREATE TABLESPACE WORKITEM MANAGED BY SYSTEM USING( 'WORKITEM' );
CREATE TABLESPACE COMP MANAGED BY SYSTEM USING( 'COMP' );
CREATE TABLESPACE TEMPLATE MANAGED BY SYSTEM USING( 'TEMPLATE' );
-- start import scheduler DDL: createTablespaceDB2.ddl
CREATE TABLESPACE SCHEDTS MANAGED BY SYSTEM USING( 'SCHEDTS' );
-- end import scheduler DDL: createTablespaceDB2.ddl
```

6. Connect to the BPEDB database created above.

   ```
   db2 connect to BPEDB user db2admin using db2admin
   ```

7. Run the createTablespaceDb2.ddl script :

   ```
   db2 -tf createTablespaceDb2.ddl
   ```

8. Populate the BPEDB schema by running the createSchemaDb2.ddl script:

   ```
   db2 -tf createSchemaDb2.ddl
   ```

Perform the following step on the DB2 Server machine:

- Bind the required CLI packages to the database:

```
db2 connect to BPEDB user db2admin using db2admin
db2 bind /home/db2admin/sqllib/bnd/@db2cli.lst blocking all grant public
```

Tivoli Directory Server setup for Staff plug-in

Refer to “Tivoli Directory Server setup for Staff plug-in” on page 42 for instructions.

BPE container

The next task is to configure the BPE container in WebSphere Business Integration Server Foundation V5.1.

Installing Business Process Container

The BPE container will be set up to communicate with the WebSphere MQ and DB2 Enterprise Server Edition components. Tivoli Directory Server is optionally configured for Staff plug-in support in the BPE container.

Review the instructions in the InfoCenter topic found by clicking **WebSphere Business Integration Server Foundation** → **Administering** → **Applications** → **Process Choreographer** → **Configuring the business process container** → **Using the Install Wizard to configure the business process container.**

A high-level overview of the guidelines is given below.

1. Log in as root and start WebSphere Business Integration Server Foundation V5.1 application server.
2. Connect to the Administrative Console.
3. Define environment variables under **Environment** → **Manage WebSphere Variables** at the node level, as shown in Figure 4-3 on page 60.

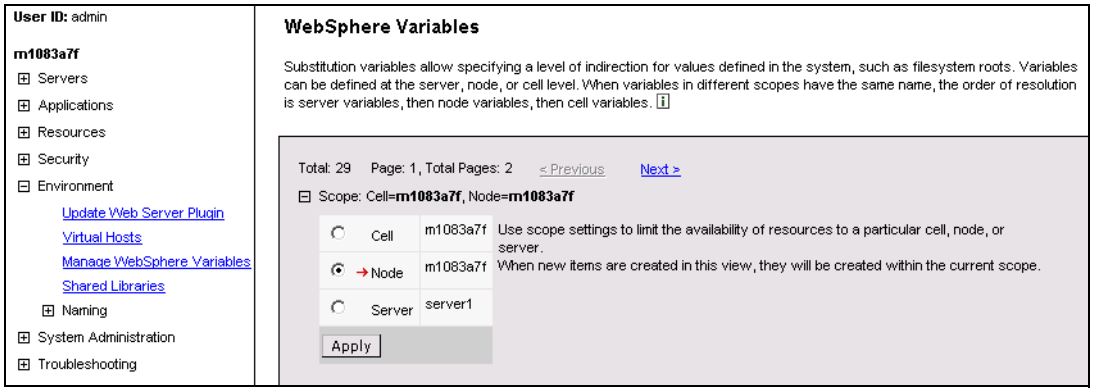


Figure 4-3 Environment Variable Definition at Node level

4. Define the following variables. Figure 4-4 on page 61 shows the output.

Table 4-7 Environment Variables

WebSphere Environment Variable	Variable Value
MQ_INSTALL_ROOT	/usr/mqm
MQJMS_LIB_ROOT	${MQ_INSTALL_ROOT}/java/lib
DB2_JDBC_DRIVER_PATH	/home/db2admin/sqllib/java
DB2UNIVERSAL_JDBC_DRIVER_PATH	/home/db2admin/sqllib/java

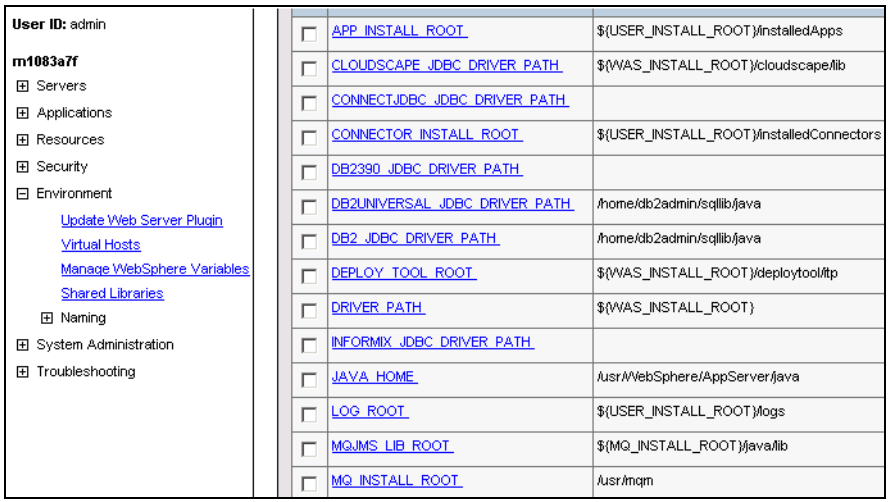


Figure 4-4 WebSphere Business Integration Server Foundation V5.1 Environment Variables

5. Run Process Choreographer Install Wizard by navigating to **Serve** → **Applications Servers** → ***<server_name>*** → **Business Process Container** → **Business Process Container Install Wizard**.
6. Select **DB2 UDB 8.1 Universal JDBC Driver Provider XA**. All the appropriate variable settings are selected automatically. Set the datasource user name to `db2admin` and the password to `db2admin` (or as appropriate to your environment). Click **Next**. Figure 4-5 on page 62 indicates this setting.

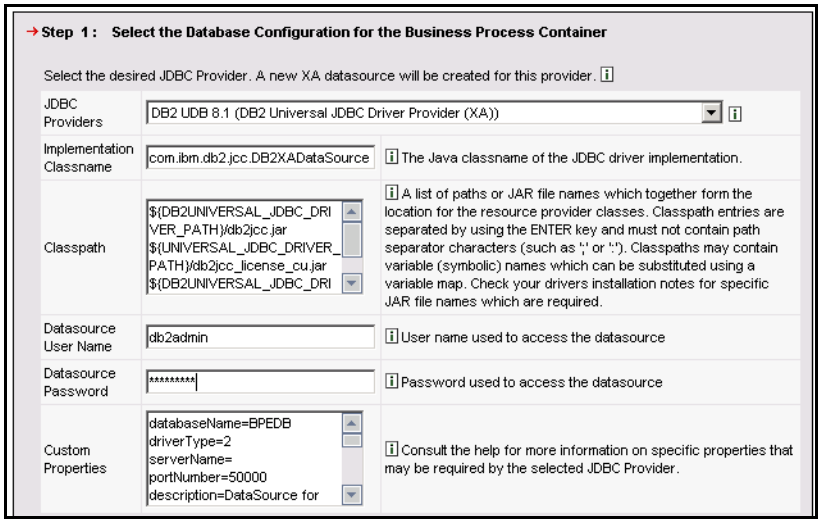


Figure 4-5 Business Process Container Install Wizard : Step 1

7. In the Step 2 window, enter the JMS Provider settings as shown in Table 4-8.

Table 4-8 JMS settings

JMS Settings	Values
Queue Manager	QM1
Security Role Mapping	mqm
JMS API User ID	mqm
JMS API Password	mqmadmin [this must be set prior to configuring MQ]

Note: The JMS API User ID is the operating system user ID under which WebSphere MQ Client/Server is installed. It can be any operating system user ID that has administrative authority to perform operations against WebSphere MQ Server/Client. The user ID should be part of the WebSphere MQ group.

Figure 4-6 on page 63 shows the settings used in Step 2.

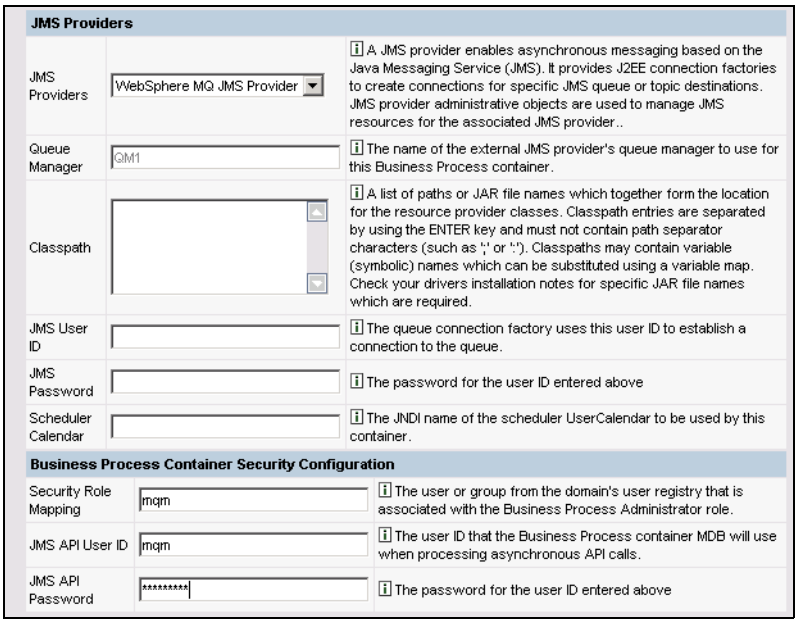


Figure 4-6 Business Process Container Install Wizard : Step 2

8. In Step 3, select the **Create new JMS resources using default values**. Also set **Check this box to install Web client** checkbox. Figure 4-7 on page 64 shows these settings.

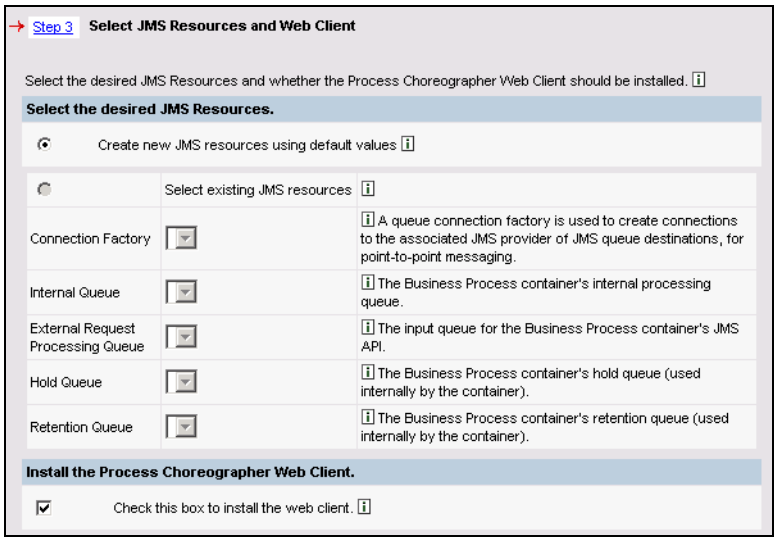


Figure 4-7 Business Process Container Install Wizard: Step 3

9. Review the summary information and click **Finish** to install the Business Process container. The summary is shown in Figure 4-8.

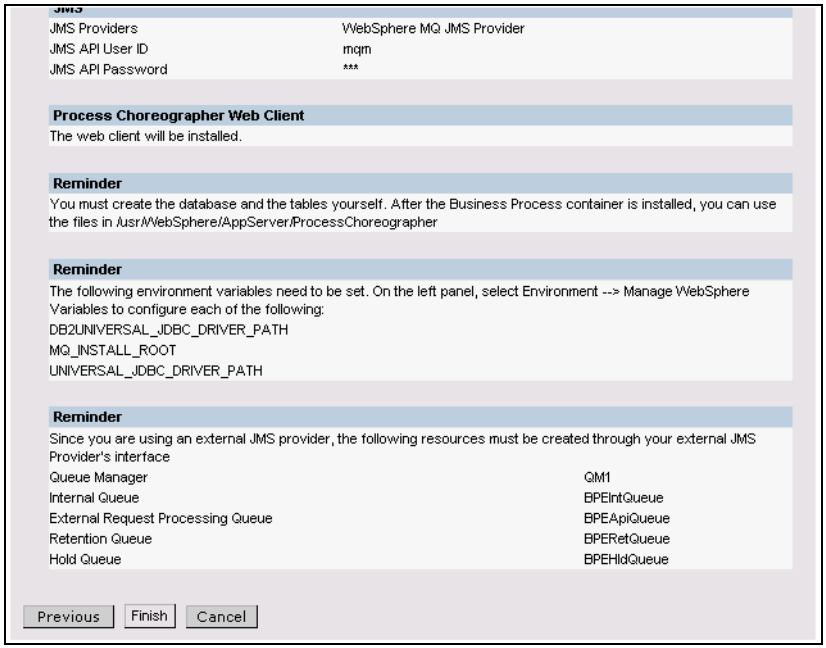


Figure 4-8 Business Process Container Install Wizard: Step 4, Summary

10. Upon successful completion, you will be prompted to `Save to Master Configuration`. Save the configuration for WebSphere.

Configuring WebSphere MQ JMS Provider

The next task is to configure the WebSphere MQ JMS Provider.

1. Navigate to **Resources** → **WebSphere MQ JMS Provider**.
2. Set the scope to `Server`.
3. Click **WebSphere MQ Queue Connection Factories**.
4. Select a Queue Connection factory and use the values in Table 4-9 to set up the properties. Accept the default values for unlisted properties. Repeat this task for both BPECF and BPECFC connection factories.

Table 4-9 WebSphere MQ JMS Provider Settings

Property	Value
Queue Manager	QM1
Host	<mq server hostname>
Port	1414
Transport Type	CLIENT

Figure 4-9 on page 66 indicates the settings used in the WebSphere MQ JMS Provider settings.

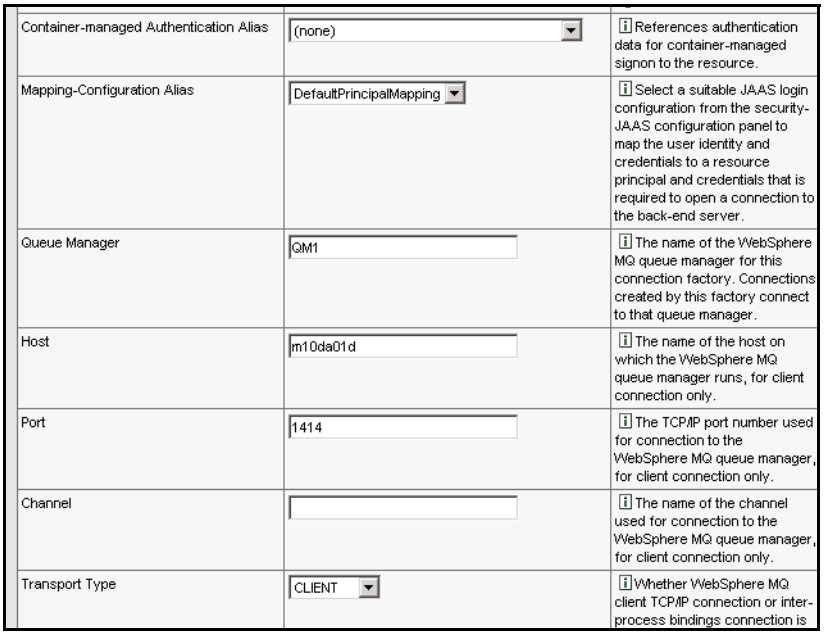


Figure 4-9 WebSphere MQ JMS Provider settings

5. Save the master configuration.
6. In order for the changes to take effect, stop and start the server.
7. Review the contents of SystemOut.log to see if any errors or exceptions are thrown by the BPE container.

Configuring Staff service for Process Choreographer

Process choreographer uses Staff plug-ins to determine who can start a process or claim an activity. Your business processes can also use the Staff plug-in services to resolve staff queries. Each type of directory service requires a different Staff plug-in. You can register multiple Staff plug-ins. The user registry and system plug-ins are already installed and can be used without any configuration. To configure a Staff Plugin Provider, consult the following InfoCenter topic. Click **WebSphere Business Integration Server Foundation** → **Administering** → **Applications** → **Process choreographer** → **Configuring the staff service for process choreographer**.

A J2C Authentication alias must be defined first in the WebSphere Business Integration Server Foundation V5.1 Security section. Navigate to **Security** → **JAAS Configuration** → **J2C Authentication Data**. Click **New** to define a new authentication alias as follows. If this alias is not set, an anonymous logon to the LDAP server is used.

Table 4-10 J2C Authentication Alias

Property	Value
Alias	<hostname>/tds
User ID	cn=root
Password	tdsadmin
Description	TDS Authentication Alias

Now configure the Staff plug-in using the following high-level guidelines.

1. Configure the Staff plug-in at the node level. Navigate to **Resources** → **Staff Plugin Provider**. Make sure the scope is set to `Node Level`.
2. Select **LDAP Staff Plugin Provider**.
3. Select **Staff Plugin Configuration**. Define a new Staff plug-in configuration. Import the /usr/WebSphere/AppServer/ProcessChoreographer/Staff/LDAPTransformation.xsl file and click **Next**. This is shown in the figure below.

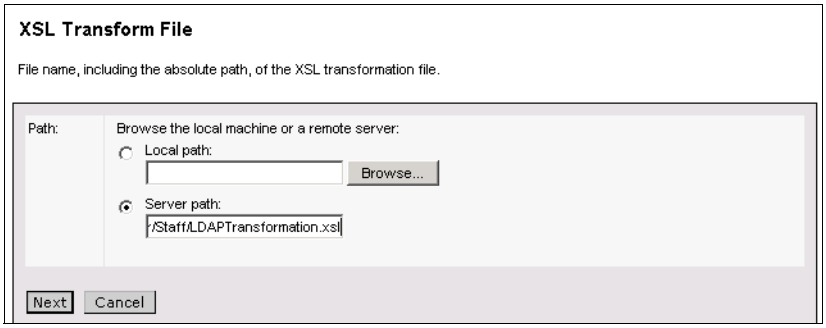


Figure 4-10 Staff plug-in path

4. The next section defines the general properties for the new Staff Plugin Provider. Use the settings as indicated in the table below. Figure 4-11 on page 68 shows the settings in effect.

   Make sure to click **Apply** because in the next step, we define the Custom Properties for the new Staff plug-in. Clicking the **OK** button will take you back to the Administrative Console main screen.

Table 4-11 New Staff Plugin Provider properties

Property	Value
Name	tds
Description	<optional>
JNDI Name	bpe/staff/ldapserver1

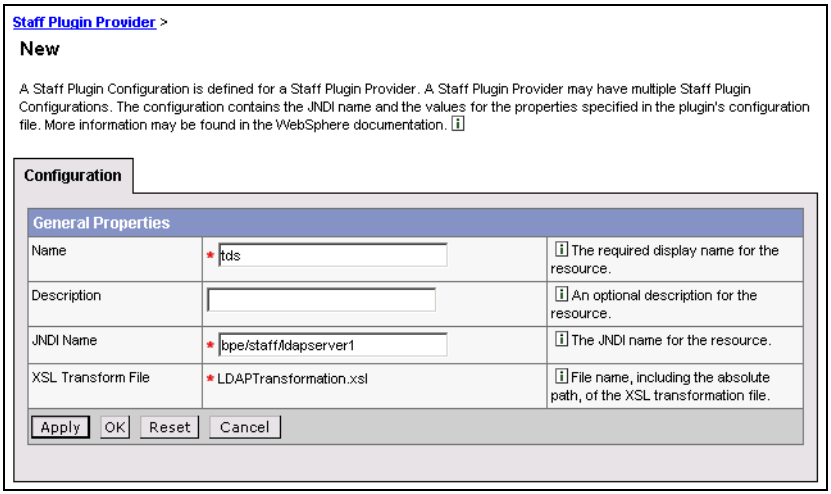


Figure 4-11 New Staff Plugin Provider properties

5. Configure the custom properties for the Staff Plugin Provider, created in the previous step, as given in the table below.

Table 4-12 Staff plug-in configuration parameters

Property	Value
AuthenticationAlias	<hostname>/tds
AuthenticationType	<optional>
BaseDN	cu=itso,o=ibm,c=us
ContextFactory	com.sun.jndi.ldap.LdapCtxFactory
ProviderURL	ldap://<hostname>:389
SearchScope	subtreeScope

The figure below shows these settings in effect.

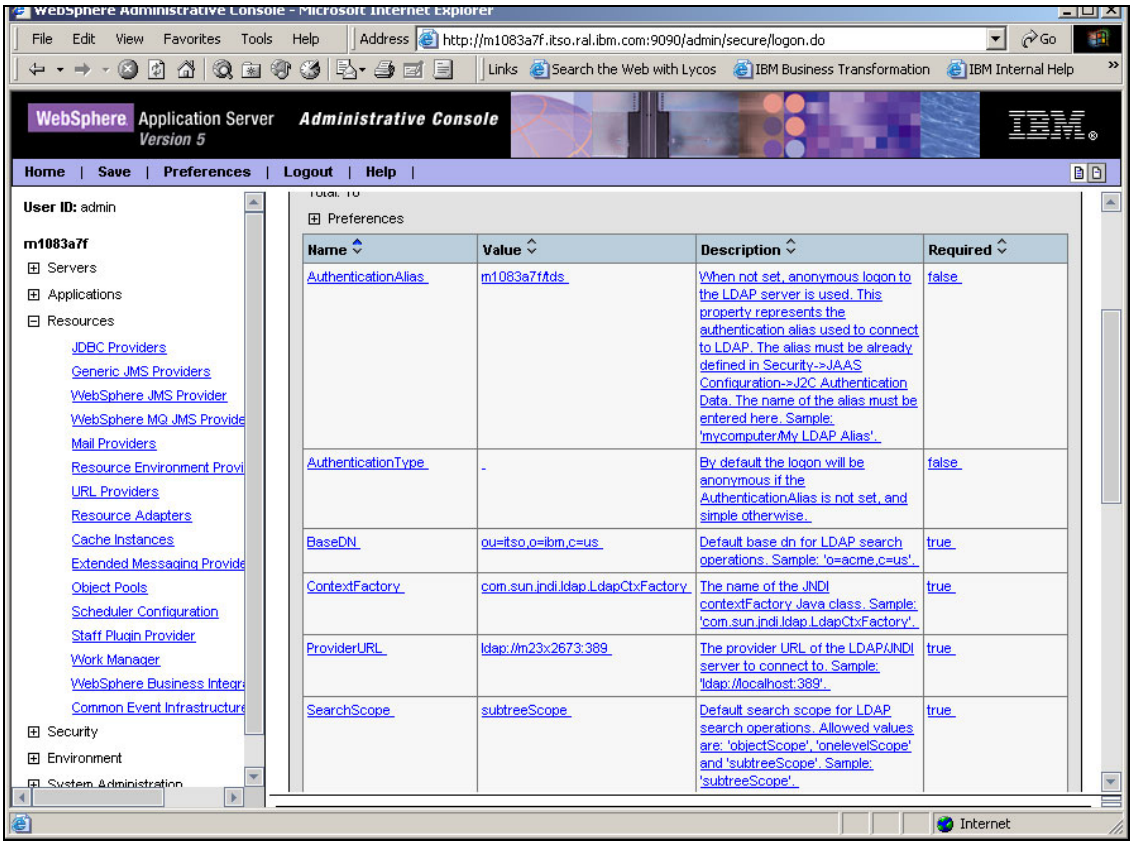


Figure 4-12 Staff Plugin Provider settings

6. Make sure to save the master configuration after making the above changes.
7. To activate the plug-in, stop and start the server.

4.4 Configuring for scalability

This section builds on the distributed configuration to add scalability.

4.4.1 Planning

A scalable WebSphere Business Integration Server Foundation installation requires multiple application server instances to be configured and work together to process the workload. There are two ways in which applications can be scaled:

- Vertical scalability

  An application is said to scale vertically if it can take full advantage of a larger server. To make use of this additional power, it is often neccesary to configure additional instances of an application. The eventual scale is limited by the maximum configuration of the machine and its ability to run additional copies of the software.

- Horizontal scalability

  An application is said to scale horizontally if more servers can be added to the group hosting the application. There may be multiple application server instances running on each of the nodes in the cluster to make the best use of the size of the machine. The eventual scale is limited only by the ability of the infrastructure to manage additional machines.

In either of the scenarios listed above, the capabilities of WebSphere Application Server Network Deployment are required to provide clustering and workload balancing. The steps required to configure a cluster are given later in this chapter.

Refer to *IBM WebSphere V5.1 Performance, Scalability, and High Availability WebSphere Handbook Series*, SG24-6198 for further details of scalable configurations of WebSphere Application Server.

Horizontal scalability

When horizontal scalability is employed, multiple copies of an application run on multiple servers. Using a WebSphere Application Server cluster allows the servers to be managed as a single entity. This allows a higher throughput to be handled by adding more physical capacity to the insfrastructure.

Figure 4-13 on page 71 shows an example of a horizontally scaled environment.

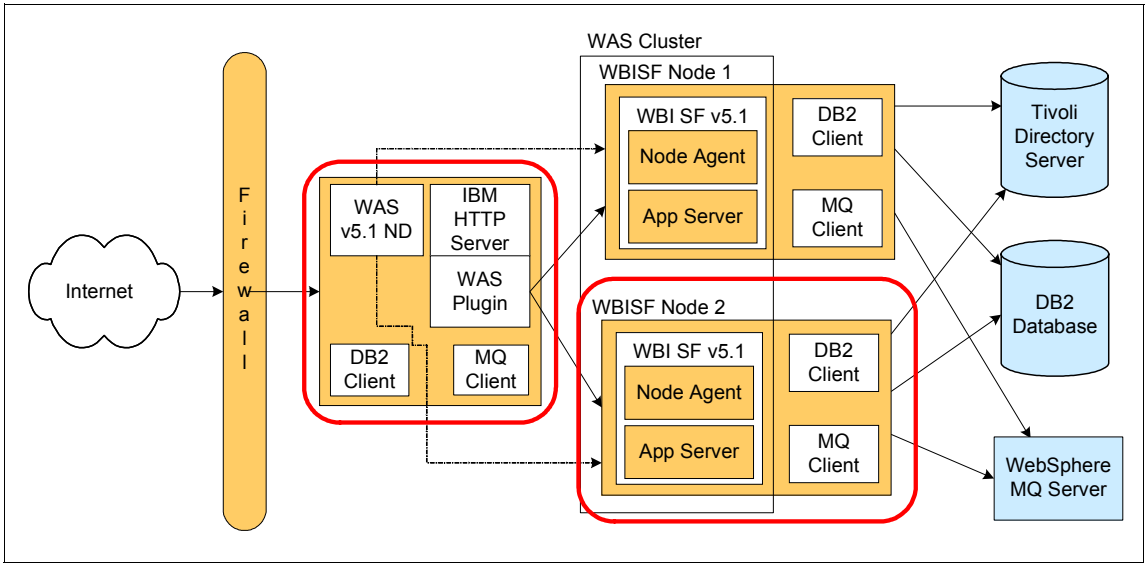


Figure 4-13 Horizontally scalable environment

The diagram shows:

- Two WebSphere Business Integration Server Foundation nodes, each of which has an application server and a node agent. These nodes are configured as a cluster. Additionally, the DB2 client and WebSphere MQ client are installed on these nodes.
- One node which hosts the IBM HTTP Server with the WebSphere plug-in and WebSphere Application Server Network Deployment. This is responsible for managing the servers in the cluster.
- Three additional servers which host Tivoli Directory Server, DB2 Enterprise Server Edition and WebSphere MQ required to support the BPE container.

The configuration is horizontally scaled since the Application Servers could be configured to host the same applications.

The additional nodes in this configuration are highlighted in the diagram. The remainder of this chapter will outline the steps required to install the additional servers.

This configuration adds a second WebSphere Business Integration Server Foundation node and a WebSphere Application Server Network Deployment server. The second WebSphere Business Integration Server Foundation node will be installed on a separate computer and the WebSphere Application Server

Network Deployment server will be installed on the same node as the IBM HTTP Server.

Vertical scalability

Running multiple copies of an application on a single computer allows the full capacity of a large machine to be realized. The same principle of creating copies of applications is used as in the horizontal scaling scenario. The difference in configuration is that new application servers are created on nodes which already host application servers.

A vertically scaled environment is shown in Figure 4-14.

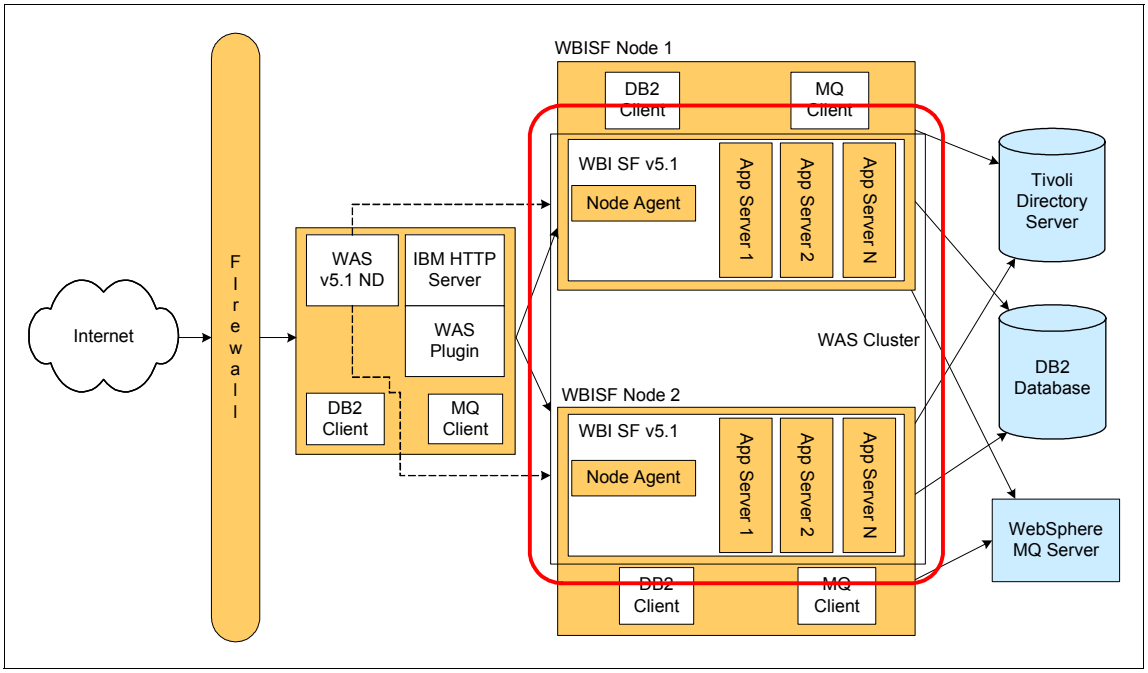


Figure 4-14 Vertically scalable environment

The diagram shows multiple copies of an application server on each node in the cluster. Each of these application servers could host an instance of the BPE container.

Planning for a scalable BPE container

As we have seen in previous sections, using scalable WebSphere Business Integration Server Foundation topologies can allow the BPE container to support higher workloads.

However, it is important to remember that the BPE container relies on the JMS provider, database and LDAP server. This means that scaling the core WebSphere Business Integration Server Foundation installation on its own is not sufficient. Each of these components could also become a bottleneck, limiting the ability of the application to handle the required throughput. For this reason, the sizing of the other components must also be taken into account. The eventual scale of the application may also mean that the configuration of the other components needs to be altered, for instance by introducing WebSphere MQ clustering. This is shown in Figure 4-15.

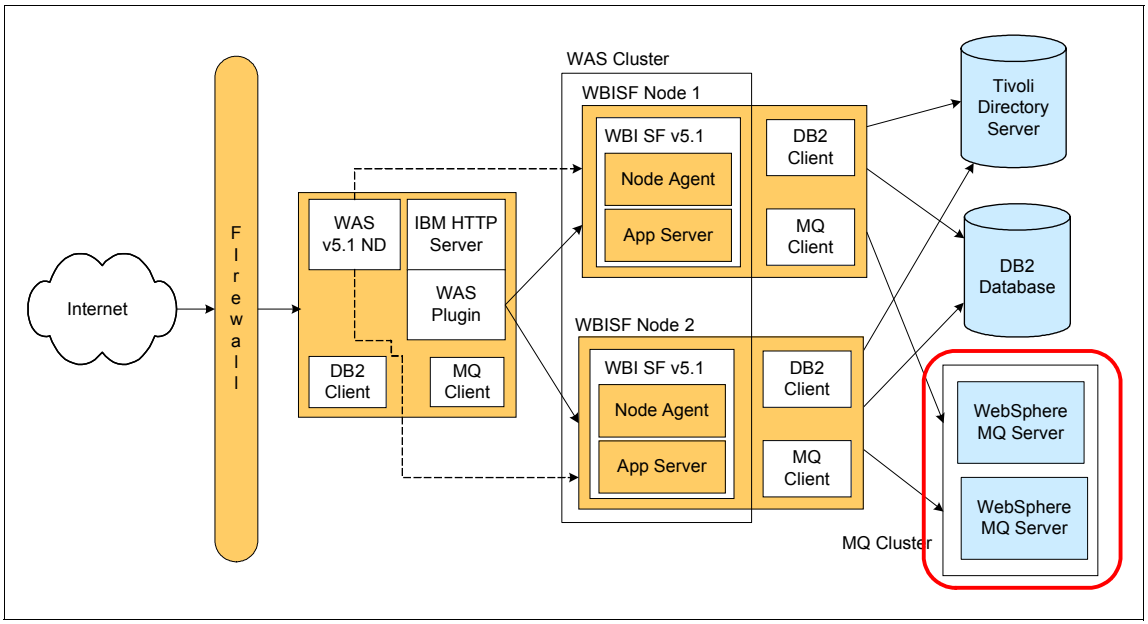


Figure 4-15 Topology using an MQ cluster

The topology is similar to the WebSphere Application Server environments set out in the previous sections. In this instance, however, the WebSphere MQ JMS provider is a clustered toplogy.

Refer to the following links for further information about WebSphere MQ clustering in WebSphere Application Server environments.

- InfoCenter topic found by clicking **WebSphere Business Integration Server Foundation** → **Administering** → **Task overviews** → **Using process choreographer** → **Process choreographer scenarios for clustering**.
- The IBM Redbook *WebSphere Application Server and WebSphere MQ Family Integration*, SG24-6878

- Queue Manager Clusters, WebSphere MQ documentation:

  http://publibfp.boulder.ibm.com/epubs/html/csqzah06/csqzah06tfrm.htm

- *WebSphere Application Server V5: Using WebSphere and WebSphere MQ clustering*, Redbook TechNote:

  http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/tips0224.html?Open

4.4.2 Software requirements

Refer to 4.3.2, “Software requirements” on page 46 for details on this topic.

4.4.3 Installation

This section details the installation instructions for the second WebSphere Business Integration Server Foundation V5.1 on the AIX platform.

WebSphere MQ Client

Refer to “WebSphere MQ Client” on page 48 for installation instructions.

IBM DB2 Client

Refer to “IBM DB2 UDB Client” on page 49 for installation instructions.

WebSphere Business Integration Server Foundation

Refer to “WebSphere Business Integration Server Foundation” on page 50 for installation instructions.

WebSphere Application Server V5.1 Network Deployment

This topic describes how to install WebSphere Application Server V5.1 Network Deployment as the root user on a AIX platform.

Note that Network Deployment component will be installed on the same machine as the Web server, as described in the topology and planning section above.

Refer to the InfoCenter for details on installing WebSphere Application Server V5.1 Network Deployment:

http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.websphere.nd.doc/info/ae/ae/tins_install.html

The following which contains the WebSphere Application Server V5. Network Deployment image.

1. Change to the mounted CDROM directory.

2. Issue the `launchpad.sh` command to bring up the installation wizard.
3. Click **Install the product** to start WebSphere Application Server V5.1 Network Deployment installation.
4. Accept the license agreement.
5. The tool starts the prerequisite checker. This tool is built into the installer program. Verify that all prerequisites are met.

   Note: If you are prompted to reconfigure the WebSphere product to coexists with othe version on the Network Deployment, skip this step. Keep in mind that Network Deployment Machine has an installation of the WebSphere Application Server 5.1 Plugin component. The prerequisite checker utility detects that and assumes that it is a full version of a WebSphere V5.1 installation.

6. Select all components except Embedded messaging client and click **Next**.
7. Install under /usr/WebSphere/DeploymentManager and click **Next**.
8. Accept the default node name, hostname and cell name. Click **Next**.
9. Review the summary and start the installation.

Note: The InfoCenter lists four installation methods at **WebSphere Business Integration Server Foundation** → **Installing** → **Getting Started** → **Installing the product** → **Installing the Integration Server on AIX platforms**. We recommend Method 1, because Method 4 proved problematic during tests.

If the Install Verification utility starts automatically after the WebSphere Application Server V5.1 Network Deployment installation finishes, close the window and exit from the Product Installation window as well. We will be doing the IVT and configuration in 4.4.4, “Configuration” on page 77.

Installing Integration Server Administrative Extensions

Before WebSphere Application Server Network Deployment can be used to administer WebSphere Business Integration Server Foundation V5.1 servers, the WebSphere Business Integration Server Foundation administrative extensions must be installed. Refer to the InfoCenter below for further information. Open the InfoCenter and navigate to **WebSphere Business Integration Server Foundation** → **Installing** → **Installing the product**. Read the section entitled *Why and when to perform this task* in this topic.

The following guidelines provide an overview of the process of installing the administrative extentions on the WebSphere Application Server Network Deployment node.

1. Mount CD1 which contains the WebSphere Business Integration Server Foundation V5.1 image.
2. Change to the mounted CD-ROM directory aix.
3. Issue the install utility to bring up the installation wizard.
4. Accept the license agreement.
5. The tool starts the prerequisite checker. Verify that all prerequisites are met.
6. The following window is displayed. Choose **Add to the existing copy of WebSphere Application Server Network Deployment V5.1**. The directory should be set to `/usr/WebSphere/DeploymentManager`. Click **Next**.

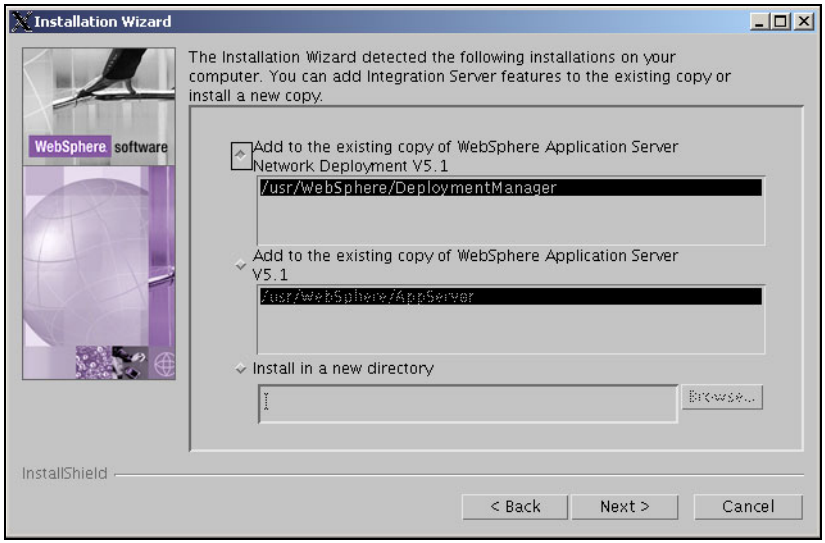


Figure 4-16 Installing Integration Server Administrative Extensions

7. In the next window, when prompted to enter WebSphere Business Integration Server Foundation Disk 2, insert and mount Disk 2. Click **Next**.
8. Insert and mount WebSphere Business Integration Server Foundation Disk 1 when prompted.
9. Review the summary. Click **Next**.
10. Install under /usr/WebSphere/DeploymentManager and click **Next**
11. Accpet the default node name, hostname and cell name. Click **Next**.
12. Review the summary and start the installation.

Installing Cumulative Fix 3 and interim fixes

At the time of writing, Cumulative Fix 3 was the latest available for WebSphere Application Server Network Deployment V5.1. It is recommended that this fix and any other interim fixes be applied since they address several key issues.

Fixes are available from the WebSphere Application Server support site:

`http://www.ibm.com/software/webservers/appserv/was/support/`

1. Open the WebSphere Application Server support site and navigate to WebSphere Application Server 5.1 Cumulative Fix 3 (5.1.0.3).
2. Download the AIX - ND package. The file is called was510_nd_cf3_aix.zip.
3. At the command prompt, set the `JAVA_HOME` variable to point to the `<WBISF_root>/java` folder.
4. Extract the fix file into the was510_nd_cf3_aix folder.

> **Note:** Extracting this file also creates files necessary to run the *updateInstaller* wizard.

5. Run **`updateWizard.sh`** from the was512_nd_cf3_aix folder.
6. Accept the default language (English).
7. Follow the on-screen instructions, making sure that **IBM WebSphere Network Deployment V5.1.0** is selected and that the correct application server installation directory is also selected.
8. Select **install FixPack** in the next window. Accept the default location for the FixPack folder.
9. Follow the remaining on-screen instructions to finish installing Cumulative Fix 3.

There are currently no interim fixes specifically aimed at WebSphere Application Server Network Deployment.

4.4.4 Configuration

This section describes how to set up WebSphere Business Integration Server Foundation V5.1 on the second node. The procedure differs from the simple distributed environment configuration, because the clustering enabled by the Network Deployment server requires some additional activities to be carried out. The differences in scalable configuration are outlined in the following sections.

This section focuses on how to integrate various software components of WebSphere Business Integration Server Foundation V5.1. These components include:

- WebSphere Application Server Network Deployment V5.1
- WebSphere Business Integration Server Foundation Installation Verification
- IBM HTTP Server
- WebSphere MQ
- IBM DB2 UDB
- Tivoli Directory Server (for Staff plug-in and Global Security)
- BPE container configuration

Installation verification

Please refer to “Installation verification” on page 54 for instructions.

BPE supporting components

The WebSphere Business Integration Server Foundation specific elements from “BPE supporting components” on page 56 need to be configured. The instructions are outlined below.

IBM DB2 UDB Client setup

Perform the following steps on the WebSphere Business Integration Server Foundation V5.1 server machine.

1. WebSphere Application Server machine only has a V8.1 client installed, and all DB2 data sources defined in WebSphere Application Server access DB2 databases through this client, the source being the db2profile file in the login profile of your V5.x instance owner.
2. Log in as root, and add the following line to the .profile file.

   ```
   . /home/db2admin/sqllib/db2profile
   ```

Perform the following steps on the WebSphere Business Integration Server Foundation V5.1 server machine as the db2admin user.

1. Log in as db2admin.
2. Catalog remote DB2 node as follows:

   ```
   db2 catalog tcpip node <remote node> remote FQHN server 50000
   ```

3. Attach to the remote node and verify that catalog operation was successful.

   ```
   db2 attach to <remote node> user db2admin using db2admin
   ```

4. Catalog the remote database in WebSphere Business Integration Server Foundation V5.1 node 2 as follows.

   ```
   db2 catalog database <database name> as <alias> at node <node name>
   ```

5. Test the connection to the cataloged database using the following command:

```
db2 connect to <database name> user db2admin using db2admin
```

Tivoli Directory Server setup for Staff plug-in

Refer to “Tivoli Directory Server setup for Staff plug-in” on page 42 for instructions.

Creating a WebSphere cluster

This additional configuration step results in the installed WebSphere Business Integration Server Foundation nodes becoming members of a cluster.

The first step in configuring Business Process Container in a clustered environment is to set up the WebSphere Application Server cluster and cluster members. A cluster member is an instance of WebSphere Application Server or WebSphere Business Integration Server Foundation.

Refer to “Creating a WebSphere cluster” on page 583 for further details.

Note: If you are creating a vertical scalability environment, then all servers in the cluster will be on the same node.

Scalable BPE container configuration

An overview of the process of configuring the BPE container in a WebSphere Application Server cluster is given in the InfoCenter topic found by clicking **WebSphere Business Integration Server Foundation.** → **Administering** → **Applications** → **Process Choreographer** → **Configuring the business process container** → **Configuring the business process container on a cluster**.

It is assumed that the steps mentioned in “Creating a WebSphere cluster” on page 79 have been followed to federate the WebSphere Business Integration Server Foundation V5.1 nodes into a cell and to create the cluster and cluster members by using the administrative console.

The tasks for configuring the Business Process Container are as follows:

1. Install Business Process Container using the wizard.
2. Configure WebSphere MQ JMS settings for Queue Connection Factories.
3. Configure Staff plug-in for Business Process Container.

Each task is discussed in detail below.

Configuring the BPE container in the cluster

In order to configure the Business Process Container in a clustered configuraton, follow the instructions given in the InfoCenter topic *Configuring the business process container on a cluster*.

There is essentially no difference between configuring Business Process Container in a distributed and scalable configuration. The high-level tasks are given below.

The first step is to configure the environment variables.

1. Log on to the Administrative Console running on the Deployment Manager machine as `admin`.
2. Define environment variables listed in Step 4 in “BPE container” on page 59. Navigate to **Environment** → **Manage WebSphere Variables** to define the listed variables. It is important to note that these variables must be defined at the Deployment Manager node level, and for all nodes that are part of the cluster.

   Figure 4-17 indicates the list of variables that must be updated at each node.

Name	Value	Scope
APP_INSTALL_ROOT	${USER_INSTALL_ROOT}/installedApps	cells:m10df56fNetwork:nodes:m10df4
CLOUDSCAPE_JDBC_DRIVER_PATH	${WAS_INSTALL_ROOT}/cloudscape/lib	cells:m10df56fNetwork:nodes:m10df4
CONNECTJDBC_JDBC_DRIVER_PATH		cells:m10df56fNetwork:nodes:m10df4
CONNECTOR_INSTALL_ROOT	${USER_INSTALL_ROOT}/installedConnectors	cells:m10df56fNetwork:nodes:m10df4
DB2390_JDBC_DRIVER_PATH		cells:m10df56fNetwork:nodes:m10df4
DB2UNIVERSAL_JDBC_DRIVER_PATH	/home/db2admin/sqllib/java	cells:m10df56fNetwork:nodes:m10df4
DB2_JDBC_DRIVER_PATH	/home/db2admin/sqllib/java	cells:m10df56fNetwork:nodes:m10df4
DEPLOY_TOOL_ROOT	${WAS_INSTALL_ROOT}/deploytool/itp	cells:m10df56fNetwork:nodes:m10df4
DRIVER_PATH	${WAS_INSTALL_ROOT}	cells:m10df56fNetwork:nodes:m10df4
INFORMIX_JDBC_DRIVER_PATH		cells:m10df56fNetwork:nodes:m10df4
JAVA_HOME	/usr/WebSphere/AppServer/java	cells:m10df56fNetwork:nodes:m10df4
LOG_ROOT	${USER_INSTALL_ROOT}/logs	cells:m10df56fNetwork:nodes:m10df4
MQJMS_LIB_ROOT	${MQ_INSTALL_ROOT}/java/lib	cells:m10df56fNetwork:nodes:m10df4
MQ_INSTALL_ROOT	/usr/mqm	cells:m10df56fNetwork:nodes:m10df4

Figure 4-17 Environment Variable Changes

3. Once variables are defined at all nodes, save the changes and make sure to set synchronization with the nodes.

The next step is to configure the BPE container.

Install BPE container as per the instructions listed above in the Distributed environment section. Refer to “BPE container” on page 59. Repeat Steps 5 to 10 from this section. Log in to the Administrative Console running on the Deployment Manager machine.

Important: Use the Business Process Container Install Wizard on the Deployment Manager machine to install and configure the business process containers in the cluster. This action causes the Business Process Container to be installed and configured on all the other application servers in the cluster.

To achieve this, select any server that is part of a cluster and use the Business Process Container Install Wizard for the installation. Once the installation is done and you are asked to save the master configuration, make sure that changes are sychronized with the nodes. Synchronizing changes will federate the business process container application among all cluster members.

Once installed, Business Process Container can be activated by starting the cluster. Look for any exceptions in the SystemOut.log on individual application servers running on each node.

Configuring WebSphere MQ Resources

The WebSphere MQ JMS configuration to support the BPE container requires that Queue Connection Factories to be set up on individual application servers.

Refer to “Configuring WebSphere MQ JMS Provider” on page 65 for instructions on how to set up WebSphere MQ JMS parameters at the server level. Use the values provided in Table 4-9 on page 65 for WebSphere MQ JMS settings.

These guidelines are also available in the WebSphere Business Integration Server Foundation InfoCenter.

Important: In the basic configuration scenario outlined in 4.2, “Basic configuration” on page 36, a BINDING transport type could be used to establish a IPC-based connection to the WebSphere MQ Server process. In a distributed and scalable environment, the WebSphere MQ Server process runs on a separate machine, hence a CLIENT-based TCP/IP connection must be established. As a result, when configuring the Queue Connection Factories, BPECF and BPECFC under WebSphere MQ JMS Provider, both the TCP/IP host and port must be specified. In addition, the Transport Type should be set to `CLIENT`.

Configuring Staff Plug-in

In order to configure the staff plug-in, follow the steps in “Configuring Staff service for Process Choreographer” on page 66.

There are, however, two differences when configuring the Staff plug-in in a scalable environment. Review the following important points before proceeding with configuring a Staff plug-in in a scalable environment.

1. J2C Authentication aliases are used by the Staff Plugin Provider. In a scalable environment, J2C authentication aliases are defined at the individual node level, similiar to the Staff Plugin Provider. The Staff Plugin Provider uses these node-level authentication aliases to authenticate against the Tivoli Directory Server server.
   - Make sure there are no authentication aliases defined from the previous distributed configuration. The key is to configure the authentication alias that is defined at the node level. The format of this alias looks like <dmgrhostManager>/<node>/tds.
   - Two J2C aliases must be defined for each node. This is shown in the figure below.

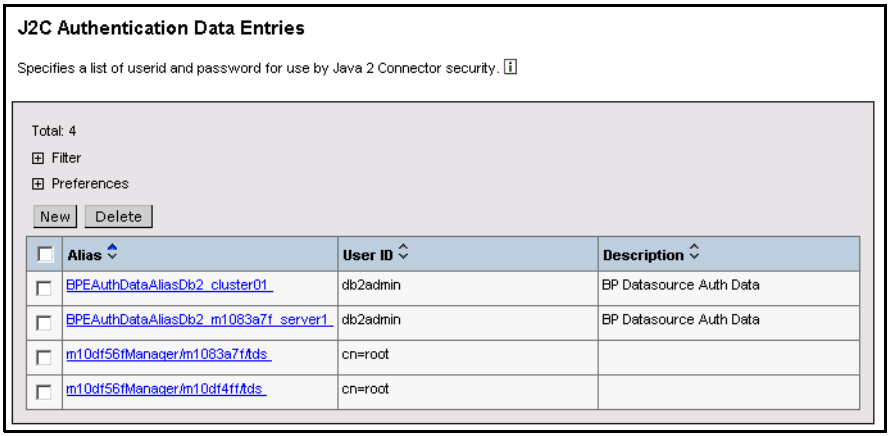


Figure 4-18 J2C Authentication Alias in a Scalable Environment

2. In order to browse to the appropriate node, use the Scope property sheet as shown in Figure 4-18.

Important: Staff Plugin Provider for Business Process Container is only defined at the node level. In order to make the Staff plug-in functional in a clustered Business Process Container, each node must be configured individually for the Staff Plugin Provider.

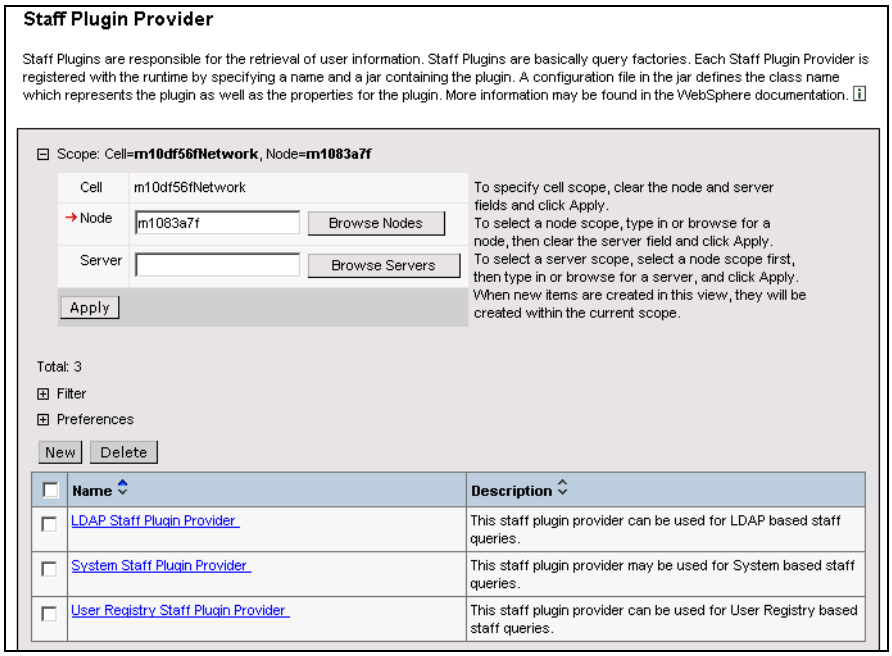


Figure 4-19 Staff Plugin Provider scope at the node level

Important: While configuring the Staff plug-in at the node level, make sure to use the appropriate J2C authentication aliases that are specific to the node the Staff plug-in is being configured for.

4.5 Configuring for high availability

A high availability configuration builds on the scalable topologies discussed in the previous section, adding protection from outages. In some cases, this could be as simple as adding a number of nodes to the WebSphere Business Integration Server Foundation cluster to allow the application to continue executing if one of the nodes fails.

It is also important to consider the supporting components such as the JMS provider, database server and LDAP server. These will need to be protected from availability. If WebSphere MQ is the JMS provider, it will probably require a clustered environment using persistent queues, with some form of takeover of transitional data. An RDBMS is likely to be either hot standby or a parallel server environment. LDAP will also need to be configured in a highly available mode.

A possible high availability topology is shown in Figure 4-20.

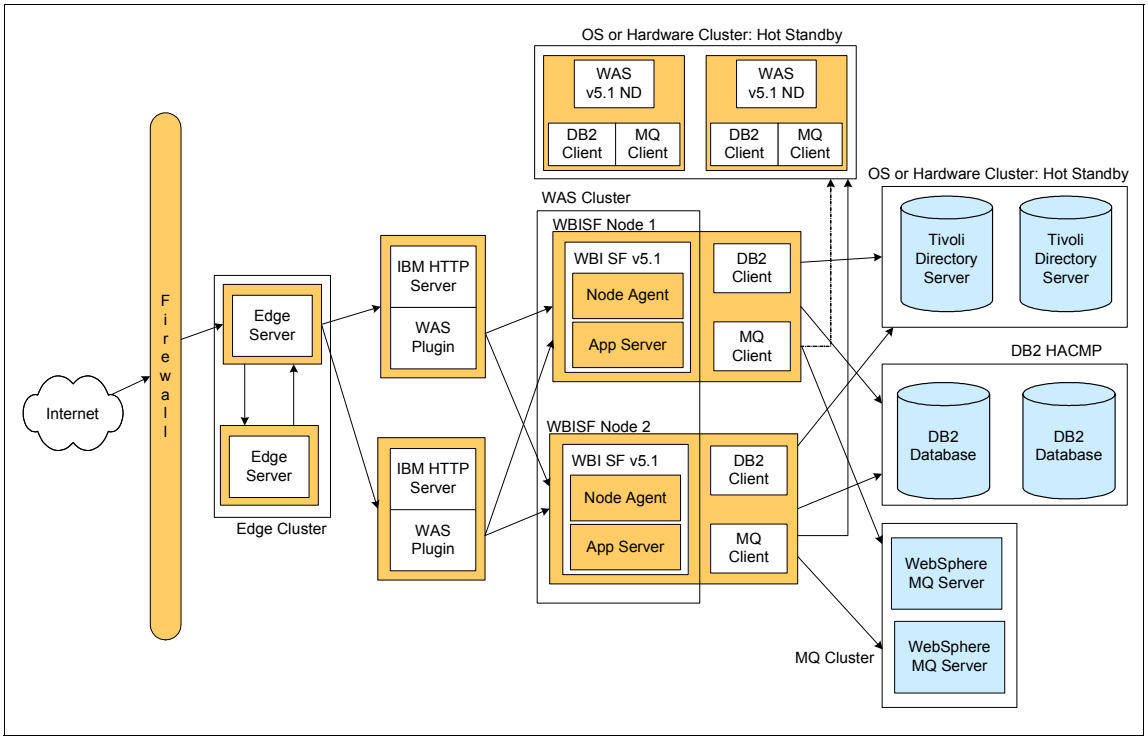


Figure 4-20 High availability environment

Some of the features of this environment are listed below:

- High availability for the WebSphere Business Integration Server Foundation is provided by the WebSphere Application Server cluster mechanism. This, however, depends on the other components in the infrastructure.
- The WebSphere Application Server Network Deployment server has a hot standby, since there can be only one active Network Deployment server in the environment.
- The WebSphere MQ JMS provider is configured in a clustered environment.
- The Data stores, DB2 Enterprise Server Edition and Tivoli Directory Server have a redundant node topology, configured with data replication so that operation can transfer to the hot standby in the event of an outage.

5

Chapter 5.

Development environment

This chapter introduces the WebSphere Business Integration Server Foundation development environment. We also include a brief discussion of what makes up an ideal WebSphere environment.

This chapter also describes features available in WebSphere Studio Application Developer Integration Edition that aid in the development of Business Processes, along with features that are part of the new version, V5.1

Finally, the WebSphere Test Environment is described, where Business Processes can be debugged and tested. To support development and testing of Business Processes, a remote test server is described. The discussion covers the options included and the pros and cons associated with each remote test server configuration.

5.1 Introduction

System architects, application architects, system administrators and developers are often faced with the challenge of creating a coherent development, system testing, and production environment. The challenge is to have identical system testing and production environments, and, more importantly, to be able to test applications in a similar system and integration test environment to the production environment.

When architecting an environment for application development and system testing, much consideration and planning is required. A successful development environment may involve rigorous software engineering, strong architecture, detailed design, quality staffing, careful planning, risk management, and other aspects.

WebSphere Studio Application Developer Integration Edition V5.1 is the application development environment for WebSphere Business Integration Server Foundation V5.1. It can be used to create everything from personal Web pages to Web sites that serve as front ends for e-business applications. WebSphere Studio can be integrated with other content development tools, and with complex applications that interact with enterprise information systems (EIS).

WebSphere Studio Application Developer Integration Edition allows for developing, testing, deploying, and managing applications. It is designed to be used in a wide variety of development roles such as Web developer, Java developer, enterprise programmer, business analyst, and system architect. A rich set of utilities and wizards helps simplify common tasks so that developers can concentrate on providing true business value and on rapidly getting robust applications into production.

In addition to a full Java 2 Platform, Enterprise Edition (J2EE) development environment, WebSphere Studio Application Developer Integration Edition V5.1 provides easy-to-use tools for creating reusable services out of a variety of back-end systems and for choreographing the interactions between those services using Business Process Execution Language for Web Services (BPEL4WS).

The WebSphere environment

Figure 5-1 on page 87 shows an "ideal" WebSphere environment. It takes into account all the necessary steps in a software development process and can be considered a first-class rigorous environment. Every individual sub-environment is a complete environment in itself to achieve a necessary task in a software development, testing, and deployment process.

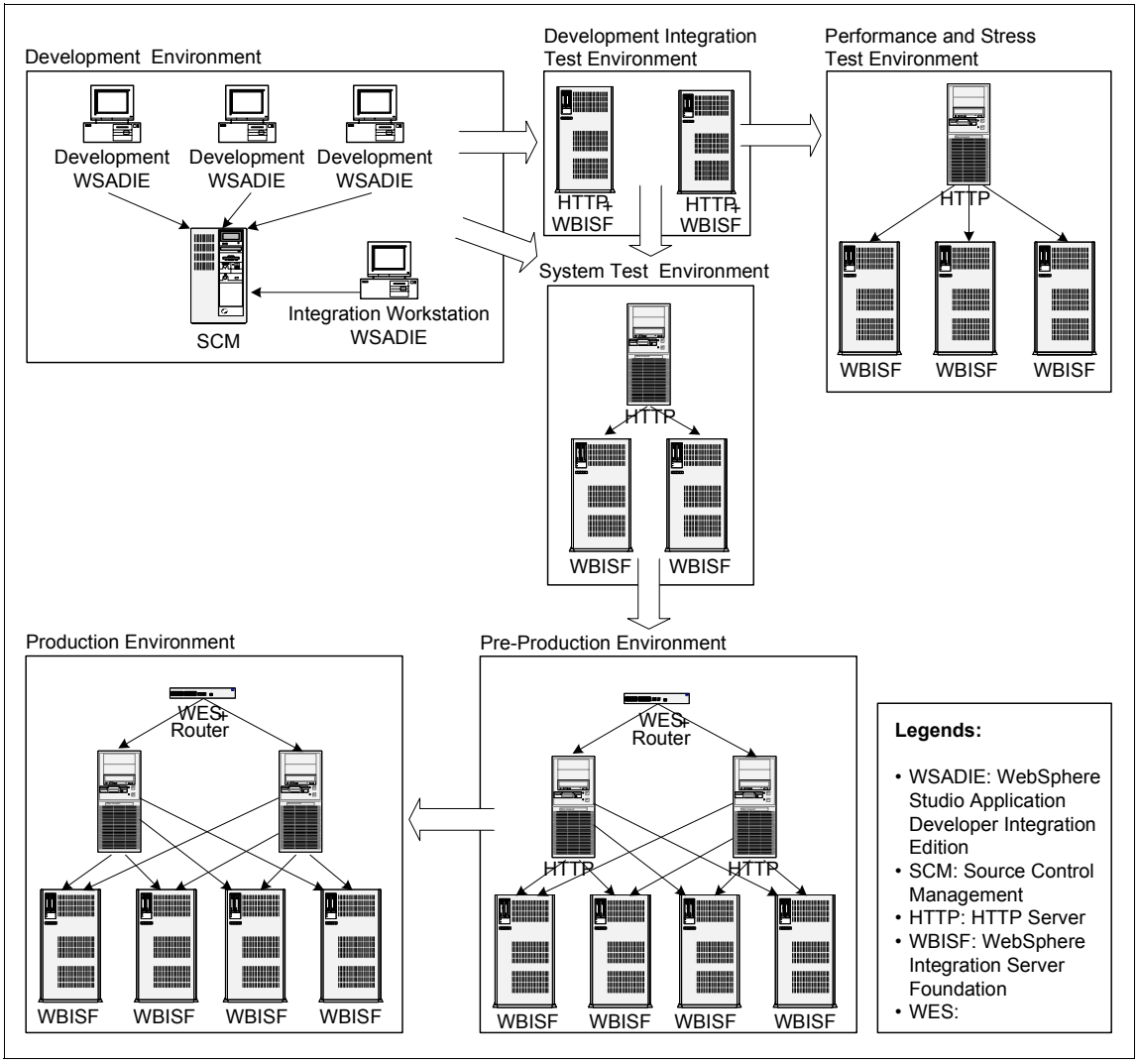


Figure 5-1 WebSphere Business Integration Server Foundation development environment

The illustrated environment requires considerable investment in terms of both time and money by an organization. It also depends on the organization as to the amount of risk it is willing to accept when development, testing, and deploying the application. At a minimum, any major software development project has a development, system test, and production environment. Organizations developing simpler applications or that are willing to assume increased risk may scale down some aspects of this ideal environment.

This redbook advocates the use of separate environments for software development, development integration, system testing, stress testing, pre-production and production.

Development environment

The development environment typically serves as the testing ground for new code, applications, and alterations of the WebSphere environment. This is where developers live and work every day. Thus, they need the best tools and the fewest barriers to progress.

There is generally a code integration environment in the development environment, which is dedicated to code integration and testing. The team or development lead should load the code onto the integration workstation and run the entire suite of unit tests.

Development integration test environment

The development integration test environment is the environment that most closely resembles the production environment, though still at the smallest possible scale. When new code is developed in the development environment and deemed ready for testing, it is moved into the test environment, where it undergoes a rigorous test plan. Testing in this environment involves uncovering issues related to subtle differences between the development and production systems as well as testing the deployment procedures. This may include such things as use of various operating system services, WebSphere Application Server security, back-end systems, and others. Developers use this environment to perform integration tests among all system components. This environment is also used to test installation and operational procedures which are often operating system specific.

system test environment

The system test environment is a carefully controlled formal test environment. A system test environment mirrors the production environment more closely than does a development integration environment, at the smallest possible scale. A key aspect of the system test environment is formality. The purpose of this environment is to ensure that the application will truly deploy and run as required in production. Thus, the system test team is responsible for testing all aspects of the application, including both functional and non-functional requirements.

Performance and stress test environment

Performance and load testing is performed to find load-related problems in applications. This testing requires highly specialized skills and equipment in order to be optimally performed. Hence, this is a dedicated environment and team. Like the system test environment, the performance test environment is a

carefully controlled formal test environment. Development teams run their applications on this environment on an even less frequent basis. A performance test environment mirrors the production environment in complexity, but it does so on the smallest possible scale.

Pre-production environment

This environment serves three purposes:

1. It gives the operations team a final place to familiarize itself with the application and its procedures.
2. It provides the opportunity to test unrelated applications running together. This is crucial when dealing with shared deployment environments. Prior to this point, the applications have been tested and built independently.
3. It provides the operations team with a chance to test their operational procedures (back-up, failover, problem resolution, etc.).

Production environment

This is where the application actually runs. The key point is that if you have carefully followed procedures up to this point, the actual roll into production will be comfortably predictable, since everything will already have been tested.

The following sections discuss WebSphere Studio Application Developer Integration Edition as the IBM preferred development environment for developing business processes and applications.

5.2 WebSphere Studio Application Developer Integration Edition V5.1

WebSphere Studio Application Developer Integration Edition V5.1, optimized for developing applications that deploy to WebSphere Business Integration Server Foundation V5.1, delivers an application development environment for building service oriented applications that extend and integrate your existing IT assets.

5.2.1 WebSphere Studio Application Developer Integration Edition V5.1 at a glance

WebSphere Studio Application Developer Integration Edition, at its core, provides easy-to-use tools for creating reusable services out of a variety of back-end systems and for choreographing the interactions between those

services using Business Process Execution Language for Web Services (BPEL4WS).

The following are the core differences between WebSphere Studio Application Developer and WebSphere Studio Application Developer Integration Edition.

Services

At the heart of the Integration Edition programming model are business services, which are used to model different kinds of service providers consistently. Services are the business functions of your enterprise, or of your business partners. Use the integration tools to develop various types of services, including Web services, processes, EIS (J2EE Connector) services, JMS services, and so on. Web Services Descriptor Language (WSDL) is used as the model for describing any kind of service.

Business process

A business process is a service implementation that represents a part of your business operations. It can be completely automated or may require human interaction at certain points. WebSphere Studio Application Developer Integration Edition provides a model that allows you to implement these processes in a very efficient, graphical way.

Human workflow support

Human workflow support expands the reach of BPEL to include activities requiring human interaction as steps in an automated business process. Business processes involving human interaction are interruptible and persistent (a person may take a long time to complete the task) and resume when the person completes the task.

Business rule beans

Business rule beans offer a powerful real-time framework for defining, executing, and managing business rules that encapsulate business policies that vary based on changes in the business environment. For example, a simple business rule might be, "If a customer’s shopping cart is greater than $X, then offer a Y% discount."

Programming Model Extensions

WebSphere Business Integration Server Foundation V5.1 and WebSphere Studio Application Developer Integration Edition V5.1 help accelerate large-scale application development by allowing the user leverage the latest innovations that build on today's J2EE standards including extended messaging, Dynamic Query service, internationalization service, application profiling, asynchronous beans,

Object pools, Startup beans, scheduler service, work area service, activity session service, and last participant support.

Additional new features in V5.1

WebSphere Studio Application Developer Integration Edition V5.1, as compared to V5.0, includes the following new features:

- Business process designer for creating Business Process Execution Language for Web Services 1.1 (BPEL4WS) process flows.
- Integrated visual BPEL4WS debugger.
- Enhanced performance for installing and debugging, including support for J9 Hot Swap.
- New visual condition builder to direct the execution of BPEL processes.
- Automated migration of process flows from Flow Definition Markup Language (FDML) to BPEL4WS.
- Unit test environment for WebSphere Business Integration Server Foundation 5.1
- Full support for all the features included in WebSphere Studio Application Developer Integration Edition V5.1.1 including support for Workbench V2.1.2, support for building Web Services Interoperability (WS-I) compliant Web services, and support for Java development kit (JDK) 1.4.1
- Programming Model Extensions for a distributed map. This PME offers an interface to enable J2EE applications and system components to cache and share Java objects by storing a reference to the object in the cache in order to improve performance
- Another Programming Model Extension is *Container Manager Persistence over Anything*. It extends the Container Managed Persistence (CMP) framework to support any back-end system or service that supports create, retrieve, update, and delete (CRUD) methods.

5.2.2 WebSphere Studio Application Developer Integration Edition Workbench

WebSphere Studio configurations are all built on the WebSphere Studio Workbench that extends the open-source Eclipse platform and provides an open, extensible plug-in architecture. Numerous plug-ins are available from partners and the open source community, or, using the included plug-in development environment, you can create your own plug-ins for specific needs.

Figure 5-2 on page 93 highlights some of the features in WebSphere Studio Application Developer Integration Edition workbench. Studio Integration Edition

workbench includes Business Integration perspective. This is the default perspective for development enterprise services. Business Integration perspective is highlighted in Figure 5-2 on page 93.

Business Integration perspective includes a process editor for composing and editing BPEL-based processes. This view is complemented by a set of tools that aids in the development of BPEL-based processes. Studio Integration Edition workbench include Business Process based views, that is, the Outline view shows components that are necessary for Business Process creation.

The Business Integration perspective BPEL Editor and the tooling are further discussed in Chapter 6, “Process choreographer: introduction” on page 121.

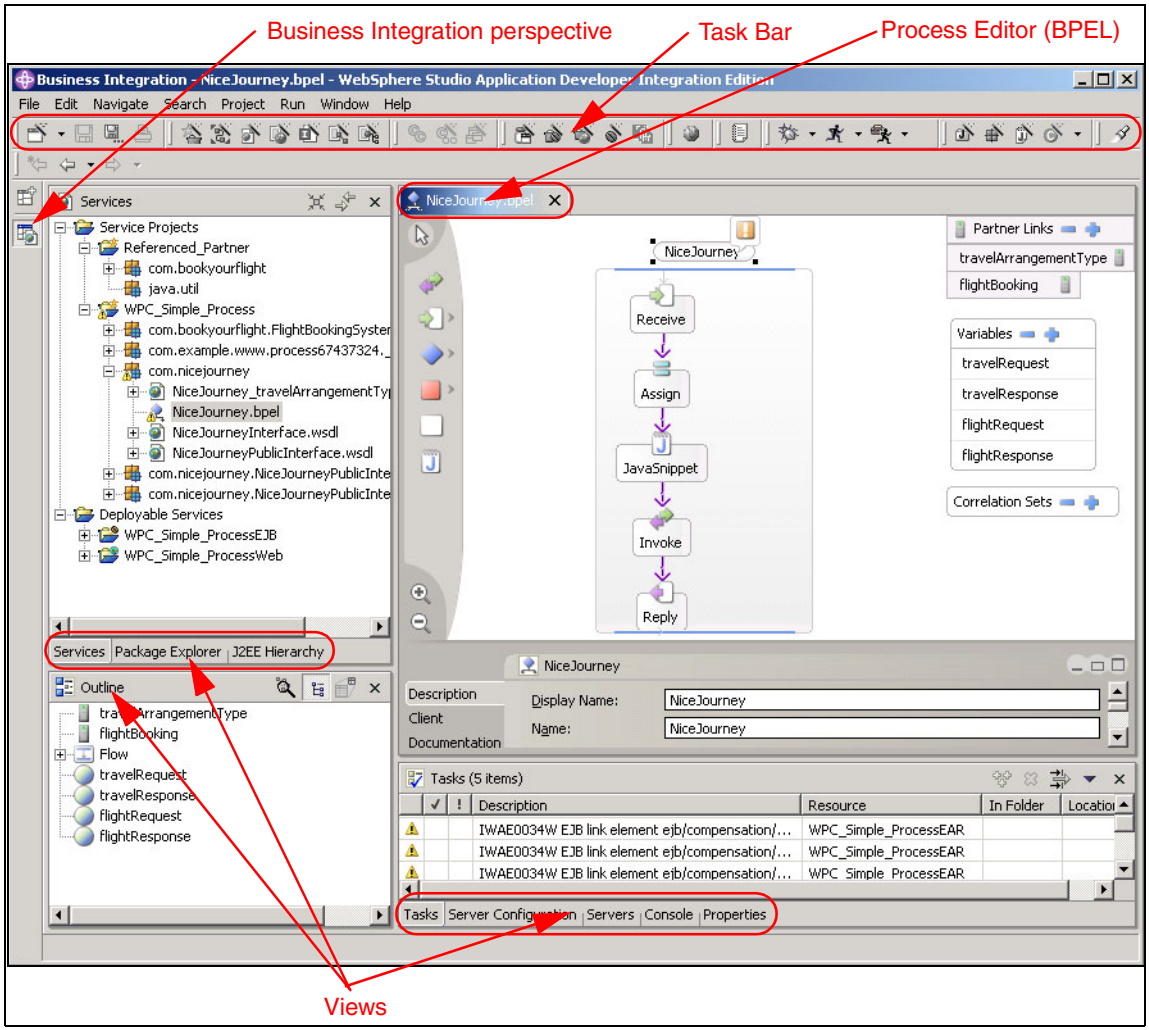


Figure 5-2 WebSphere Studio Application Developer Integration Edition V5.1 Business Integration perspective

The context-menu in the Services view of Business Integration perspective is shown below. Items in this menu are discussed further in next section.

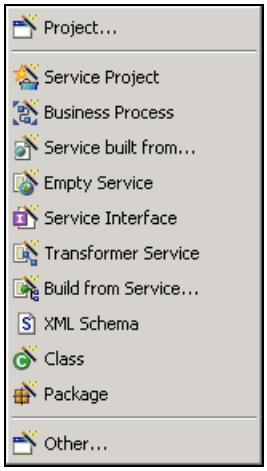


Figure 5-3 Business Integration perspective Services context menu

5.2.3 Integration Edition tooling

WebSphere Studio Application Developer Integration Edition V5.1 has complementary tools for aiding in the development of Services and Business Processes.

Getting Started

Users can start working with WebSphere Studio Application Developer Integration Edition immediately. A welcome page provides links to key concepts. Cheat sheets are a useful way of quickly building a service. Topology diagrams provide a map of the kind of service being developed.

Business Integration perspective

This is the default perspective for the Integration Edition. It puts together all the tools, services, and processes used to develop enterprise services.

Service Project

This is the center of Integration Edition as it manages the development of services and processes. This also manages the files associated with the project. The files are WSDL files, Java, and process files.

Integration Application

An integration application is an application created from a set of services.

Service Definition

A service definition is a WSDL document. Effectively, this is a set of three files that collectively conform to the WSDL specification of a service definition.

Integration Edition provides a wizard to help build the WSDL document. There is an Empty Service wizard that generates a WSDL interface with only a definition name and target namespace. Once a service definition is created, Integration Edition provides an editor for modifying it from several viewpoints.

Services Based on existing assets

Services are usually based on existing applications that have been tested and represent considerable investment. Basing a service on an application is considered a bottom-up approach. Integration Edition provides a *Service built from* wizard, to base a service on a Java application, a stateless session bean, or on one of the resource adapters. It is possible to use a custom resource adapter using the JCA plug-in tool.

There are importer wizards for using existing applications. You can import C files, COBOL copybook files, HOD 3270 terminal files, Java files, or MFS files. In addition, the JCA plug-in can also be used to import RAR files to the Integration Edition development environment.

Processes

A process is a form of service implementation. In the implementation, you can use existing services arranged in a sequence to create another service. Integration Edition provides a Process editor that arranges and links services into a process; a Process debugger that visually debugs errors in a process; and a Process Web client to test a process.

Transformer Editor

Messages define the input and output data for an operation in a WSDL file. They also define the content of a variable in a process. The transformer editor transforms a message by mapping it to another message. Transforming messages with this editor is often done when building a process.

Services Based on implementing Service interfaces

A service based on implementing a service interface is a top-down approach to development, unlike a service based on an existing asset. To achieve this, Integration Edition provides the Build from service wizard. Generally, this editor is used when a service interface file exists but there is no implementation of it. It allows you to develop an implementation for the WSDL file.

Generate Deploy Code wizard

The Generate Deploy Code wizard generates the deploy code for the services you want to offer. The deploy code gets generated into modules in the Enterprise Archive (EAR) file to become your integration application on deployment to a production server.

Service Proxy wizard

The Service Proxy wizard creates a file to represent your service on a client. An application can interact with it as if it were just another Java module in the application. The application simply invokes the proxy's methods, and the proxy transparently handles getting the data from service at the server.

5.2.4 Development with WebSphere Studio Application Developer Integration Edition

This section provides details about development-related topics in WebSphere Studio Application Developer Integration Edition.

Server targeting support

WebSphere Studio Application Developer Integration Edition V5.1 is the primary development environment for WebSphere Business Integration Server Foundation V5.1. It not only supports the Integration Server test environment, but also supports the extensions to develop J2EE applications.

In order to enable the support for the J2EE extensions, you have to enable the server targeting support in WebSphere Studio Application Developer Integration Edition.

Important: The server targeting support is not enabled by default. If you develop for WebSphere Business Integration Server Foundation, you may want to enable this feature as the very first step.

Server targeting support is stored in the workspace preferences (workspace scope), as every other Eclipse preference.

To enable, select **Window** → **Preferences** from the menu, then select **J2EE** from the list on the left side. On the panel, shown in Figure 5-4 on page 97, check the **Enable server targeting support**, then click **OK**.

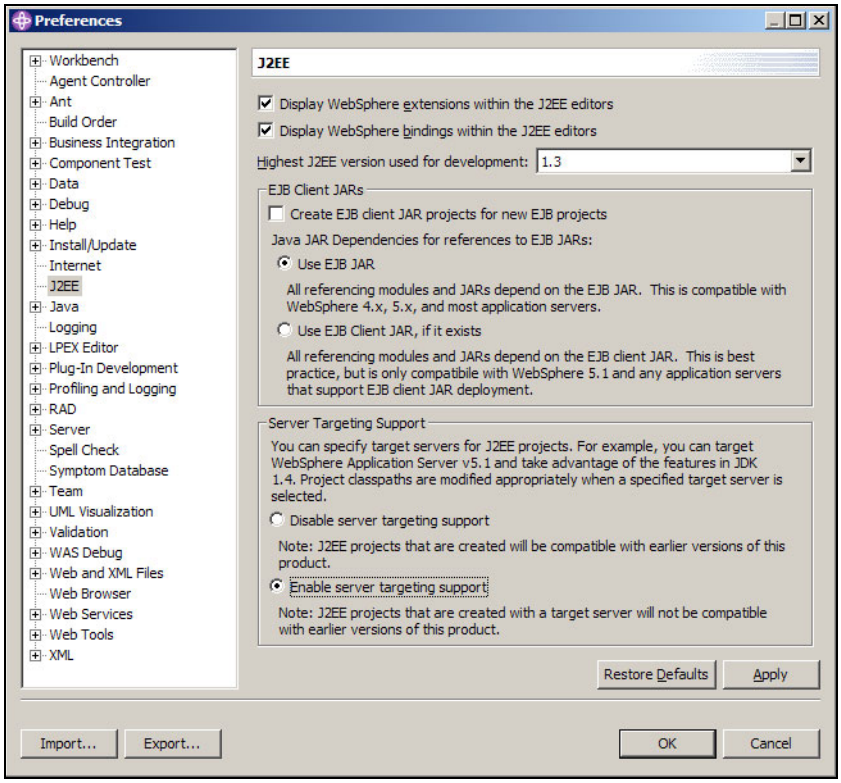


Figure 5-4 Preferences

Once you have server targeting support enabled, you can create J2EE applications and import J2EE applications with the support for extensions.

When you create a new Enterprise Application Project, you can select the target server on the wizard panel.

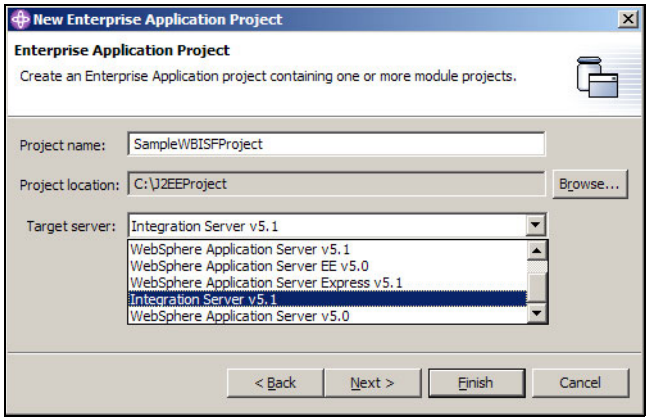


Figure 5-5 New Enterprise Application project

When importing an EAR file, you get a window similar to Figure 5-5, where you can select the targeting server.

If you already have a J2EE project on your workspace that has no server targeting support, you can change the project to enable it in the project properties panel. Right-click the EAR project folder, then select **Properties**, select **J2EE** on the panel, then change the Target Server to `Integration Server V5.1`, as show below.

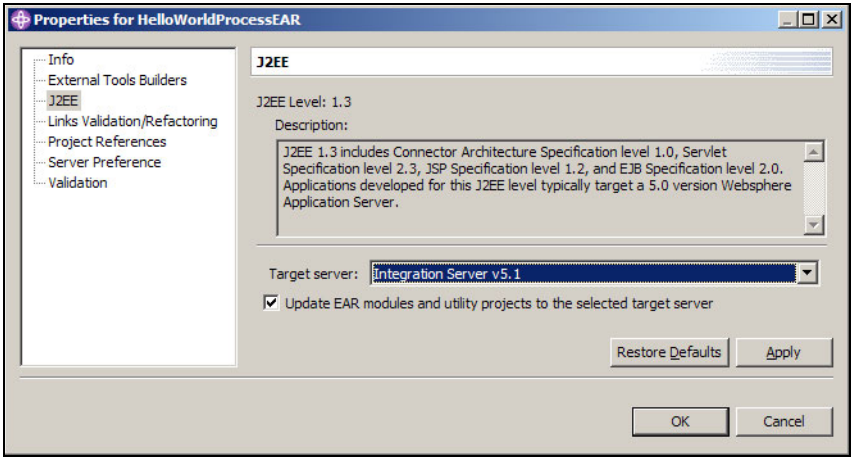


Figure 5-6 J2EE application properties

Important: You have to have server targeting support enabled in WebSphere Studio Application Developer Integration Edition in order to select or change the server target to Integration Server v5.1. Otherwise, the option is not enabled.

The differences that enabling server targeting makes can be seen in the deployment descriptors, for example in the application descriptor.

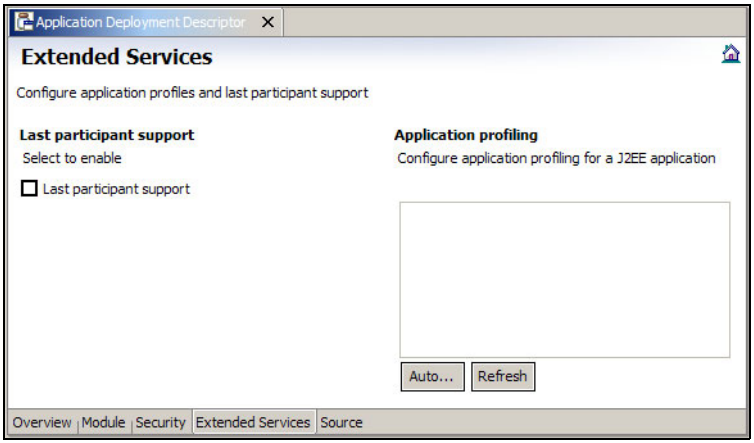


Figure 5-7 Extended services in the application.xml

Another difference is in the EJB deployment description.

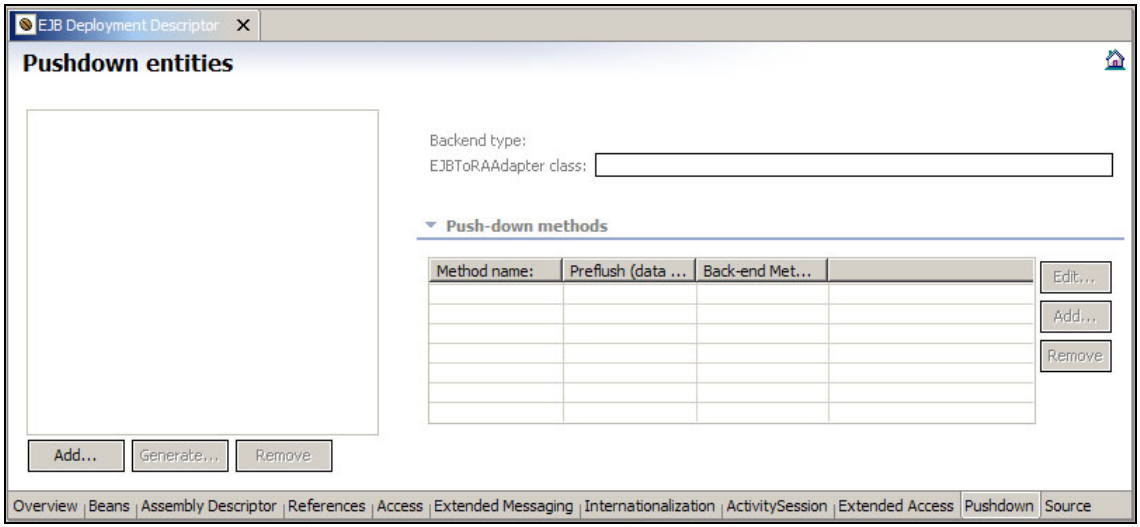


Figure 5-8 Extensions in the ejb-jar.xml

Note the Extended Messaging, Internationalization, ActivitySession, Extended Access and Pushdown tabs on the editor.

BPEL Editor

The BPEL Editor is a visual graphical user interface (GUI) for developing BPEL4WS processes. Since the editor is a brand new component in WebSphere Studio Application Developer Integration Edition and it is quite a complex one, you may encounter some unexpected behavior. The following list is a series of recommendations to avoid problems in the future using the BPEL Editor.

- Make sure you save your work regularly in the BPEL Editor. On the other hand, do not make it an automatic process, because you may overwrite your work after an unexpected change in the editor.
- When the diagram does not refresh in the editor, the only option might be to save the work, close the file, then open the file again in the editor.
- Some of the components can be moved around, opened/closed or expanded/collapsed. The hot areas, where the pointer becomes active when you click, might be very small and can be at unexpected locations. Be patient and try to find the right spot for your actions (click).
- When you are working with large processes, build your processes using embedded flows and sequences. With this method, you can collapse and expand parts of your process so you can see it and handle it better.

Unfortunately, some functions of the visual design need further advancement, including the zoom, the automatic arrangement and component positioning. Be patient and accept the fact that you may have to do some extra work just to arrange your components and move them where you can see or edit them.

Enterprise services

Enterprise services offer access over the Internet to applications in a platform-neutral and language-neutral fashion. They offer access to enterprise information systems (EIS) and message queues and can be used in a client/server configuration without the Internet. Enterprise services can access applications and data on a variety of platforms and in a variety of formats.

An enterprise service wraps a software component in a common services interface. The software component is typically a Java class, EJB, or J2C resource adapter for an EIS. In services terminology, this software component is known as the implementation.

WebSphere Studio Application Developer Integration Edition offers two approaches for building enterprise services:

- Bottom-up approach
- Top-down approach

Bottom-up approach

Use the bottom-up approach when building an enterprise service from existing assets. In the bottom-up approach, you start with an existing implementation and then add the description for implementation in WSDL. The WSDL is generated by the tools in the Integration Edition.

You can generate enterprise services for Java classes and EJBs. Integration Edition can also generate enterprise services that support J2C resource adapters to access EISs.

The main steps to create and install an enterprise service are as follows.

1. Create a service project.
2. Develop or import the implementation into the service project.
3. Generate the service.
4. Generate the deployed code.
5. Deploy the enterprise service as an EAR file to an application server.

Top-down approach

Use the top-down approach to create an implementation from an interface WSDL file. In the top-down approach, you create the service definitions first and then generate the implementation for the service from the service definition.

If you already have a WSDL file that contains the service interface definitions (port types, messages and operations), you can then follow these steps to add bindings and port definitions for the service and generate either a Java or EJB skeleton for the implementation of the service:

1. Create a service project.
2. Import the interface files (WSDL).
3. Create an EJB project if an EJB skeleton should be generated (this step is not necessary when creating a Java skeleton).
4. Generate the service skeleton using the New Service Skeleton wizard.
5. Implement the business logic in the skeleton.
6. Deploy the enterprise service as an EAR file to an application server.

More information

Developing enterprise services is beyond the scope of this redbook. For detailed information about development, refer the IBM Redbook: Exploring WebSphere Studio Application Developer Integration Edition V5, SG24-6200.

You can also find details about enterprise services in the WebSphere Studio Application Developer Integration Edition help at **WebSphere Studio** → **Developing** → **Enterprise services**.

5.3 WebSphere Test Environment

WebSphere Studio Application Developer Integration Edition V5.1 provides runtime environments that allows for testing JSP files, servlets, HTML files, enterprise beans, Java classes, and enterprise services including BPEL and FDML based processes. A universal test client (UTC) is provided to test EJB modules, Web services and BPEL4WS processes. WebSphere Test Environment also provides the capability to configure other local or remote servers for integrated testing and debugging J2EE applications.

The server tools allow you to test applications in different local or remote runtime environments. You can also use the tools to publish to these environments. The following two options are provided by the server tooling for publishing J2EE applications:

- WebSphere Studio Application Developer Integration Edition V5.1 includes WebSphere Business Integration Server Foundation V5.1 runtime environment, which can be used to test applications directly from the development environment. Each test environment provides all the function of the full runtime environment, but eliminates dependencies on network connections.

- It is also possible to publish to one or more separately installed versions of WebSphere Business Integration Server Foundation V5.1 application servers that reside either locally or on a remote machine. If the test server is remote, Agent Controller must be installed on the remote machine.

In WebSphere Studio Application Developer Integration Edition, the server tools contain the complete runtime environment of the WebSphere Application Server including WebSphere Business Integration Server Foundation V5.1. This environment is called the WebSphere Test Environment. The test environment includes a local copy of the full WebSphere Business Integration Server Foundation V5.1 and various WebSphere Application Server V5 runtime envirnments. Optionally, during installation of WebSphere Studio Application Developer Integration Edition V5.1, you can install WebSphere Application Server V4 test environments.

5.3.1 WebSphere Test Environment benefits

WebSphere Test Environment provides the following benefits:

- Standalone all-in-one testing
- No dependency on WebSphere Application Server installation or availability
- No dependency on an external database even when entity bean support is required
- Supports hot-code replace
- Provides the ability to debug live server-side code at full speed
- Supports configuring multiple Web applications
- Supports multiple servers that can be configured and run at the same time
- Provides access to the profiling feature
- Provides the ability to version server configurations
- Provides a Universal Test Client where you can test your enterprise beans and Java classes
- Provides access to the WebSphere Application Server Administration Console
- Provides the ability to automatically create tables and data sources for testing CMP beans
- Supports sharing servers with multiple development clients

The following table summarizes the supported runtime environments in WebSphere Studio Application Developer Integration Edition V5.1.

Table 5-1 WebSphere Application Server types

Version	Types
WebSphere Application Server V5.1	- WebSphere Business Integration Server Foundation - WebSphere Business Integration Server Foundation [remote] - Express Server [remote] - V5.1 Server [remote]
WebSphere Application Server V5.0	- Express [remote] - EE Remote Server [remote] - V5.0 Server [remote]
WebSphere Application Server V4.0	- V4.0.7 Server [remote]
Apache Tomcat V4.1	Local Server Test Environment
Apache Tomcat V4.0	Local Server Test Environment
Apache Tomcat V3.2	Local Server Test Environment
Other	- Application Server Attach - J2EE Publishing Server - Static Web Publishing Server - TCP/IP Monitoring Server

The WebSphere Application Server V4 test environment is based on WebSphere WebSphere Application Server V4.0.7. The WebSphere Application Server V5 test environment is based on WebSphere WebSphere Application Server V5.0.2. The WebSphere Application Server V5.1 test environment is based on WebSphere Application Server V5.1. When you migrate from a previous version of WebSphere Studio, any e-fixes to the WebSphere Test Environment will be removed and you must reinstall them manually.

5.3.2 WebSphere Test Environment overview

Components that make up the WebSphere Test Environment are illustrated in Figure 5-9 on page 105. These components are dicussed briefly in the following sections.

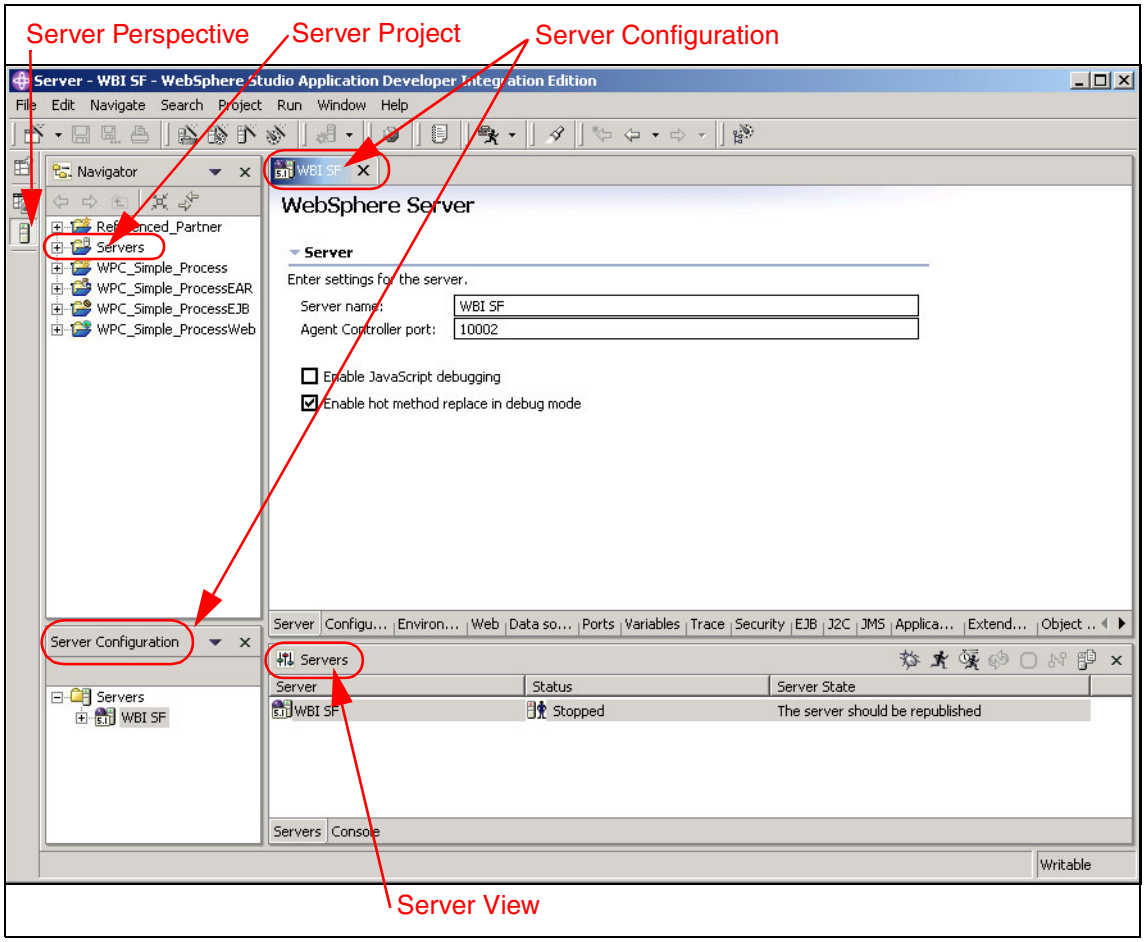


Figure 5-9 WebSphere Test Environment

Server perspective

WebSphere Studio provides a server perspective that allows you to manage servers and server configurations. A Server Configuration view allows you to create or delete servers and server configurations. The Servers view gives you the ability to stop, start, and restart servers; launch a server in debug or profiling mode; and publish. The Console view allows you to monitor runtime messages.

Server definition

The server tools use servers and configurations to test and publish projects. Servers are definitions that identify where you want to test projects. Server configurations contain setup information. You can either have the development

environment create the servers and configurations automatically for you, or you can create them using a wizard.

Types of servers

Theserver tools support the following types of servers. This lists summaries different types of servers that are supported by WebSphere Studio Application Developer Integration Edition V5.1:

- WebSphere test environment
- WebSphere local server
- WebSphere remote server
- Tomcat test environment
- Tomcat local server
- J2EE publishing server
- Static Web publishing server
- Remote server attach
- TCP/IP monitoring server
- Application server attach

Integration Server View tools

In addition to the tooling provided in WebSphere Studio Application Developer Integration Edition V5.1 Server View context menu, WebSphere Business Integration Server Foundation V5.1 Server View context menu provides addtional tooling for Editing Business Container Deployment Descriptor, listing the deployed processes and for launching the Web client. This is indicated in Figure 5-10 on page 107.

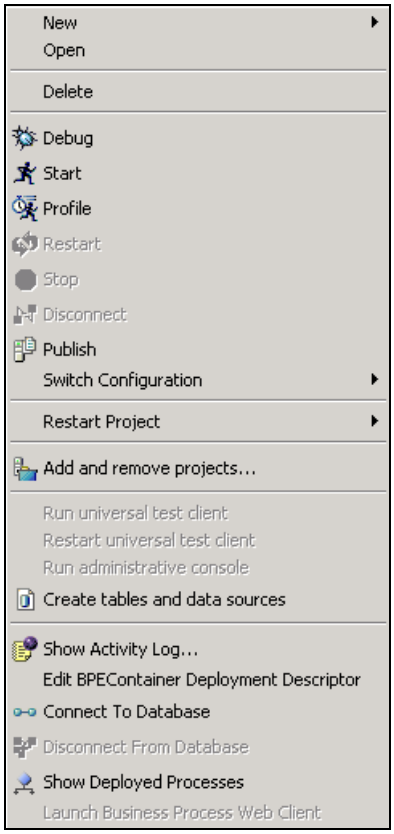


Figure 5-10 Server context-sensitive menu

Relationship between server resources

For each server, you can specify the server configuration that should be used for that server. A server can only point to one server configuration. However, a server configuration can be pointed to from one or more servers.

A relationship between a server configuration and a project, for example an EAR project, is created when a project is added to the server configuration. A server can point to one or more projects. A project can be pointed to from one or more server configurations.

The following figure illustrates the possible relationship between the server tools resources resources

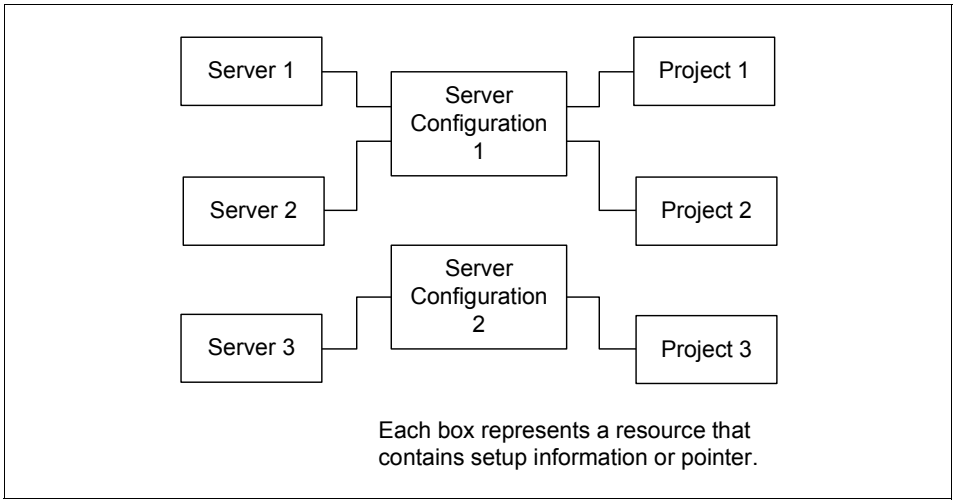


Figure 5-11 Relationship between Server Resources

5.3.3 Supported software components

WebSphere Studio ships two JMS providers for testing Business Processes.

- WebSphere MQ Simulator for Java Developers
- WebSphere MQ Embedded Messaging

The following figure shows the supported JMS Providers in WebSphere Test Environment.

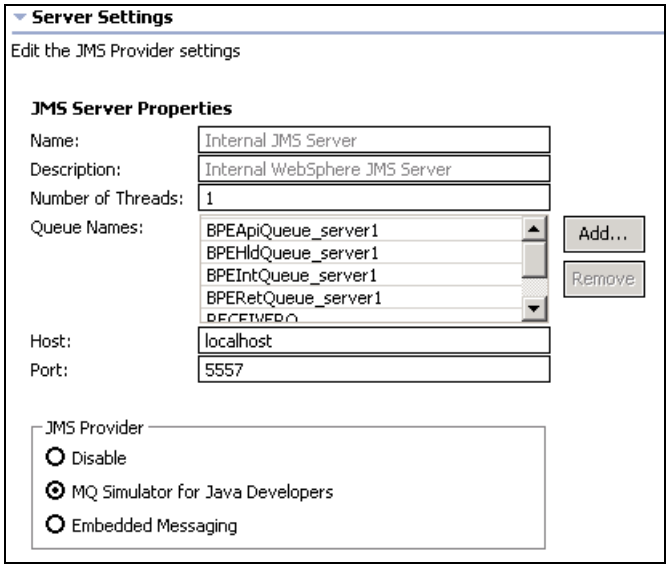


Figure 5-12 Supported JMS Providers

In addition, the following databases are supported by the WebSphere Test Environment:

- IBM DB2
- Cloudescape
- Informix®
- Sybase
- Oracle
- MS SQL Server

Figure 5-13 on page 110 indicates the supported databases in WebSphere Test Environment. However, it is important to note that only Cloudescape is supported for testing in both local and remote WebSphere Test Environments. This point is further discussed in the following sections and in 5.4, “Remote test server” on page 111.

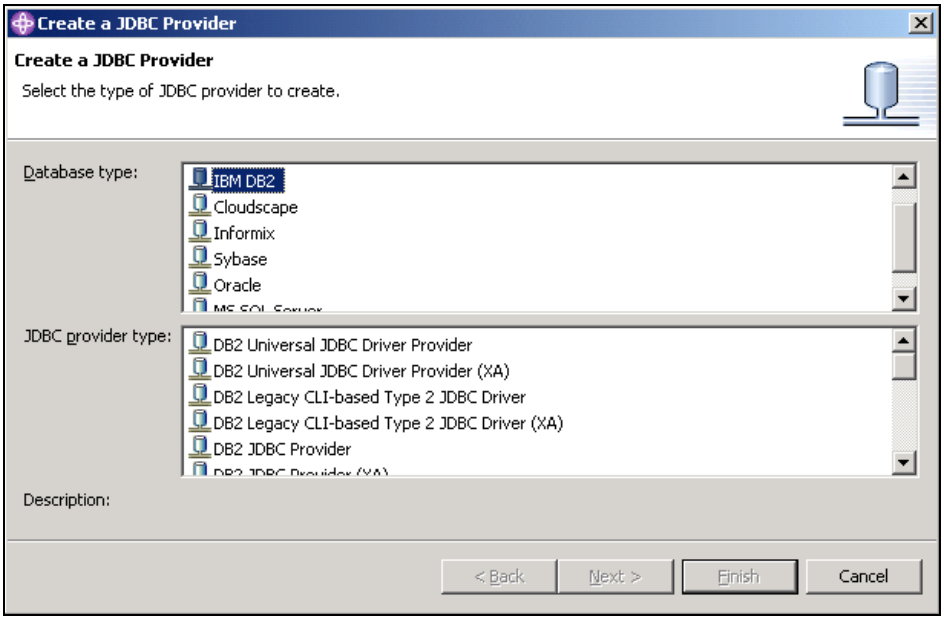


Figure 5-13 Supported Databases in WebSphere Test Environment

Embedded MQ

WebSphere MQ Embedded Messaging allows you to fully test and publish your applications within a JMS runtime environment. When you install the product, you can choose to install MQ Embedded Messaging and once installed, it is set up as the default messaging support for the WebSphere Test Environment.

MQ Simulator for Java Developer

WebSphere MQ Simulator for Java Developers implements Sun's Java Message Service specifications. The JMS provider works with the WebSphere Integration Test Environment so that you can test JMS services that you create with the WebSphere Studio business integration tools.

WebSphere MQ Simulator for Java Developers is an in-process JMS server. It can be used to easily unit test JMS applications within the development environment. It does not support persistence or communication between processes. No configuration is required; it is ready to run in the WebSphere Studio's Test Environment.

WebSphere MQ Simulator for Java Developers is installed with the WebSphere Studio product. WebSphere MQ Embedded Messaging requires separate installation from the WebSphere Studio CD. When MQ Embedded Messaging is installed, it becomes the default JMS provider that will be used by the

WebSphere Test Environment. However, you can switch between these two JMS providers for the Test Environment.

Restriction: At the time of writing this book, WebSphere MQ was not tested with Business Process container as a JMS provider in WebSphere Test Environment.

Cloudscape™

Process choreographer only supports the embedded Cloudscape, which does not allow remote access. Cloudscape Network Server is not supported because it has no XA support.

Restriction: Currently, you must use a Cloudscape database if you want to test FDML or BPEL business processes using either a WebSphere remote server or test environment server in the WebSphere test environment of WebSphere Studio.

IBM DB2 UDB

Using IBM DB2 UDB in the WebSphere Test Environment together with the Integration server V5.1 is not supported. You can still use DB2 for application data together with other applications.

5.4 Remote test server

For more information about how to set up a single server for remote testing refer to 4.2, “Basic configuration” on page 36.

5.4.1 Agent Controller

The Agent Controller is a daemon process that enables client applications to launch host processes and interact with agents that coexist within host processes. A single configuration file is used to manage the extent of its behavior.

The Agent Controller provides a means for extending application behavior so that information regarding the application's execution can be externalized and then collected either locally or remotely. The Agent Controller interacts with the following components:

Host process

This is the process that contains the application under test.

Agent

A reusable binary file that provides services to the host process, and more importantly, provides a portal by which application data can be forwarded to attached clients. A host process can have one or more agents currently running within it. Even if the host process does not contain an agent initially, some processing condition can result in the creation of an agent at some point during the life cycle of the process.

Client

A local or remote application that is the terminal destination of host process data that is externalized by an agent. A single client can be attached to many agents at once. However, a client does not always have to be attached to an agent.

Agent Controller

A daemon process that resides on each deployment host and provides the mechanism by which client applications can either launch new host processes, or attach to agents that coexist within existing host processes. The client can reside on the same host as the Agent Controller, or it can be remote. The Agent Controller can only interact with host processes on the same node.

The Agent Controller's deployment model consists of multiple development hosts that use the test client to interact with multiple applications that reside on many different hosts on the network. A simple deployment diagram is illustrated in Figure 5-14 on page 113.

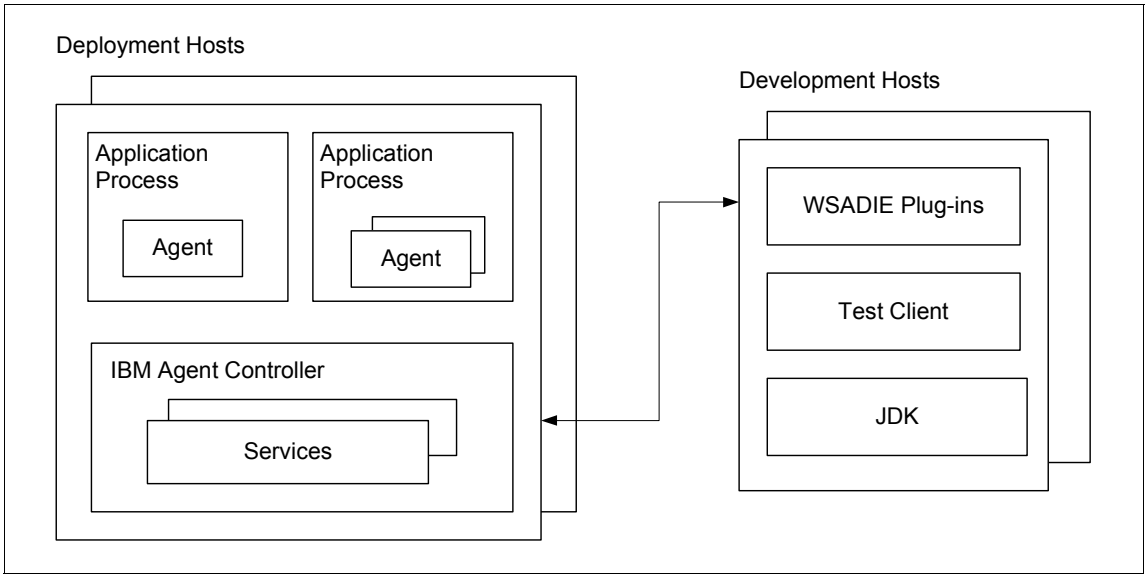


Figure 5-14 Agent Controller deployment model

Each application process can have zero or more agents running within it. Each host also has services that are provided by the Agent Controller. The Agent Controller handles all communication meaning that the test client does not interact with either the services or the agents directly. Instead, the client sends all of its requests to the Agent Controller on the host that contains the application. It is on the deployment hosts that these requests are authenticated and routed to the target agent or service. Commands and data that are generated by the agents and services must also be routed to the Agent Controller, from where they are then transmitted to the test client.

Note: WebSphere workbench plug-ins can leverage the test client and provide an interface for the developer to interact with the Agent Controller on the deployment machines. An example of this is the Profiling perspective

5.4.2 Supported remote server testing scenarios

There are several options available for deploying and debugging a process on a remote server. The following scenarios were tested for the both Process deployment and debugging

WebSphere Studio Application Developer Integration Edition development machine configured with Remote Integration Server (WebSphere Business

Integration Server Foundation) that communicates with Agent Controller running on WebSphere Business Integration Server Foundation machine

WebSphere Studio Application Developer Integration Edition development machine directly communicating with the Business Process Engine for process debugging.

Scenario 1

Advantages:

- Gives the benefit of allowing Profiling, Administration, Configuration through WebSphere Test Environment.
- Debugging is also performed through Debug perspective. Debugging command line can be customized to pass in BPE specific performance enhancements like -Xj9.
- Allows for installation, deployment of processes through WebSphere Test Environment.

Disadvantages:

- Only Cloudscape and Embedded MQ supported on remote machine.
- It is not a true representation of the System Test or Production environment.
- Software components may not exactly match the production and system test environment, (WebSphere MQ and DB2).

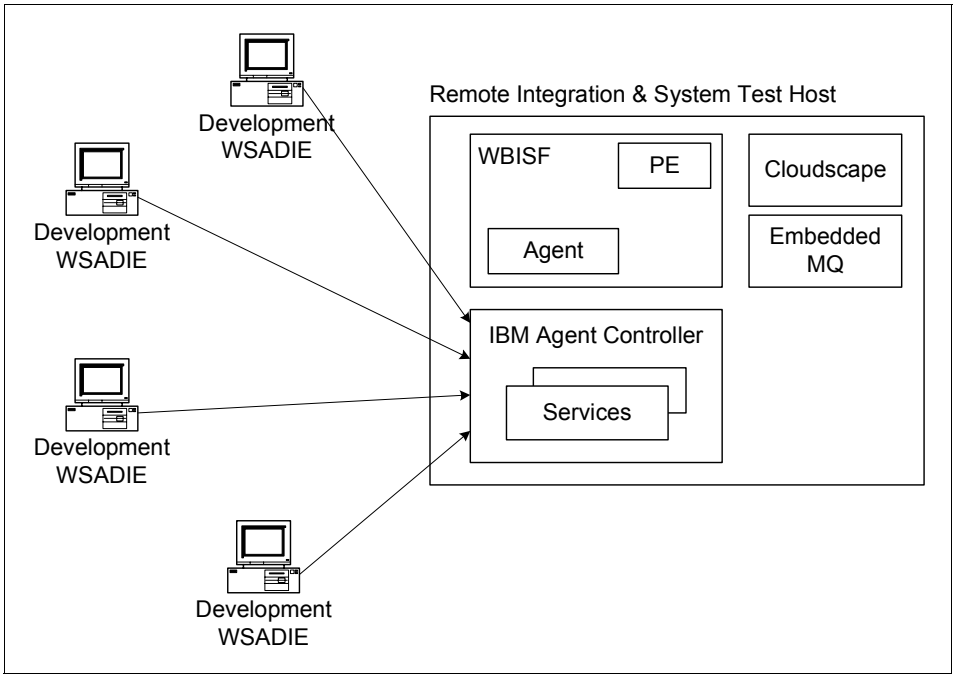


Figure 5-15 Scenario 1

Scenario 2

Advantages:

- Full support of DB2, WebSphere MQ and Tivoli Directory Server
- Development environment much closely resembles the system test and production environment.
- Full debugging support through the debug perspective.

Disadvantages:

- The connectivity is established through WebSphere Studio Application Developer Integration Edition debug perspective by directly attaching to a remote WebSphere Business Integration Server Foundation running process.
- No administration, configuration features are available in WebSphere Test Environment.
- Must have the Java code available in the local workspace for the specified Process being debugged.
- Debugging may not work if deployed process is not exactly the same as the one available in the workspace.

- Remote Application server must be restarted in debug mode and for adding customized debug parameters, that is, -Xj9, Administrative Console has to be used via a Web browser prior to debugging the process.
- No support is available for installation and deployment of processes through WebSphere Test Environment.

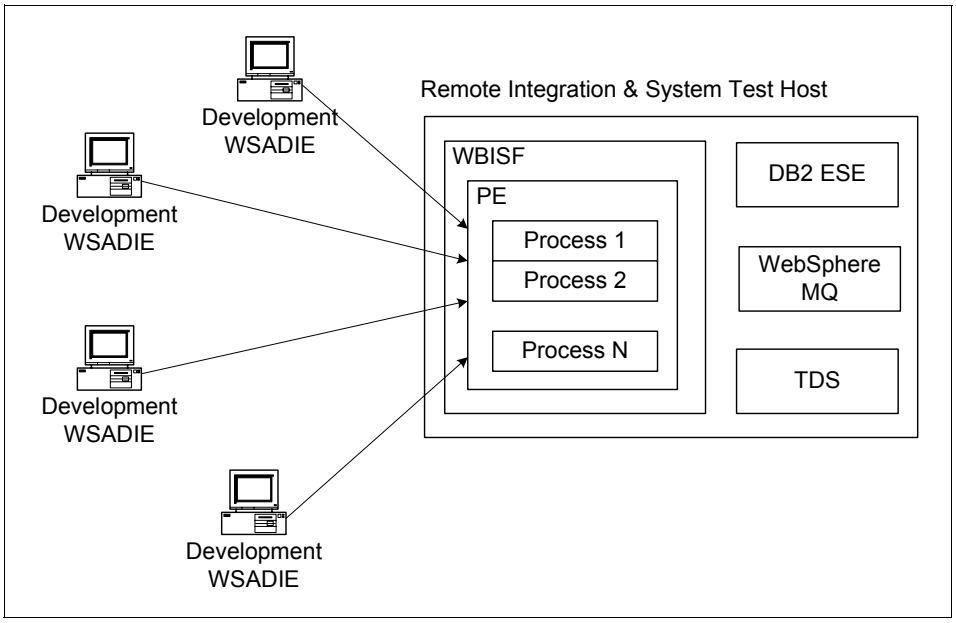


Figure 5-16 Scenario 2

Key:

- WSADIE: WebSphere Studio Application Developer Integration Edition
- TDS: Tivoli Directory Server
- WBISF: WebSphere Business Integration Server Foundation
- PE: Process Engine

5.4.3 Configuring the IBM WebSphere Test Environment for the remote test server

Before you can deploy a process to a remote test server from WebSphere Studio, you need to ensure that you have met the following requirements

- WebSphere Business Integration Server Foundation V5.1 (Integration Server) is installed with the Embedded Messaging feature.

- Ensure that the Cloudscape database BPEDB exists in %WAS_HOME%\ProcessChoreographer.
- IBM Agent Controller is installed and running. Agent Controller is available as a separate install option on the WebSphere Studio CD.

Part 3

Implementing WebSphere Enterprise solutions

6

Process choreographer: introduction

This chapter introduces WebSphere Process Choreographer, the Business Process Engine and the process description language BPEL4WS. It also provides an overview of the business process related terms, the different process types and some considerations. It provides a list of the different activities and the process editor where you can use these activities to build a process.

6.1 Concepts

It is important to have an understanding of the concepts and terminology used within WebSphere Process Choreographer before going into the details of creating business processes. This chapter introduces the key concepts used when talking about business processes.

The latest versions of WebSphere Business Integration Server Foundation and WebSphere Studio Application Developer Integration Edition include several significant features not provided in previous versions, so even if you are familiar with the V5.0 products, you should read this chapter to familiarize yourself with these changes.

The InfoCenter contains a more detailed overview of these and other concepts. To access this, select **Help** → **Help Contents** from the WebSphere Studio Application Developer Integration Edition menus, then navigate to **WebSphere Studio** → **Developing** → **Processes** → **The Process editor (BPEL)** in the help file.

6.1.1 Process languages

WebSphere Business Integration Server Foundation V5.1 introduces Business Process Execution Language for Web Services (BPEL4WS, which is often abbreviated as BPEL) as a replacement process language to the FDML-based flows. BPEL4WS provides a more flexible standards-based approach to defining and executing business processes.

FDML flows can still be developed and executed although their use is deprecated. WebSphere Studio Application Developer Integration Edition provides a wizard to migrate FDML flows to BPEL4WS. Although BPEL4WS processes and FDML flows can both be used in a single application, it is not possible to develop and store both types of process in the same WebSphere Studio Application Developer Integration Edition Service project.

The implementation of the business process engine includes some added capabilities which extend the basic BPEL4WS specification. These add support for staff-related activities and embedding of Java code as Java snippets which increase the power and productivity of the tool.

BPEL4WS processes will be used exclusively throughout the rest of this book. For details about FDML, refer to the InfoCenter or to the IBM Redbook *Exploring WebSphere Studio Application Developer Integration Edition V5*, SG24-6200.

6.1.2 Non-interruptible and interruptible processes

Processes may be *interruptible* or *non-interruptible*. As the names suggest, interruptible processes can be suspended and resumed, whereas non-interruptible processes stay active from the time they start to the time they complete.

Note: In the product documentation:

- Interruptible processes are sometimes called *long-running processes* or *macroflows*. The macroflow name is no longer used to describe interruptible processes.
- Non-interruptible processes are sometimes called *short-running processes* or *microflows*. The microflow name is no longer used to describe non-interruptible processes.

The key differences between the two modes of operation are:

- State and status persistence

  Interruptible processes persist their state and status to disk between activities. This contrasts with non-interruptible processes which manage state and status in memory.

- Transactionality

  Interruptible processes may contain multiple transactions. In contrast, non-interruptible processes execute within a single transaction.

If a process is likely to execute for an extended period of time, it is recommended that the process be interruptible, therefore persisting state and status. This enables system resources to be released for other processes active in the process container. It also allows the BPE container to be shut down and restarted without losing the state and status of the process.

However, the overhead associated with storing the state to disk means that interruptible processes are not so well suited to high throughput applications. In this case, non-interruptible processes should be used.

Tip: Here are a few other general comments about interruptible and non-interruptible processes:

- Asynchronous activity

  A process must be interruptible if it contains an asynchronous activity such as pick or staff. Otherwise, it will result in an error being raised.

- Configuring a process to be interruptible

  Processes can be configured to be interruptible by selecting the **Process is long-running** checkbox on the Server tab of the process details area.

- Binding of interruptible processes

  Interruptible processes cannot bind to a synchronous port type such as EJB, but must be bound to asynchronous port types such as JMS.

- Threading behavior

  The activities of an interruptible process are queued internally before they are executed by the process container. Hence, they may be executed in different threads. In contrast, non-interruptible processes are always executed in a single thread within the process container. This means that even if a non-interruptible process contains parallel activities, these will be executed sequentially.

  There are potentially more serious implications for processes which invoke other processes directly via a BPEL Partner Link. If the invoked process is also non-interruptible, it will run in the same thread as the parent process and any time-out set on the parent process Invoke activity will not trigger even though it takes longer than the time-out setting.

6.1.3 Transactional behavior

Transactional behavior depends on the interruptibility of the process. Non-interruptible processes execute within a single transaction. In contrast, each activity within an interruptible process can have its transactional behavior set to one of four states:

- Commit before

  Pending transactions are committed before the activity starts.

- Commit after

  The activity continues the existing transaction, committing the transaction once it has completed.

- Participates

  The activity continues the transaction.

- Requires own

  The activity commits pending transactions before it starts and commits its own transaction once it completes.

In either case, if a failure is detected, the current uncommitted transaction is rolled back and any pending compensation activities are applied. The result of this is that the system is returned to a balanced state.

Compensation is a separate, WDSL-described service which balances the action if an Invoke activity is used. A compensation service is applied to an Invoke activity on the compensation tab on the process details area.

6.1.4 Sequences and flows

BPEL4WS provides *flow* and *sequence* structures which allow nesting. Processes can include combinations of sequences and flows contained within one another. The features of each structure are outlined below.

- Flow

  Activities within a flow execute concurrently. The addition of *links* and their associated conditions provides additional control over order of execution. The end of a Flow activity is a synchronization point: all activities must complete before the process continues.

- Sequence

  Activities in a sequence execute sequentially. The order of execution is defined by the positioning within the block.

Either a sequence or a flow can be used as a starting point for development. The process has exactly the same characteristics and capabilities. The decision is thus largely a matter of personal preference.

6.1.5 Parts of a business process

This section introduces the major building blocks of a process. Further details on these concepts can be found in the BPEL4WS specification.

Partner links

The parties that will interact with the business process are defined as *partner links*. Partners may be one of a number of types: WSDL service, BPEL, EJB Java Class or Transform Service. Transform services are defined in WSDL.

For WSDL-based partner links a *partner link type* enables the *role* of the partner and/or the process to be defined. There may be one or two roles defined within a

partner link type, depending on the requirements of the process. Roles are assigned to participants in the partner link definition. Each role is associated with a port type derived from a WSDL file. This defines the operations that the role supports.

Note: In the following discussion, the terms *synchronous* and *asynchronous* refer to the style of interface in the context of the process, and whether or not a response is expected to a request before the process continues. A synchronous interface could be provided by an asynchronous transport such as JMS.

- One role is used for synchronous style interfaces.

  Some examples of use of partner link types with one role are given below:

  - If a process invokes a synchronous WSDL service, a partner link type with one role is used. In this case, the partner role is defined and the process role is blank.
  - A synchronous interface offered by a process also uses a partner link type with one role. In this case, the process role is defined and the partner role is blank.

- Two roles are used for asynchronous style interfaces.

  The overall flow of the interaction is outlined below:

  a. A process invokes a service asynchronously and continues with processing.

  b. At some later time, the invoked service responds via an interface that the process provides.

  Both request and response are attached to a single partner link, the request using the partner role and the response using the process role.

Tip: This is a rule of thumb for working out which roles need to be defined within a partner link. If the process

- *provides* a service, the partner link must contain a *process role*
- *consumes* a service, the partner link must contain a *partner role*

These rules are not mutually exclusive: a process may both provide and consume a service via a single partner link.

Variables

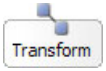
WebSphere Process Choreographer uses the concept of *variables* to pass data between steps in the process. The type of each variable is described by a

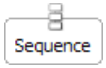
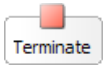
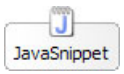
message comprising one or more message parts. The message is defined in WDSL.

Activities

An overview of the available BPEL4WS process activities is given in Table 6-1. For more detailed descriptions and examples, refer to the InfoCenter.

Table 6-1 WebSphere Process Choreographer activities

Activity	Description
	The Invoke activity performs an operation. The operation is defined by a partner link and may be synchronous or asynchronous.
	The Receive activity waits for an external input to the process before continuing. The operation supported by the Receive activity is defined by a partner link.
	The Reply activity sends a message to the partner defined by a partner link. This is typically used in processes which need to return a message to the partner which instigated the process.
	The Pick activity waits for an incoming message and selects a path appropriate to the first message received. A time-based path can be configured to manage situations where no message is received. A partner link is associated with each message path.
	The Staff activity delegates a task within the process to a human. The user interface in this case is either a custom application based on the process choreographer API or the Web client provided that comes with WebSphere Business Integration Server Foundation.
	The Transformer activity maps the contents of one or more message types to the contents of another.
	The Assign activity copies information from one part of the process to another.
	The Switch activity evaluates the conditions on a series of control paths and follows the first one which matches.
	The While activity repeats the activities which it contains as long as a condition is met.

Activity	Description
	The wait activity stops the process until a point in time has occurred or a time interval has elapsed.
	The sequence activity defines a serial control path within a process. See 6.1.4, “Sequences and flows” on page 125 for further details.
	The flow activity defines a potentially parallel control path within a process. See 6.1.4, “Sequences and flows” on page 125 for further details.
	The terminate activity stops the process immediately without performing any compensation or fault handling. The behavior of this activity depends on the location within the process.
	The throw activity signals that an error has occurred. This is typically handled by a Fault Handler element associated with a higher level of process structure.
	The empty element does nothing. It can be used as a placeholder during process design, and then changed to the appropriate activity when the process is implemented.
	Java code can be embedded into the process using the JavaSnippet activity. While it is possible to embed business logic into this type of activity it is not advisable, as it removes the clarity of the process modeling. Snippets are designed to perform lightweight utility activities such as data mapping.

Tip: The activity type can be changed by right-clicking the activity in the process editor and selecting the **Change Type** menu item. Use this facility with caution; changing the type of a structured activity such as sequence or flow will delete the contents of that block.

Correlation sets

Correlation sets identify the participants in a process by reference to some unique information contained in the messages that are passed. This allows the business process container to decide whether an incoming message should launch a new process or use an existing instance. Correlation sets can be associated with service activities such as receive, reply and invoke as well as with pick activities.

6.2 Development tooling support

WebSphere Studio Application Developer Integration Edition provides support for creating processes based on BPEL4WS. This section describes the editor that is provided.

6.2.1 BPEL Editor

The BPEL Editor (also referred to as the BPEL process editor) is a graphical programming environment that is used to visually create and manipulate business processes. Figure 6-1shows the BPEL Editor.

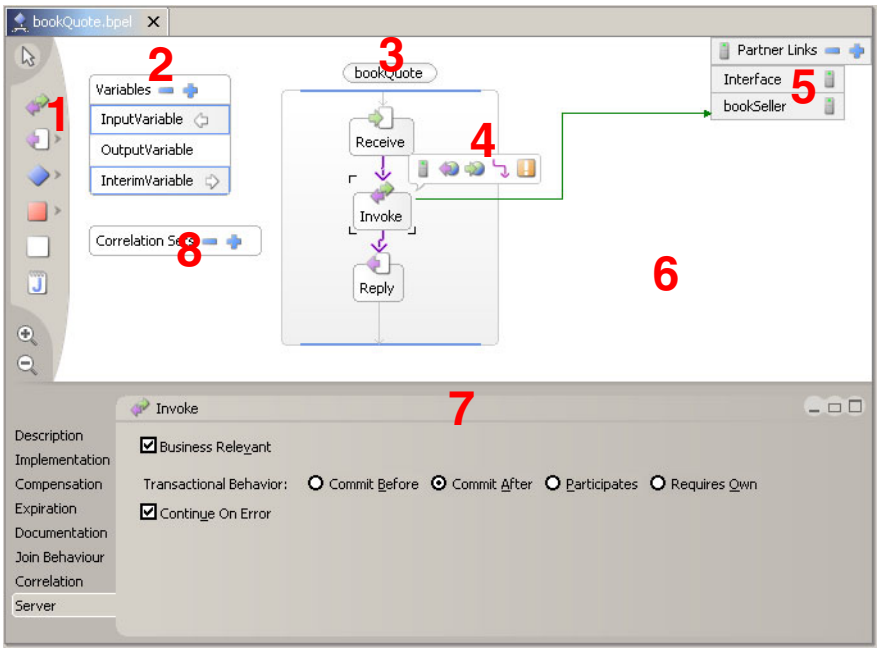


Figure 6-1 The BPEL Editor

Processes are constructed in the *process area* (3) of the *canvas* (6) by dragging activities from the *palette* (1). Definitions of variables, partner links and correlation sets are held in separate *areas* (2, 5, 8) on the canvas.

Selecting any activity brings up the *action bar* (4) which contains a series of icons related to the activity, including adding Fault Handlers. The *details area* (7) below the canvas provides the means for configuring the currently selected activity.

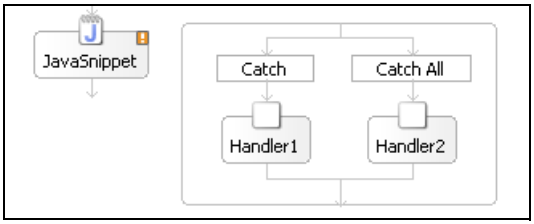


Figure 6-2 A Fault Handler

A small exclamation point in the top right corner of any activity means that a *Fault Handler* is defined for that activity. Figure 6-2 shows a Fault Handler block in the BPEL Editor. The Fault Handler can be opened and closed either by double-clicking the exclamation point or by right-clicking the activity and selecting the **Show Fault Handler** menu item. Catch elements can then be added to the Fault Handler.

Tip: Always close the Fault Handler block when you have finished editing it. There appears to be an intermittent bug in the BPEL Editor which causes the process canvas to be cleared if a Fault Handler block is left open while navigating around the canvas.

6.2.2 The Web client

The Web client provides access to a running process. Although it can be used as a general purpose user client, it is quite likely that a specialized application will be created for end users of the system. The Web client is useful during process development and testing to instantiate a process with a certain set of input parameters.

The Web client is described in more detail in a number of articles available at:

`http://www.ibm.com/developerworks/websphere/zones/was/wpc.html`

6.3 Runtime environment

This section contains an overview of the BPE container. Further details can be found either in the WebSphere Business Integration Server Foundation InfoCenter or in the white paper *WebSphere Application Server Enterprise Process Choreographer: Concepts and Architecture*, which is available from:

`http://www.ibm.com/developerworks/websphere/zones/was/wpc.html`

6.3.1 Business Process Execution container architecture

WebSphere Business Integration Server Foundation provides the runtime environment for WebSphere Process Choreographer. This is the Business Process Execution (BPE) container. Figure 6-3 shows the components of the container. It is implemented as a J2EE application that uses the underlying WebSphere Application Server runtime services and resources.

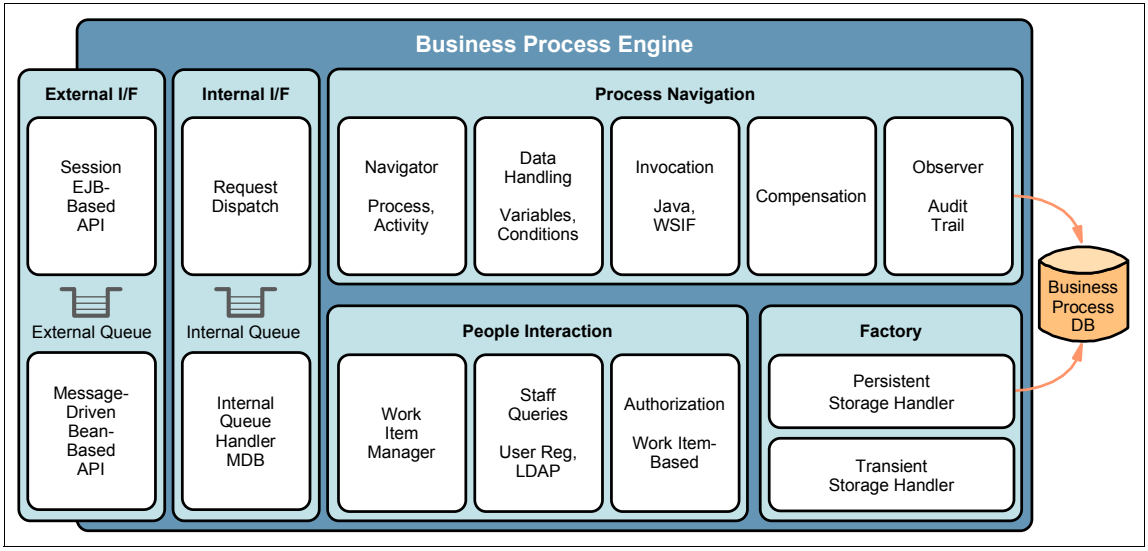


Figure 6-3 Business process execution container architecture

The key components of the BPE container are:

- Process navigation
- Factory
- People interaction
- Internal interface
- External interface

Each of these will be discussed briefly in the remainder of this section.

Process navigation

The process navigation component consists of the navigator and some plug-in components.

Navigator

The navigator is the heart of the BPE container. It manages the state of all process instances and the activities that they contain.

The life of a process instance begins with a start request. This creates the instance based on a process template and puts it into a *running* state. When all its contained activities have reached an end state, the process instance is marked *finished*. The instance remains in this state until it is deleted, either implicitly or via an explicit API call.

The process instance might encounter a fault that was not processed as part of the process logic. In this case, it waits for the completion of the active activities before putting the process into its *failed* state. Compensation is then invoked if it was defined for the process.

A process instance can also be terminated by a process administrator. In this case, after completion of the active activities, the process instance is put into its *terminated* state.

Plug-ins for process navigation

The core capabilities of the navigator are extended using plug-ins. These provide future flexibility and extensibility for the product.

Plug-ins are provided for:

- Invocation of activity implementations

  There are currently two plug-ins that support invocation: the invocation of external processes via WSIF and the invocation of Java snippets.

- The handling of data in the process, such as evaluating conditions

  The process engine has a plug-in that understands conditions written in Java against WSDL messages.

- The logging of events in an audit trail

  The process engine has a plug-in that writes data to the audit trail table of the process engine's database.

Factory

The factory component is responsible for state data that the process engine deals with. It allows data to be stored in one of the following forms:

- Transiently in memory

  This is used to support the efficient execution of non-interruptible processes

- Persistently in a database

  This is used to provide durability to interruptible processes. Many popular databases are supported, including DB2 Enterprise Server Edition.

Human interaction

Note: WebSphere Process Choreographer supports business processes with people interaction only when WebSphere Application Server security is enabled. This is because the user needs to be authenticated to determine the appropriate work items.

The main components involved in interaction with people are:

- **Web client or other client**

  It is possible to interact with the process instances via the Web client described in 6.2.2, “The Web client” on page 130. This can be tailored to the requirements of the business application.

  Alternatively, the WebSphere Process Choreographer API can be used to create a custom client.

- **Work item manager**

  Work items are created when the BPE container encounters a Staff activity. The work item manager component is responsible for handling work items. This entails:

  – Creating and deleting work items

  – Resolving queries from process participants

  – Coordinating staff queries

  – Authorizing activity on process instances

    This ensures that participants only gain access to process instances for which they have a valid work item.

  The work item manager has a number of performance-related features, notably an internal cache for resolved staff queries.

- **Staff support service, staff resolution plug-ins and staff repositories**

  The staff support service manages staff resolution requests on behalf of the work item manager. It actually delegates execution to the staff resolution plug-ins. These plug-ins work with the staff repositories to fulfil requests. There are operating system repositories, user registries or LDAP registries.

  For more details about staff resolution, see the document *WebSphere Application Server Enterprise Process Choreographer: Staff Resolution Architecture* available at:

  `http://www.ibm.com/developerworks/websphere/zones/was/wpc.html`

Internal interface

Interruptible processes use a JMS queue between activities to provide durability. In most production environments, this should be based on a robust external JMS provider, such as WebSphere MQ.

External interface

The interface to the container is via a façade. This is provided both asynchronously as a Message-Driven bean and synchronously as a session EJB.

7

Chapter 7

Process choreographer: developing a simple process

This chapter shows how to implement a simple business process using the BPEL Editor of WebSphere Studio Application Developer Integration Edition.

From a technical point of view, implementing a business process consists of the steps to develop and deploy (and perhaps debug) this business process. These steps will be shown in this chapter.

7.1 Sample scenario

A fictional travel agency called NiceJourney will provide its travel arrangement service over the Internet as a Web Service to its customers. To do this, the NiceJourney travel agency has to communicate with an external partner, in order to reserve flights for its customers. Therefore, the external partner is a Flight Booking System which is also exposed as a Web Service. The customers communicate with the NiceJourney Web Service via a Web Service client.

Hence, this scenario uses both the Business to Customer (B2C) and Business to Business (B2B) model.

Currently, when a customer requests a travel arrangement, the NiceJourney Web service requires the following information:

- Customer details
- Travel details
- Booking options
  - Flight
  - Car
  - Hotel
- Price limit
  - Complete journey
- Payment details

To simplify the example in this chapter, not all of the information shown above will be used by the NiceJourney Web Service and the Flight Booking System Web Service. However, in the complex scenario in Chapter 8, “Process choreographer: developing a complex process” on page 203, all the provided information will be utilized and the business logic will be more meaningful and realistic.

The outline of the NiceJourney Web Service is as follows:

1. Receive customer information
2. Send travel information to the Flight Booking System
3. Receive a reservation ID from the Flight Booking System
4. Reply to the customer with the reservation ID

For the purpose of illustrating the basic functionality of WebSphere Process Choreographer, no erroneous paths have been included in this scenario. For a more complex business scenario, refer to Chapter 8, “Process choreographer: developing a complex process” on page 203.

7.1.1 Interactions between involved partners

- The customer might be any Web Service client that can communicate via WSDL. Refer to Chapter 9, “Process choreographer: clients” on page 287 to learn how to implement such a client. In this chapter, we will use the Business Process Web Client, which comes with WebSphere Studio Application Developer Integration Edition.
- The NiceJourney Web Service will be implemented as a BPEL process; for further information on the BPEL process, refer to 6.1.1, “Process languages” on page 122.

  This Web Service is offered by the operation travelArrangement on a WSDL port Type called travelArrangementType. The operation is of type request/response, hence it is used to receive customer details and travel details and in response to this, it will reply with the reservation ID.
- The Flight Booking System will be a Java class exposed as a Web Service.

  To make a flight reservation, the operation getReservation is offered on WSDL port type FlightBookingSystem. This operation is of type request/response so it is used to receive travel details and in response to this, it will reply with a reservation ID.

  The following figure shows the scenario for the simple process. The customer interacts with the NiceJourney Web Service and the NiceJourney Web Service itself interacts with a third party, the Flight Booking System, to make an appropriate flight reservation. The reservation ID generated by the Flight Booking System is returned to the NiceJourney Web Service. To simplify the scenario, this reservation ID is also returned to the customer by the NiceJourney Web Service without any changes.

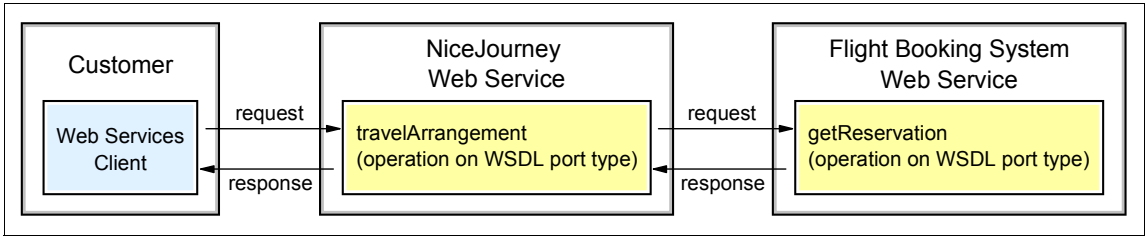


Figure 7-1 The simple process scenario

7.1.2 Input messages and output messages

This section will discuss the input and output messages required for the NiceJourney and Flight Booking System Web Services.

Interface to NiceJourney Web Service

The input and output messages required for the NiceJourney Web Service are defined in NiceJourneyPublicInterface.wsdl. The WSDL files containing the binding and service information for the NiceJourney Web Service will be generated by WebSphere Studio Application Developer Integration Edition.

- **Operation:** travelArrangement

  Receives the message *travelAgencyIn* from the customer, which consists of:

  - `[complextype: customer]`
    - firstName xsd:string
    - lastName xsd:string
    - address xsd:string
    - city xsd:string
    - zipcode xsd:nonNegativeInteger
    - state xsd:string
  - `[complexType:travel]`
    - cityFrom xsd:string
    - cityTo xsd:string
    - dateDeparture xsd:dateTime
    - dateReturn xsd:dateTime

    Replies with the message *travelAgencyOut* to the customer:

    - reservationID xsd:long

Interface to the Flight Booking System

The input and output messages required for the Flight Booking System are defined in FlightBookingSystem.wsdl.

- **Operation:** getReservation

  This needs to receive the message getReservationRequest that contains these simple types:

  - cityFrom xsd:string
  - cityTo xsd:string
  - dateDeparture xsd:dateTime
  - dateReturn xsd:dateTime

  It replies with the message getReservationResponse:

  - result xsd:long

Note: The partial result of the message getReservationResponse shown above is the reservation ID generated by the Flight Booking System. Since the WSDL files were generated from a Java class, this name was generated.

7.2 Activities in the sample

The activities defined in BPEL4WS are divided into *basic activities* and *structured activities*.

The *basic activities* within a business process are used to define tasks; the *structured activities* are used to manage the complexity of the business process. Hence, a *basic activity* can be compared to be a step while a *structured activity* is a programming construct in the business process.

In addition to the BPEL4WS activities, there are additional activities defined in the BPEL Extensions to enhance WebSphere Process Choreographer.

7.2.1 Receive activity

The Receive activity is a *basic activity* in the definition of BPEL4WS.

It has a blocking runtime behavior, waiting for an external input message to the process. This input message is covered by an operation of a WSDL port type that is associated with a certain role of a partner link.

The following figure shows the implementation view for the Receive activity. This view is located in the details area of the BPEL Editor.

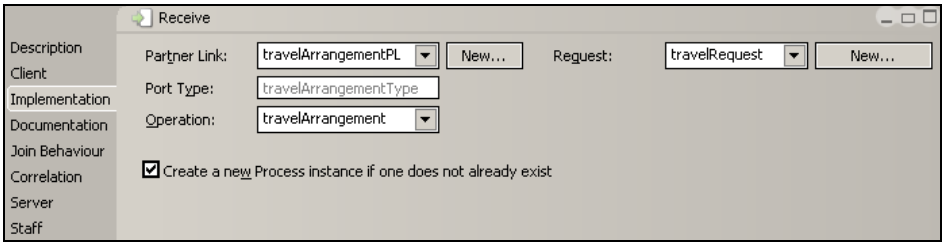


Figure 7-2 Implementation view of the Receive activity

Once the message has arrived, the process will continue or a new process instance will be created, depending on the setting of the check box (see Figure 7-2). The arrived message is mapped to a variable to make it accessible within the complete process.

Select the Receive activity to see the related partner link and the highlighted variable to which the input message is mapped (there will be a bold border around the variable name).

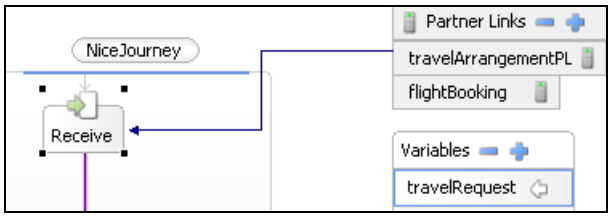


Figure 7-3 Receive activity associated with Partner link and mapped to variable

If the input message is sent to a certain process instance, a correlation has to be defined on the Receive activity, identifying this process instance uniquely. Refer to 6.1.5, “Parts of a business process” on page 125 to learn more about the correlation set.

The Receive activity can be used to implement a Synchronous Interface (see also “Synchronous interface” on page 145). For this, a matching Reply activity (see also 7.2.2, “Reply activity” on page 141) has to be placed in the process, associated with the same partner link as the Receive activity.

The following figure shows how this can be implemented in the BPEL Editor; the definition of the WSDL port type is shown below.

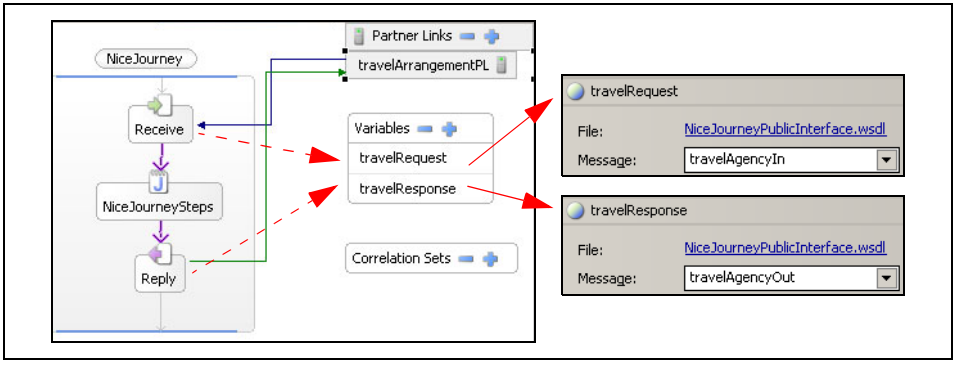


Figure 7-4 implementation of a synchronous interface in the BPEL Editor

Note: The arrows in Figure 7-4 show the relation between the components. The input message of the Receive activity is related to the travelRequest variable. The output message of the Reply activity is related to the travelResponse variable. Both variables are related and defined to map to the travelAgencyIn and travelAgencyOut messages from the NiceJourneyPublicInterface.wsdl interface definition.

The red arrows are not part of the BPEL Editor.

The WSDL port type that is associated with this shared partner link, travelArrangementPL, has to define the synchronous request/response operation, as shown in Example 7-1.

Example 7-1 WSDL port type to implement a synchronous interface, defined in NiceJourneyPublicInterface.wsdl

```
<portType name="travelArrangementType">
   <operation name="travelArrangement">
      <input message="tns:travelAgencyIn"   name="travelArrangementIn"/>
      <output message="tns:travelAgencyOut" name="travelArrangementOut"/>
   </operation>
</portType>
```

An Asynchronous Interface can be implemented using the Receive activity without a matching Reply activity (refer to “Asynchronous interface” on page 146). For this, the operation of the WSDL port type has to consist of a one-way operation only.

7.2.2 Reply activity

The Reply activity is a *basic activity* in the definition of BPEL4WS.

It is always used in response to an input message that previously arrived in the process. The output message to be sent is covered by an operation of a WSDL port type that is associated with a certain partner link role.

Figure 7-5 on page 142 shows the implementation view for the Reply activity for a normal reply type. This view is located in the details area of the BPEL Editor.

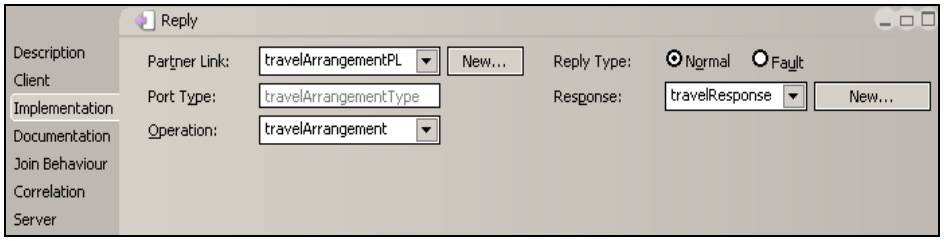


Figure 7-5 Implementation view of the Reply activity for a normal reply type

The output message of the Reply activity is gained from a variable of the process.

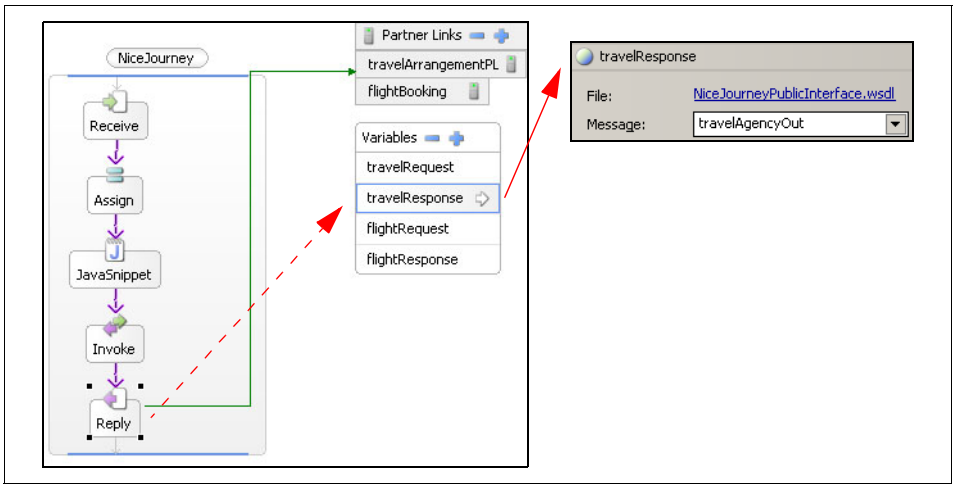


Figure 7-6 Reply activity associated with Partner link and mapped to variable

The output variable has to cover an output message that is defined in the same WSDL port type as the input message for which this is the response; it is shown in the following example.

Example 7-2 WSDL port type with output message and with two fault messages f

```
<portType name="travelArrangementType">
   <operation name="travelArrangement">
      <input message="tns:travelAgencyIn"   name="travelArrangementIn"/>
      <output message="tns:travelAgencyOut" name="travelArrangementOut"/>
      <fault message="tns:travelAgencyError"
                                  name="travelArrangementErrorFlight"/>
      <fault message="tns:travelAgencyError"
                                  name="travelArrangementErrorCar"/>
   </operation>
</portType>
```

The example above also shows the definition of two fault messages which can be used for a fault reply type. The Reply activity can return either an output message, as shown in Figure 7-5 on page 142, or a fault message to its associated WSDL port type. Since an operation of a WSDL port type might have several fault messages defined (see above), one of these has to be selected as the fault target of the Reply activity.

The following figure shows the implementation view for the Reply activity for a fault reply type.

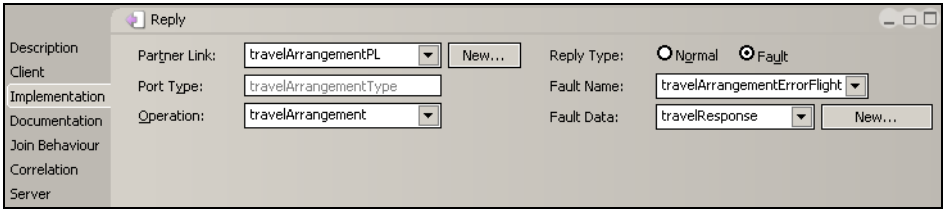


Figure 7-7 implementation view of the Reply activity for a fault reply type

In contrast to the Receive activity, the Reply activity cannot be used to implement an Asynchronous Interface. It is only available as the response to a synchronous request, as shown in Figure 7-4 on page 140. Therefore, each Reply activity must have a corresponding Receive or Pick activity.

7.2.3 Invoke activity

The Invoke activity is a *basic activity* in the definition of BPEL4WS.

The general purpose of this activity is to invoke an operation offered by a partner, by exchanging messages. Technically, this is done by invoking an operation on a WSDL port type that is associated with a certain role of a partner link. Depending

on the definition of this WSDL port type, the Invoke activity can by used to implement a request/response or a one-way operation.

The following figure shows the implementation view for the Invoke activity for a request/response operation. This view is located in the details area of the BPEL Editor.

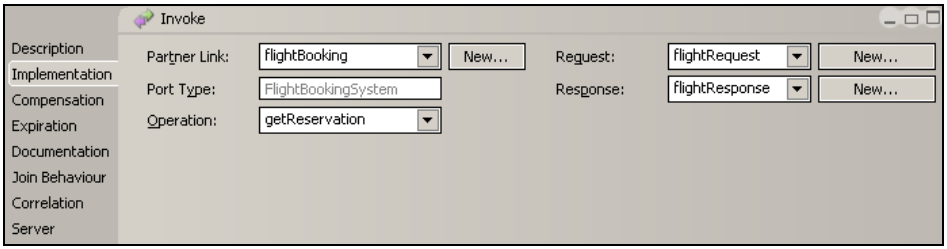


Figure 7-8 Implementation view of the Invoke activity for a request/response operation

The messages, sent to or received from the referenced WSDL port type, must be defined in the same namespace as the WSDL port type itself. It is important to emphasize that it will not be enough for the data type(s) of the part(s) of exchanged messages to be the same.

The following figure shows the Invoke activity and its associated partner link; the variables associated with the Invoke activity are also shown. The output message to be sent to the partner is gained from the variable named flightRequest; the input message received from the referenced partner will be mapped to the variable named flightResponse.

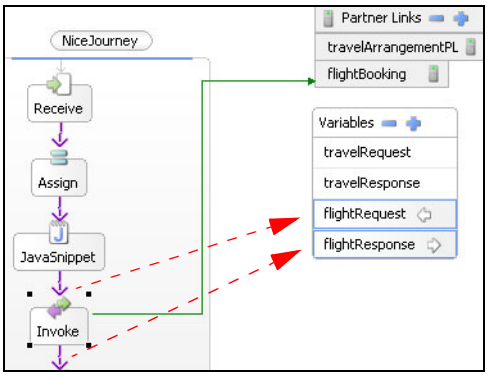


Figure 7-9 Invoke activity associated with Partner link and mapped to variables

By changing the type of the partner link (right-click the partner link then select **Change Type**) that is wired with the Invoke activity, it is possible to invoke:

- WSDL service
- BPEL process
- EJB
- Java class
- Transform service

According to the chosen type, the generation of appropriate WSDL files is more or less implicitly done by WebSphere Studio Application Developer Integration Edition. Finally, a partner will always be called over an operation on a WSDL port type, regardless of the previous chosen type.

Synchronous interface

The Invoke activity can be used to implement a synchronous interface. The activity has to be associated with a WSDL port type containing a request/response operation; see Port Type B-PT in the following figure.

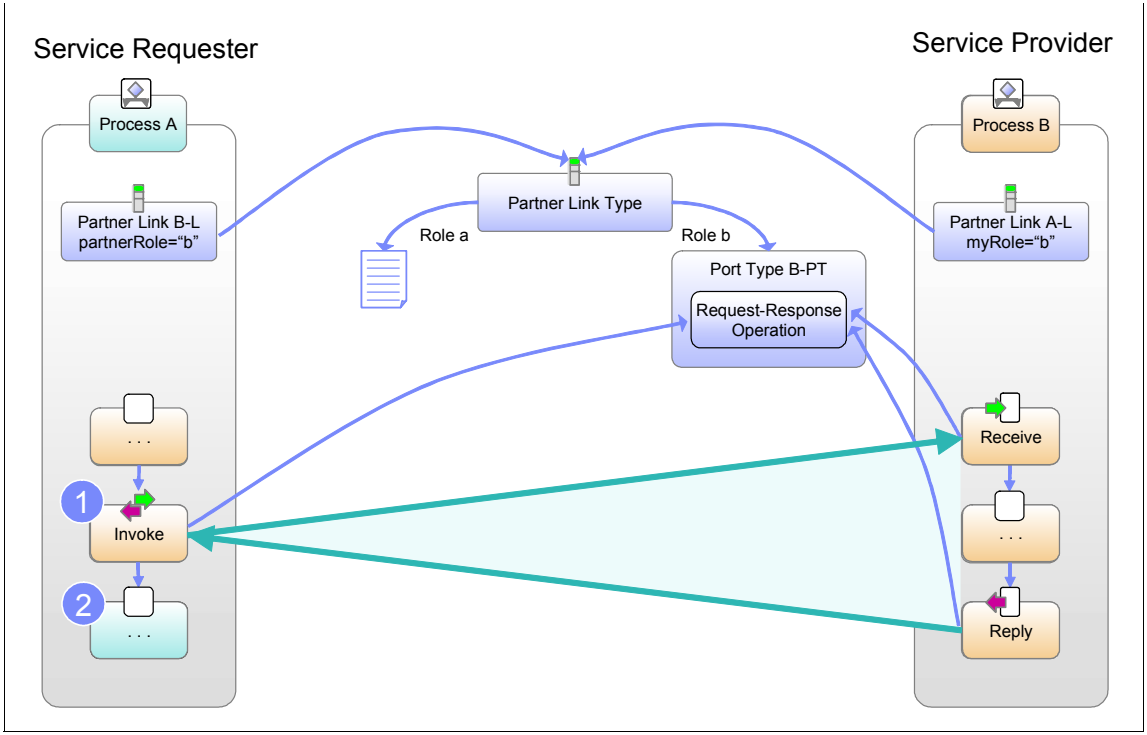


Figure 7-10 Synchronous Interface

The use case to invoke a partner (service provider) in a synchronous manner the service requester can not go on with the process execution without the result delivered from the service provider.

#1 inFigure 7-10: Process A invokes process B and waits for the result. Once the result from process B has arrived, process A can go on with its execution in #2.

In the implementation, the partner link type has to consist of one role only. This role is associated with a WSDL port type that covers a request/response operation in the namespace of the service provider.

The service requester implements the partner link type in its partner link to refer to the service provider. Since the service requester expects a service from its referenced partner, its partner link defines only a role for the partner; see Partner Link B-L in Figure 7-10.

The service provider has to implement the same partner link type as the service requester. The fact that the service provider is the invoker in this scenario is reflected in the implementation of the partner link type; the partner link defines that process B owns the role “b”, see Partner Link A-L in Figure 7-10.

Asynchronous interface

A common way to invoke a partner without waiting for the response is to use a Invoke activity associated with a WSDL port type that consists of an one way-operation only.

In an asynchronous interface, there are two partners and each of them are performing an one way-operation to its partner where both partners implement the same partner link type. See Figure 7-11 on page 147 for more details.

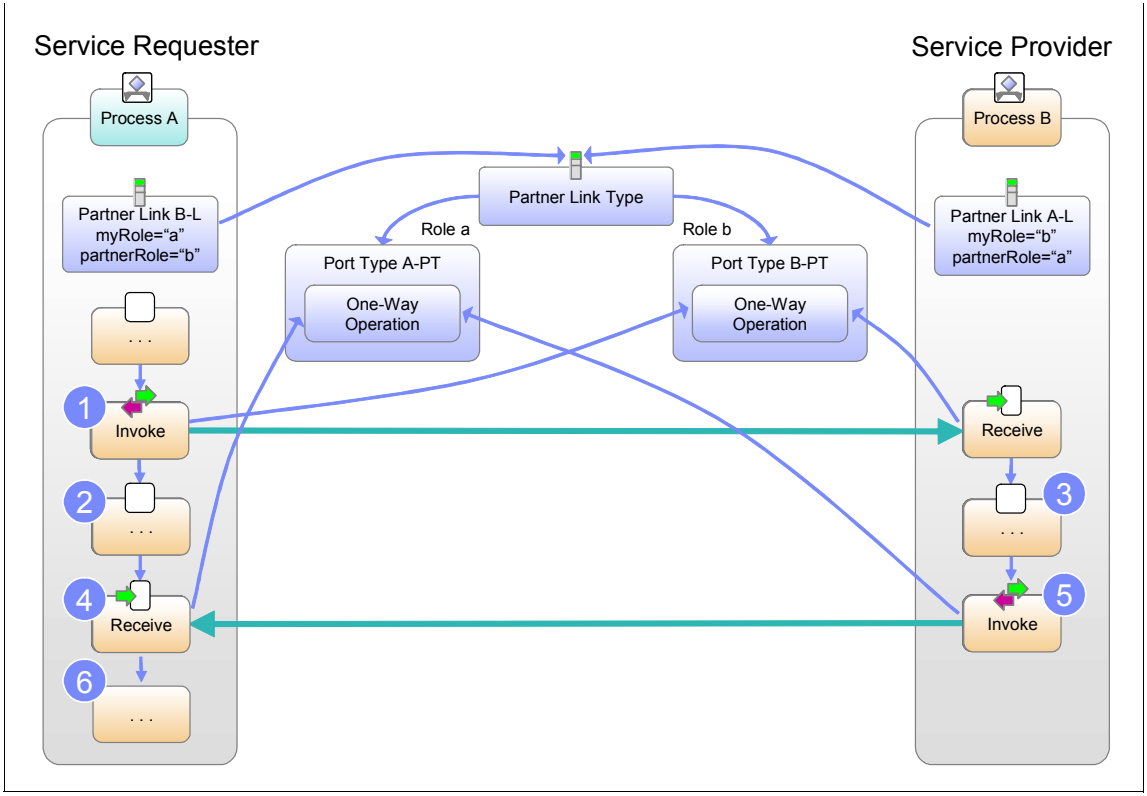


Figure 7-11 Asynchronous interface

The use case for an asynchronous interface is if a service requester invokes a partner (service provider) to get a result for later usage and the service requester can go on with the process execution (do several other steps), before the result requested from the partner becomes necessary.

#1 in Figure 7-11: Process A invokes Process B and goes on with the process execution, going through further steps for which the requested result is not necessary. Process B gets this request and processes it. At this point in time, both processes may be running in parallel; see #2 and #3. In a certain execution state of Process A, the previous requested result of the service provider will be necessary. To get this result, Process A will do a blocking wait using a Receive activity, (#4). If this result is determined by Process B, it will be delivered to Process A, using an Invoke activity (#5). Once the result has arrived to Process A, it can go on with the process execution, using the result delivered from Process B.

To implement an asynchronous interface, the partner link type has to define two roles. Each of it associated with a WSDL port type which consists of an one-way operation in the namespace of that partner to which the message should be delivered. See figure above, WSDL port type B-PT has to be defined in the namespace of the service provider, WSDL port type A-PT has to be defined in the namespace of the service requester.

The service requester implements the partner link type in its partner link to refer to the service provider, see partnerRole=”b” in Partner Link B-L. Due to the definition of myRole=”a” in the same partner link, the service requester knows the port type to which the service provider will respond.

The service provider has to implement the partner link type in its partner link in the same manner as the service requester, but with swapped roles. See Partner Link A-L; the service provider has the definition of myRole=”b” which means the service requester will communicate with it over the WSDL port type associated with role “b”. Due to the definition of the partnerRole=”a”, the service provider knows the WSDL port type to send the response to the service requester.

7.2.4 Assign activity

The Assign activity is a *basic activity* in the definition of BPEL4WS.

A common task in business processes is to copy the content of variables. Using the Assign activity, the content of type-compatible variables (see Figure 7-12) can be copied from a source to a destination. Source and destination are named as `From` and `To` in the following figure. It shows the implementation view of the Assign activity. This view is located in the details area of the BPEL Editor.

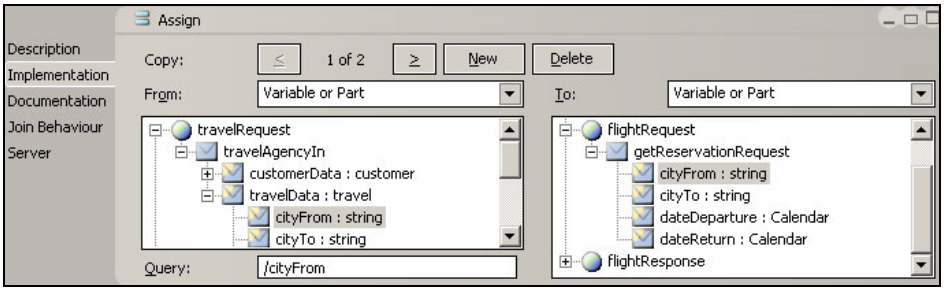


Figure 7-12 Implementation view of Assign activity

For the *From* and *To*, shown in the figure above, the following kinds of assignments are available in the drop-down list:

- Variable or Part
  - Copy the complete content of the From-variable to the To-variable.

    Both variables must cover the same message in the same namespace.
  - Copy a part of the message covered by the From-variable to a part of the message covered by the To-variable.

    The parts must be type-compatible according to their XML Schema definition, also known as xsd.
- Property of a Variable

  Copy the message type of the From-variable to the To-variable, not the content.

  This is not completely supported by WebSphere Studio Application Developer Integration Edition V5.1.
- Partner Link Reference

  Copy an endpoint references to and from partner links.

The query field, as shown in Figure 7-12, can be used to specify how to find the data in a selected part with the usage of an XPath query. If an element of a complex type is selected in the From pane, the query field suggests an XPath query to use.

For the From, shown in Figure 7-12, the following additional kind of assignment is available in the drop-down list:

- Fixed Value

  Define a fix value, valid xsd type, in the pane below *From* and assign it to a type-compatible part of a message shown in the pane below *To*.

The following figure shows an example of how to assign a fixed value to a variable part.

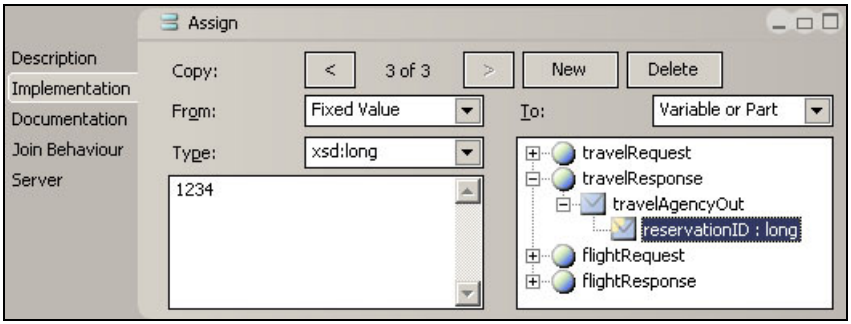


Figure 7-13 Assign a fixed value to a part

Within one Assign activity, several assigns can be implemented; by clicking the **New** button, as shown in Figure 7-13, an additional assign can be added. When you have multiple assign definitions, use the **<** and **>** buttons to navigate between them. Hence, the implementation of a multiple assign can be visualized as follows.

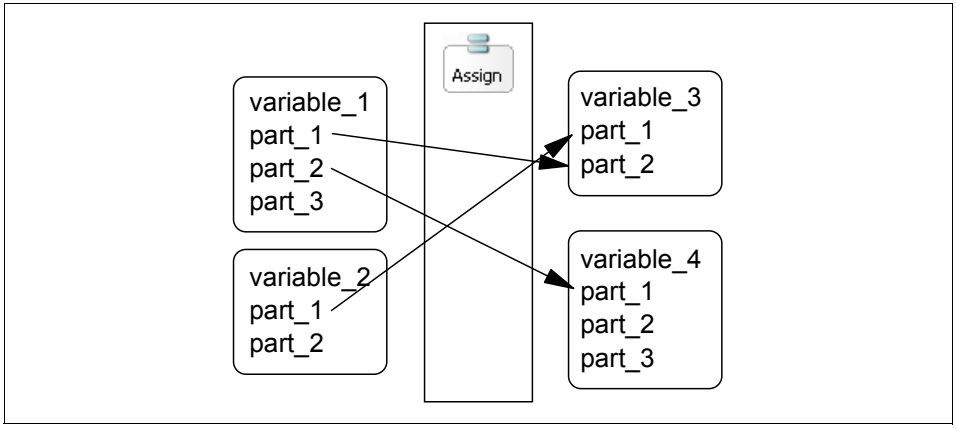


Figure 7-14 Visualization of a multiple assigns within one Assign activity

Assigning values to variables can also be done within Java snippets, but this is not recommended in general. To assign values to complex types, Java snippets might be necessary if those values have to be converted beforehand. Refer to section 7.2.7, “Developing a new process” on page 156 for more details.

7.2.5 Java snippet

The Java snippet is not a part of the BPEL4WS definition; it is a basic activity in the definition of BPEL Extensions to enhance WebSphere Process Choreographer.

To invoke an inline snippet of Java code, this activity can be added to a business process. It may contain any valid Java code as of Java V1.4.

The following figure shows the implementation view of a Java snippet. It is basically an in-line Java code in the process.

Figure 7-15 Implementation view a Java snippet

Every Java snippet is added as a method to a Java class which is generated for the BPEL business process. This class also provides access to the variables defined in the BPEL business process. For each variable, several getter and setter methods are available, each of which is a wrapper for the message of this variable. For each message, a Java class is generated by WebSphere Studio Application Developer Integration Edition. This will be explained in detail next.

Example 7-3 shows the definition of two messages:

- travelAgenyIn

  A complex element of type tns:travel is defined in the types section of the WSDL definition.

- travelAgencyOut

  A simple type of xsd:long is primitive; no further definition is necessary.

Example 7-3 WSDL messages, one with a simple part, one with a complex part

```
<types>
   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified"
      argetNamespace="http://nicejourney.com/NiceJourneyPublicInterface">
      <xsd:element name="travel">
         <xsd:complexType>
            <xsd:sequence>
```

```
                <xsd:element name="cityFrom" type="xsd:string"/>
                <xsd:element name="cityTo" type="xsd:string"/>
             </xsd:sequence>
          </xsd:complexType>
       </xsd:element>
   </xsd:schema>
</types>
<message name="travelAgencyIn">
   <part name="travelData"   element="tns:travel" />
</message>
<message name="travelAgencyOut">
   <part name="reservationID" type="xsd:long"/>
</message>
```

In the BPEL Editor, two variables are defined; each covers a message, as shown in Figure 7-16.

- travelRequest - covers message travelAgencyIn
- travelResponse - covers message travelAgencyOut

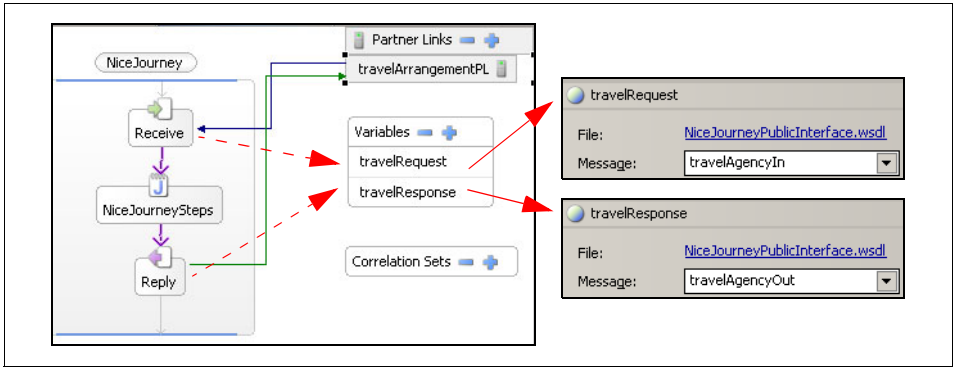


Figure 7-16 Variables defined in BPEL Editor

For each variable, getter and setter methods are generated by WebSphere Studio Application Developer Integration Edition.

- travelRequest
  - getTravelRequest()
  - getTravelRequest(boolean forUpdate)
  - setTravelRequest(TravelAgencyInMessage message)

- travelResponse
  - getTravelResponse()
  - getTravelResponse(boolean forUpdate)
  - setTravelResponse(TravelAgencyOutMessage message)

Now that we have everything together, how can we access the data of the parts of the messages? This is shown in Example 7-4; the comment line shows what we want to do and the code line(s) below shows how to do it. Look at Figure 7-3 on page 151 to compare it with the parts.

Example 7-4 Usage of variables in a Java snippet to access the data of message parts

```
//get variable travelResponse for update of part reservationID
getTravelResponse(true).setReservationID(1234);
//get variable travelRequest, get travelData (complex type) get part cityFrom
String cityFrom = getTravelRequest().getTravelData().getCityFrom();
//create a new instance of message travelAgencyOut
TravelAgencyOutMessage myMessage = new TravelAgencyOutMessage();
//set a value to this new created instance of message travelAgencyOut
myMessage.setReservationID(5678);
//set the new instance of message travelAgencyOut to variable travelResponse
setTravelResponse(myMessage);
```

Important: To get a variable for update purposes, it is necessary to use the getter with a parameter boolean of value `true`, as shown in Figure 7-15 on page 151.Otherwise, the value set to a part of the message may be lost.

Java snippets can be used to:

- Add custom logic to a business process
- Update variables which cannot be updated using the Assign activity, for example, because of complex types.

These may be done in preparation for the activity following the Java snippet or to reflect the result(s) of the activity prior to the Java snippet.

7.2.6 Preparing to develop the process

Before we start developing the process, there are some additional tasks we need to perform. In a real-life situation, when developing a process, a collection of services and code is already available for use in the process. In our case, we do not have any code or service we can use, so we need to create a simple code, a Java class, that we can use as an external service.

There are two options in this book to implement the external service:

- The simplest one is to create an application with the service, consume the service locally and run it on the same test server where the process is running. It is easier to develop this solution but less realistic. This chapter is going to follow this path.
- The other option is to have a more realistic scenario and make the external service really external. In this case, the service is running on a separate test server in the same test environment (this requires two test servers). This scenario is a bit more complex and requires more memory and a better processor to run the development, but it is more realistic. If you want to follow this path, skip this section and refer to “External service for the simple process” on page 565, then continue with 7.2.7, “Developing a new process” on page 156 from this chapter.

The following steps describe how to develop the external service for the same test environment where the process is running.

We will create an Java class called FlightBookingSystem that will be called by the NiceJourney Web Service to reserve a flight.

1. Start WebSphere Studio Application Developer Integration Edition with a new workspace, make sure you enable the server targeting for the workspace under **Preferences** → **J2EE**.
2. Switch to the Business Integration perspective and open the **Services** view.
3. Create a new service project in the Business Integration perspective by selecting **File** → **New** → **Service Project**. Name the service project `Referenced_Partner` and click the **Finish** button.
4. Go to **File** → **New** → **Class** and fill out the details as per Figure 7-17.

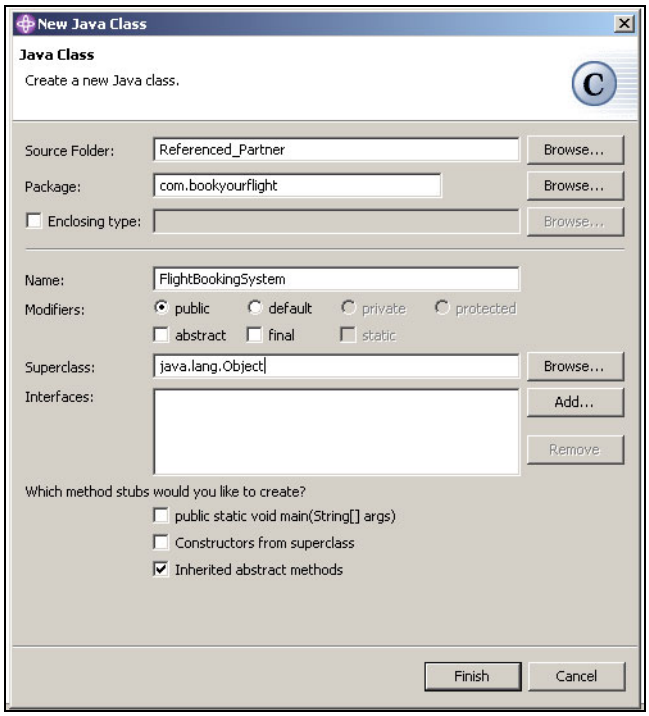


Figure 7-17 New Java class

Click **Finish**.

5. Double-click **FlightBookingSystem.java** in the Package Explorer view of the Business Integration perspective to open the file. Select all (**Ctrl-A**) and press the **Delete** key.
6. Copy the code below into the FlightBookingSystem.java file.

Example 7-5 External service implementation

```
package com.bookyourflight;

import java.util.Calendar;
   public class FlightBookingSystem {

   public FlightBookingSystem() {
      super();
   }

   public long getReservation(String   cityFrom,      String   cityTo,
                          Calendar dateDeparture, Calendar dateReturn) {
      long reservationID;
```

```
System.out.println("-----------------------------------------------------");
System.out.println(getClass().getName()+" reservation done for: ");
System.out.println(" cityFrom= "+cityFrom);
System.out.println(" cityTo= "+cityTo);
System.out.println(" dateDeparture= "+dateDeparture.getTime().toString());
System.out.println(" dateReturn= "+dateReturn.getTime().toString());
reservationID = System.currentTimeMillis();
System.out.println(" reservation ID= "+reservationID);
System.out.println("-----------------------------------------------------");

      return reservationID;
   }
}
```

7. Save and close the file.
8. Right-click **FlightBookingSystem.java** and select **New** → **Service built from**. Select **Java** and click the **Next** button. Select the check box next to **FlightBookingSystem** and click the **Finish** button.

The “external” service, FlightBookingSystem, is ready for use. We will use this service from the simple process introduced in this chapter.

7.2.7 Developing a new process

This section describes how to use the BPEL Editor in WebSphere Studio Application Developer Integration Edition V5.1 to implement a solution for the NiceJourney business scenario.

To design the process for NiceJourney Travel, you will perform the following tasks:

- Create service projects
- Create the WSDL file for the business process
- Create the business process
- Add variables to the business process
- Define PartnerLinks for the business process
- Add the activities

Creating projects for NiceJourney and Flight Booking Service

Important: When developing a process, you should always start with defining the interfaces for the process before you start implementing it.

Generating the interface from the process (bottom-up) is not the recommended method.

1. Switch to the Business Integration perspective and open the **Services** view.
2. Business processes are stored in service projects. Create a new service project in the Business Integration perspective by selecting **File** → **New** → **Service Project**. Name the service project `WPC_Simple_Process` and click the **Finish** button. This project will contain all of your code for the NiceJourney Web service.
3. Right-click the service project **WPC_Simple_Process** and select **New** → **Package** and enter `com.nicejourney` in the Name field.

Creating the NiceJourneyPublicInterface WSDL file

In this example, we will create the NiceJourneyPublicInterface.wsdl file to demonstrate how to use the visual process editor to create a WSDL file and define a WSDL port type.

1. In the Package Explorer view, right-click the **com.nicejourney** package (in the WPC_Simple_Process project) and select **New** → **Empty Service**.
2. Type `NiceJourneyPublicInterface` in the File name field and click the **Finish** button.
3. Double-click **NiceJourneyPublicInterface.wsdl** so that it is active. Select the **Graph** tab in the WSDL Editor.
4. In the next few steps, we are going to show how to use the visual editor to define an interface. Because it is a lengthy process, we are not going to provide all the steps to create the whole interface. We are only going to perform one sample step.

   Right-click the **Types** box. Select **Add Child** → **schema** and click **OK**.
5. Right-click the **Port Types** box. Select **Add Child** → **portType** and type `travelArrangementType` for the Name, then click **OK**.

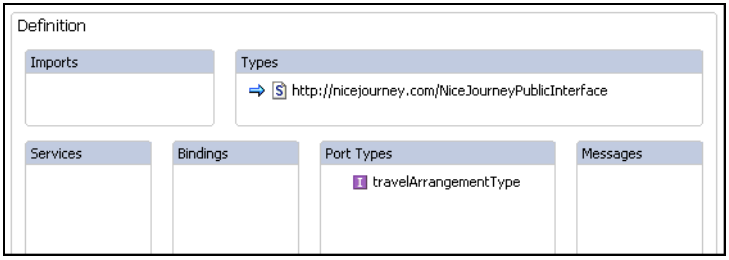


Figure 7-18 Port Types box

   Save your file (**Crtl-S** or **File** → **Save**).
6. Click the **Source** tab in the WSDL Editor to view the generated WSDL code.

7. Before you insert the actual XML code for the interface, you have to delete the code you have just created as a sample.

   Delete the existing code. Select All (**Ctrl-A**) and press the **Delete** key.

8. Copy the code below and paste it into the NiceJourneyPublicInterface.wsdl.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="NiceJourneyPublicInterface"
    targetNamespace="http://nicejourney.com/NiceJourneyPublicInterface"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://nicejourney.com/NiceJourneyPublicInterface"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <types>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
targetNamespace="http://nicejourney.com/NiceJourneyPublicInterface">
<xsd:element name="customer">
   <xsd:complexType>
    <xsd:sequence>
          <xsd:element  minOccurs="0" name="firstName" type="xsd:string"/>
          <xsd:element                name="lastName" type="xsd:string"/>
          <xsd:element                name="address" type="xsd:string"/>
          <xsd:element                name="city" type="xsd:string"/>
          <xsd:element                name="zipcode" type="xsd:nonNegativeInteger"/>
          <xsd:element  minOccurs="0" name="state" type="xsd:string"/>
    </xsd:sequence>
   </xsd:complexType>
</xsd:element>
<xsd:element name="travel">
   <xsd:complexType>
    <xsd:sequence>
          <xsd:element name="cityFrom"      type="xsd:string"/>
          <xsd:element name="cityTo"        type="xsd:string"/>
          <xsd:element name="dateDeparture" type="xsd:dateTime"/>
          <xsd:element name="dateReturn"    type="xsd:dateTime"/>
    </xsd:sequence>
   </xsd:complexType>
</xsd:element>
</xsd:schema>
   </types>
   <message name="travelAgencyIn">
        <part element="tns:customer" name="customerData"/>
        <part element="tns:travel" name="travelData"/>
    </message>
    <message name="travelAgencyOut">
        <part name="reservationID" type="xsd:long"/>
    </message>
   <portType name="travelArrangementType">
       <operation name="travelArrangement">
```

```
            <input message="tns:travelAgencyIn"   name="travelArrangementIn"/>
            <output message="tns:travelAgencyOut" name="travelArrangementOut"/>
        </operation>
    </portType>
</definitions>
```

9. Save and close the NiceJourneyPublicInterface.wsdl file.

Creating the business process for NiceJourney

Now we have defined the WSDL file for the NiceJourney Web service, we will create the business process for NiceJourney Travel using the BPEL Editor. This file will be named NiceJourney.bpel.

1. Right-click the **com.nicejourney** package and select **New** → **Business Process**; type `NiceJourney` in the File name field and click the **Next** button.

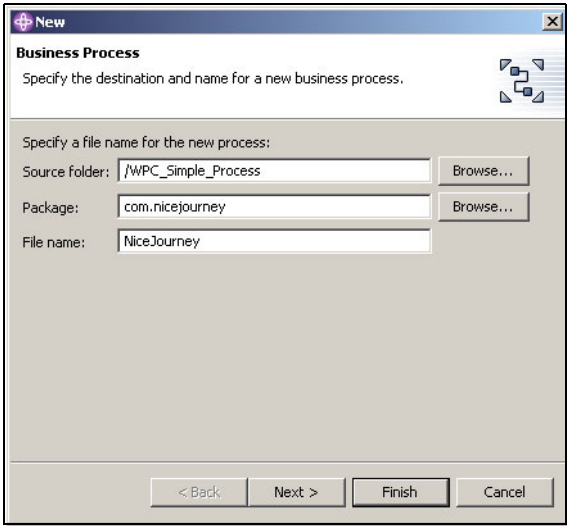


Figure 7-19 New business process - naming details

2. Verify that the **Flow-based BPEL flow** radio button is selected and click the **Finish** button. For further information on Flow- versus Sequence-based processes, see 6.1.4, “Sequences and flows” on page 125.

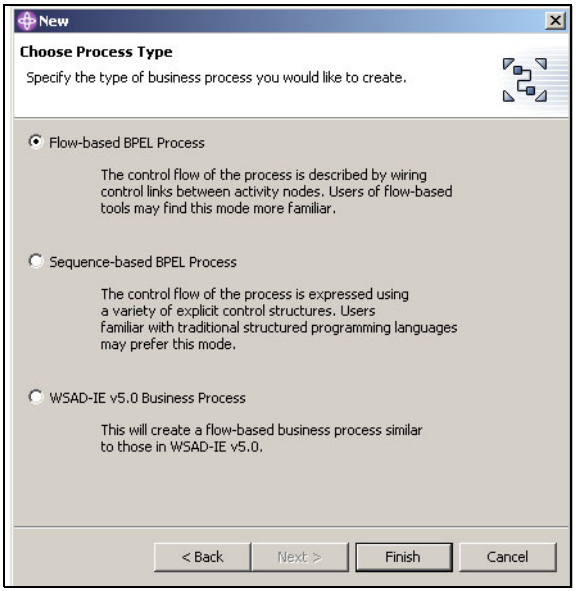


Figure 7-20 New business process - flow type

Adding variables to the NiceJourney process

Now we will add two variables called travelRequest and travelResponse to the BPEL process.

These variables are used as the input and output, respectively, to the travelArrangement operation as defined in NiceJourneyPublicInterface.wsdl. Also, any redundant variables or PartnerLinks will be deleted in this section.

1. Double-click **NiceJourney.bpel** in the Package Explorer window to make it active.
2. Click the **+** graphic on the Variables table to add a variable and name it `travelRequest`.

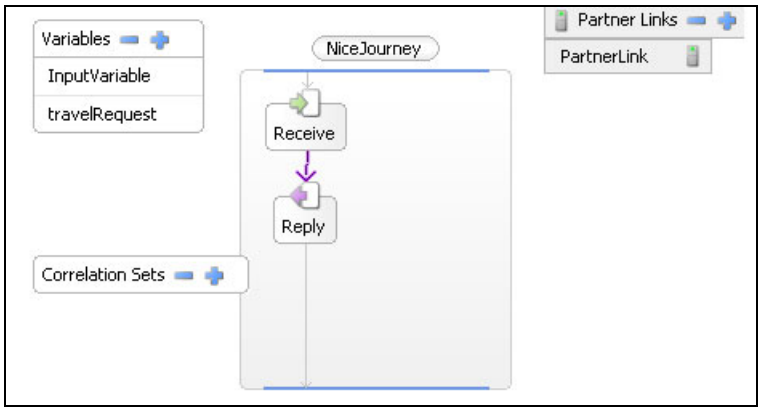


Figure 7-21 Current process design

3. While the travelRequest variable is selected, click the **Message** tab in the Detail Area and click the **Browse** button.
4. Navigate to com.nicejourney.NiceJourneyPublicInterface.wsdl and select **travelAgencyIn** from the drop-down list. Then click the **OK** button.

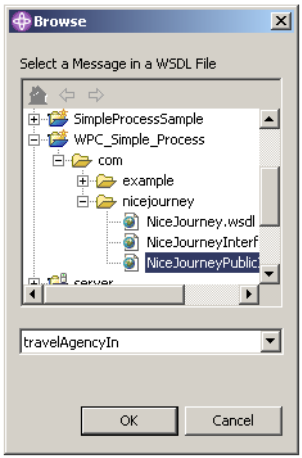


Figure 7-22 Selecting the WSDL for the message

5. Add the variable `travelResponse` and select **travelAgencyOut** from the drop-down list (using the same process used in Steps 2 on page 160, 3 on page 161 and 4 on page 161).
6. Right-click **InputVariable** and select **Delete** since this variable is not used.
7. Right-click **PartnerLink®** and select **Delete**.

Note: Ignore any errors (red crosses); they will be fixed later.

8. Save the NiceJourney.bpel file.

Defining a PartnerLink for NiceJourney

Now we will define the PartnerLink for the NiceJourney BPEL process. Here we will drag and drop the NiceJourneyPublicInterface.wsdl file onto the canvas and then define the roles and operations for the NiceJourney Web service.

Then we will define the messages for the Receive and Reply activities for the travelArrangement operation.

1. Select the **NiceJourneyPublicInterface.wsdl** file from the Package Explorer window and drag it onto the canvas.
2. In the pop-up box, you can see that **travelArrangementType** has been selected. Click **OK**.

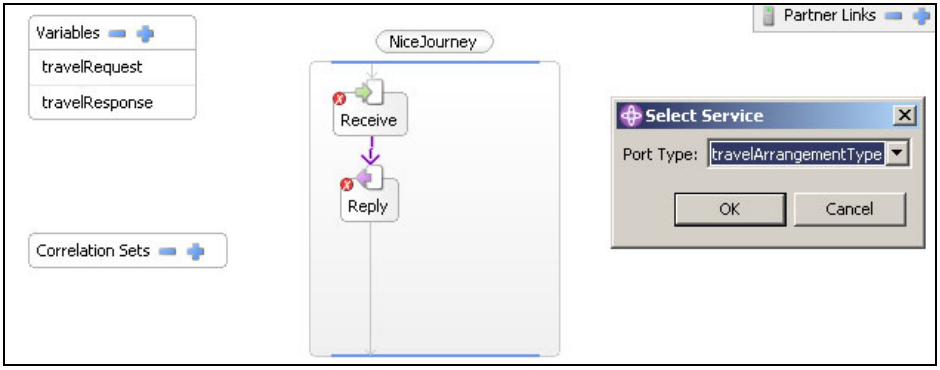


Figure 7-23 Drag and drop the service description WSDL

3. While the **travelArrangementType** partner link is selected, Inspect the Detail Area under the Implementation tab and click the arrows (**<-->**) button to switch the roles to those shown in Figure 7-24.

Note: Switching the roles is necessary, because when we drop a WSDL file on the canvas, WebSphere Studio Application Developer Integration Edition assumes that a partner link should be generated to invoke a partner to this NiceJourney Web service. Hence, it defines a partner's Role Name and no Process Role Name, as shown in Figure 7-24.

In our example, we want to define a partner link with a role that can be used to invoke the NiceJourney Web service. For this, one role is enough and has to be the role of the process, because the NiceJourney Web service does not expect any Web service from the partner which invokes it. For more details about partner links, refer also to 6.1.5, "Parts of a business process" on page 125.

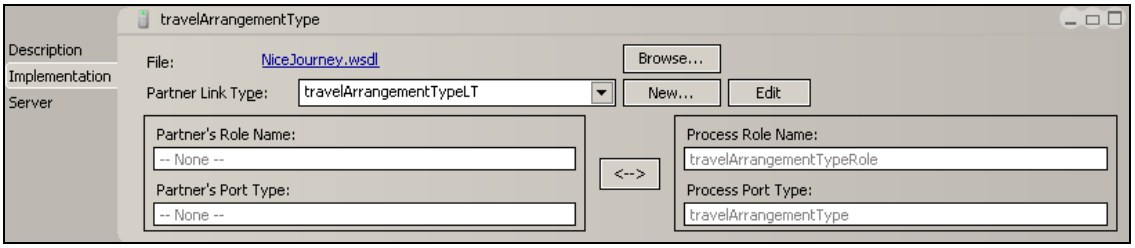


Figure 7-24 Switching the roles for the process

4. Right-click the **Receive** activity and select **Set Partner Link** from the menu. Click the partner link **travelArrangementType** to associate it with the Receive activity. This enables the Receive activity to receive a request via the travelArrangement operation.
5. Click the **Receive** activity and click the **Implementation** tab in the Detail Area. From the Operation drop-down list, select **travelArrangement** and from the Request drop-down, ensure **travelRequest** is selected (this variable covers the right message which is required as input for the travelArrangement operation). Also, verify that the **Create a new Process Instance if one does not already exist** check box is selected.

Note: In the simple process, we have only one Receive activity. If a message arrives to it, a new instance of our simple process should be created, therefore the check box has to be selected (see Figure 7-25).

Note: Assuming that we have two Receive activities in a process, each is able to create a new instance for this process, if both have the check box ticked and both are allowed to create an instance according to the definition on the Correlation tab, as shown in Figure 7-25. The Receive activity which receives the message first will create the instance. The process instances are identified by a Correlation which is mandatory if we have more than one Receive activity in a process. So, the Correlation is also used to check whether or not an instance already exists, to avoid the unnecessary creation of another one. For more details about Correlation, see also 6.1.5, “Parts of a business process” on page 125 and 8.5.7, “Correlation sets” on page 239.

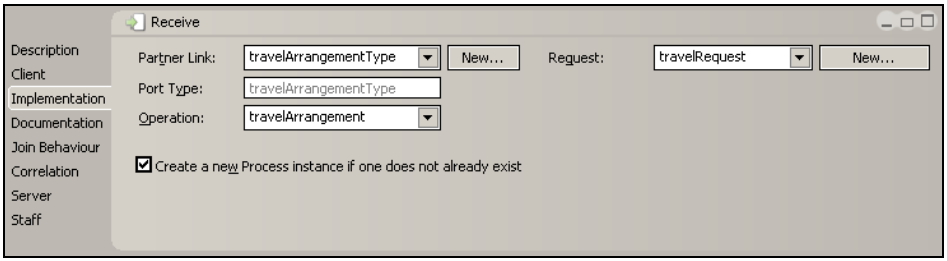


Figure 7-25 Setting the receive partner link

6. Right-click the **Reply** activity and select **Set Partner Link**. Click the partner link **travelArrangementType** to associate it with the Reply activity to send a response using the travelArrangement operation.
7. Click the **Reply** activity and click the **Implementation** tab in the Detail Area. From the Operation drop-down list, select the **travelArrangement** operation. From the Request drop-down menu, ensure **travelResponse** is selected.

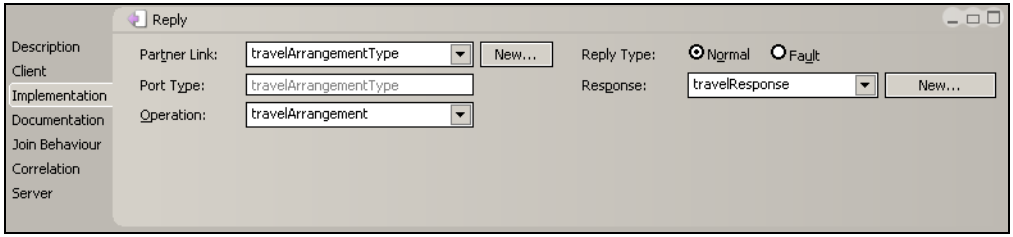


Figure 7-26 Setting the reply partner link

8. Save your file. The errors (red crosses on the Receive and Reply activities) should disappear from the Task view.

Cleaning up the namespace of the process (optional)

This section addresses clean-up of the namespace of our recently created process. If you wish, you can skip this section and go on to the next section “Creating the variables for the Flight Booking Service” on page 166.

Note: If you cannot find the files described in this section, open the Resources perspective to find them.

Because of certain issues related to the workspace, you may see this error in WebSphere Studio Application Developer Integration Edition.

1. Open the NiceJourney.bpel editor if it is not already open and find the top point of the process, labeled `NiceJourney`. Click this and then open the Server tab at the bottom of the BPEL Editor.

   This is where you set properties related to the process itself as opposed to individual activities.
2. Change the target namespace to `http://nicejourney.com/NiceJourney`.
3. Save and close the .bpel file.
4. Switch to the Package Explorer view and open NiceJourney.wsdl from the WPC_Simple_Process/com/nicejourney folder. This will open the file in the WSDL editor. Switch to the Graph view, rather than the Source code view, if it did not open automatically.
5. In the bottom left of the editor is the Definitions section. Click the **Edit Namespaces...** button.

   Tip: If you do not see the Edit Namespaces... button then you may have selected one of the import types, services, bindings, port types or messages. You may have to click an area of white space in the top section to deselect all of these, at which point the Edit Namespaces... button should appear.
6. Change the Target Namespace value to `http://nicejourney.com/NiceJourney` and click **OK**.
7. In the top half of the editor, right-click the import for NiceJourneyInterface.wsdl and click **Delete**. This removes the import for the default namespace. This import will be correctly regenerated later with the updated namespace when we update the process partner link.
8. Save and close the file. Ignore the errors that may appear in the Tasks view at this point.

9. Now open the NiceJourneyPublicInterface.wsdl file and click the **Edit Namespaces...** button in the same way. Change the Target Namespace value, this time using `http://nicejourney.com/NiceJourney/interface` to indicate that this is the namespace for the process *interface*.
10. Select the item under Types, then change the URI under the Schemes to `http://nicejourney.com/NiceJourney/interface`. Click **Apply** on this view.
11. Save and close all files.
12. Optionally, delete the WPC_Simple_Process/com/example folder and its contents from the Navigator view in the Resource perspective (this view allows you to work with the underlying file system directly). This will remove the remaining unwanted entries, produced by default from the old namespace values that were set by creating a new business process.

Creating the variables for the Flight Booking Service

We are going to define the variables required for the Flight Booking Service.

1. Open the NiceJourney.bp el if it is not already open.
2. Create two new variables called `flightRequest` and `flightResponse` (see “Adding variables to the NiceJourney process” on page 160 for how to do this).

Note: The messages covered by these variables are described in FlightBookingSystem.wsdl, where flightRequest covers getReservationRequest and flightResponse covers getReservationResponse.

Figure 7-27 List of variables

3. Select the **flightRequest** variable and in the Message tab under the Detail Area, click the **Browse** button. Navigate to FlightBookingSystem.wsdl, select the **getreservationRequest** at the bottom and click the **OK** button.

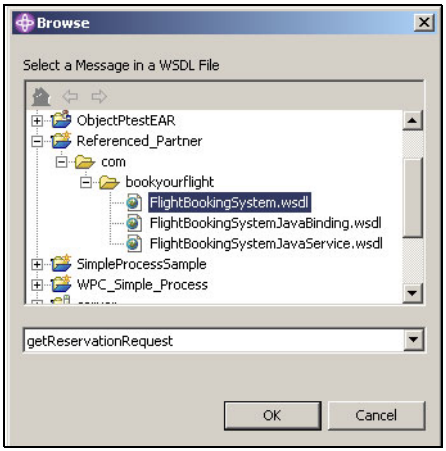


Figure 7-28 Selecting the message from the WSDL

4. You will see that the getReservationRequest message is now displayed in the Message tab in the Detail Area.

Figure 7-29 flightRequest message

Repeat Steps 3 on page 166 and 4 on page 167 and select **getReservationReponse** from the drop-down list. The Parts box now contains a result, which is the Reservation ID returned from the Flight Booking System to NiceJourney Travel.

Figure 7-30 flightResponse message

5. Save the file.

Creating a PartnerLink for the Flight Booking Service

At this point, you have to use the external service to define your partner link. If you chose to use a separate server for a more realistic scenario, then this is the time to get the service description from the external server. If you chose to use the external service locally, running on the same test server, then skip the next six steps and start with the first step under “Using the external service in the partner link” on page 168.

1. Create a new simple project called `Referenced_Partner`, and create a new folder structure: com/bookyourflight.

Open a Web browser (outside WebSphere Studio Application Developer Integration Edition) and access the URL:

```
http://localhost:9087/bookyourflight.comWeb/wsdl/com/bookyourflight/
FlightBookingSystem.wsdl
```

2. Save the file to a temporary directory, for example: C:\temp.
3. Right-click the **com/bookyourflight** folder, then select **Import**. Select **File System**, then click **Next**.
4. Browse for the directory: C:\temp, select the file: **FlightBookingSystem.wsdl**.
5. Click **Finish**.

Using the external service in the partner link

1. Create a new Partner Link named `flightBooking` by clicking the **+** graphic.

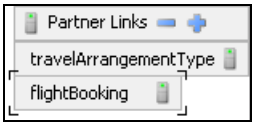


Figure 7-31 List of Partner Links

2. In the Detail Area, on the Implementation tab, click the **Browse** button, navigate to NiceJourney.wsdl and click the **OK** button.
3. In the Detail Area, on the Implementation tab, click the **New** button next to the PartnerLinkType box.
4. Fill out the dialog box as shown below.
   a. For File, select **Browse** and navigate to WPC_Simple_Process/com/bookyourflight/NiceJourney.wsdl.
   b. Verify that the **One Role** radio button is selected.

c. Name the First role: `flightBooker`.

d. For Port Type File, select **Browse** and navigate to Referenced_Partner/com/bookyourflight/FlightBookingSystem.wsdl.

e. For Port Type, select **FlightBookingSystem** from the drop-down list.

f. The dialog box should look like the screen capture below (now we have only the role of a service requester). Click **OK**.

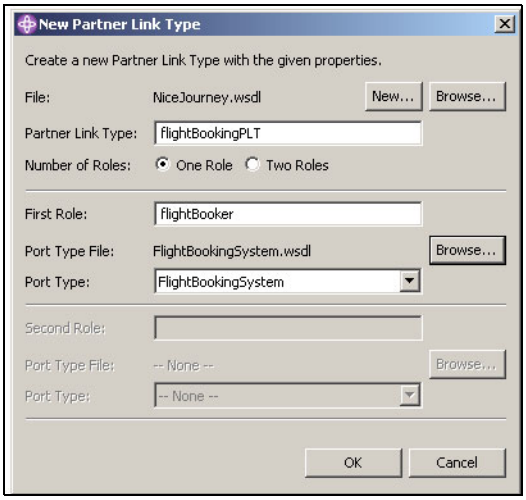


Figure 7-32 New partner link type

5. Select the link between the Receive and Reply activities and press the **Delete** key.

> **Note:** Since it is not possible to reassign a link to another activity, we have to delete it and set a new link between involved activities. Deatils regarding setting the new link will follow later on.

Note: Links indicate the flow of the process at runtime, from the source to the target of the link. A link is evaluated if the previous activity has finished. For a link, a condition on which the link evaluates to `true`, can be defined. On a selected link, the Condition tab can be used to define this. In the combo box of the Condition tab, one of the following conditions can be selected:

- Visual Expression - for visual building of branches on existing variables and fixed values, to define the true condition.
- Expression - Java Code can be defined, as of Java snippets, to define the true condition.
- True - fixed value of true, this is the default value for a new link.
- False - fixed value of false.
- Otherwise - relevant for an activity with several outgoing links, to ensure that this link evaluates to `true` if all others have evaluated to `false`.

The result of a link is used in the target activity of it, on the Join Behavior tab in the Detail area.

Adding the Invoke activity

In this section, we will add an Invoke activity to the NiceJourney BPEL process that will invoke the getReservation operation on the Flight Booking Web Service.

1. Click the **Invoke** activity on the Palette (graphic with the two arrows) and click the flow again to drop it onto the canvas.

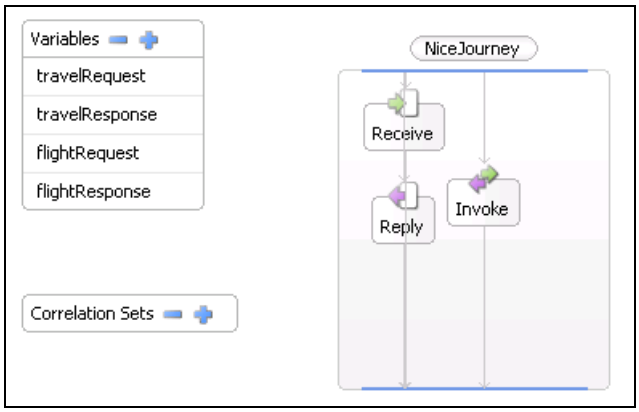


Figure 7-33 Current process design

2. Now we will create a link between the Invoke and Reply activities.

   Right-click the **Invoke** activity on the canvas and select **Set link between flow activities**. Then click the **Reply** activity to link the activities together.

   Tip: To align the activities and links, right-click the flow activity and select **Arrange Flow Contents**.

3. Next, we will create a link between the Invoke activity and the Flight Booking System PartnerLink so NiceJourney Travel can organize flight reservations. Select the **Invoke** activity on the flow. On the Implementation tab in the Detail Area, fill out the fields as shown in the next figure.

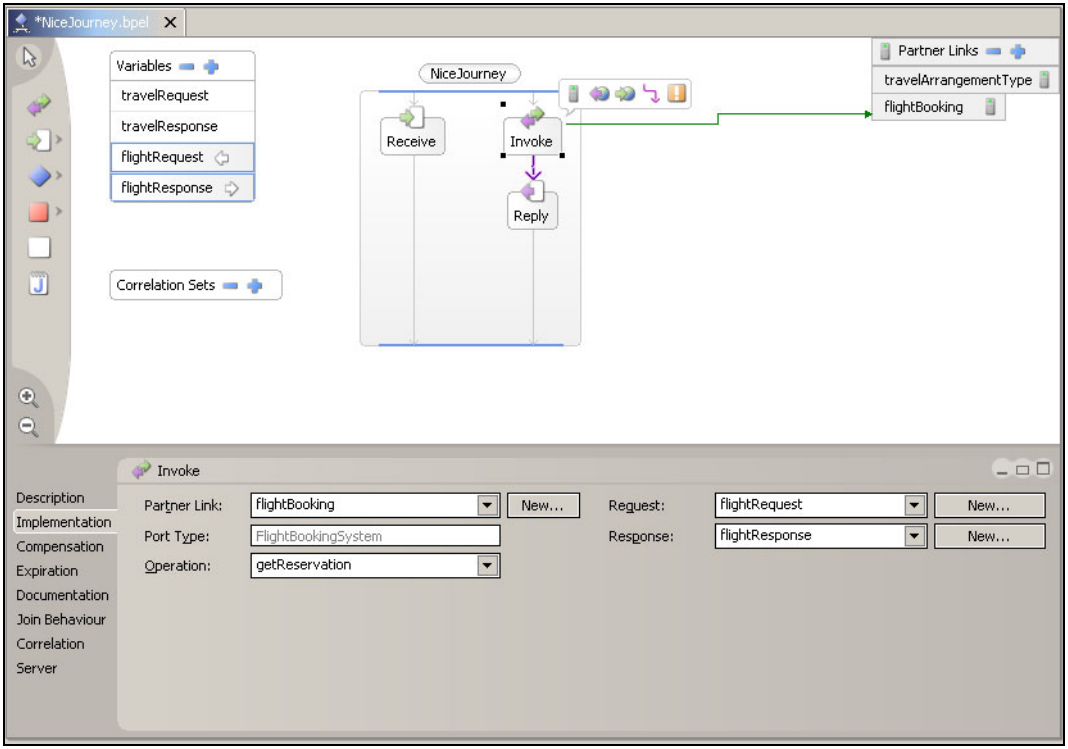


Figure 7-34 Setting up the PartnerLink

4. Save the .bpel file.

Adding an Assign activity

In this section, we need to assign the values of the parts in travelRequest and travelResponse to the same (and currently empty) parts in the flightRequest and flightResponse variables.

Note: The Assign activity is required because the travelRequest variable contains more parts than are needed for the getReservation operation. Hence, we will assign the values from the original Request to a new variable flightRequest that we can send successfully to the getReservation operation for the Flight Booking Service.

Even if the travelRequest contained only those parts which are required by the flightRequest, it would not be possible to use them directly, because the messages are defined in different namespaces.

1. From the Palette, select the **Assign** activity and click the flow activity again to drop it onto the canvas.
2. Create a link between the Assign and Receive activities. Right-click the **Receive** activity and select **Set link between flow activities**. Then click the **Assign** activity.
3. Click the **Assign** activity and then the **Implementation** tab in the Detail Area. Select the values from the drop-down list and box to match the figure shown below.

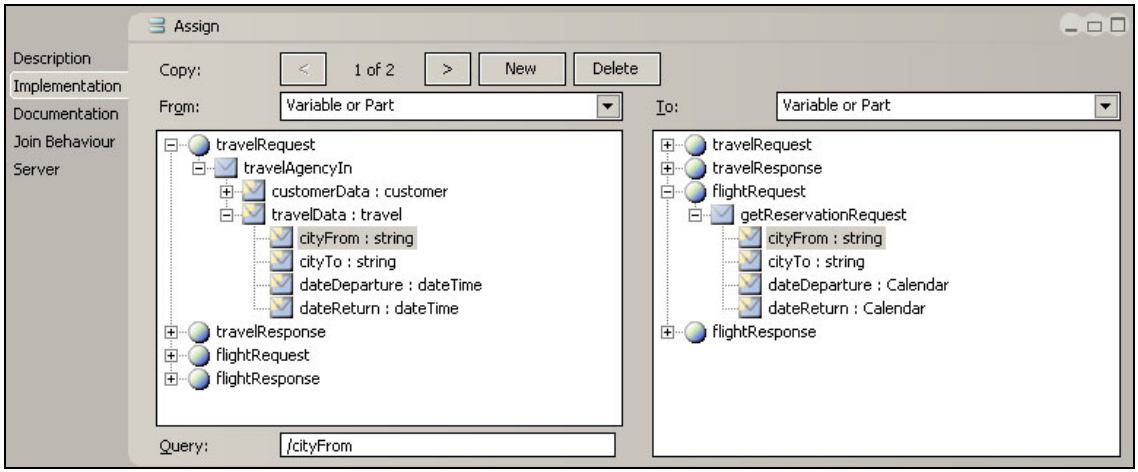


Figure 7-35 Variable mapping in the Assign activity

4. Click **New** in the Detail Area to assign the value travelData → cityTo to flightRequest → cityTo.

5. You should have a figure similar to the one below in your BPEL Editor.

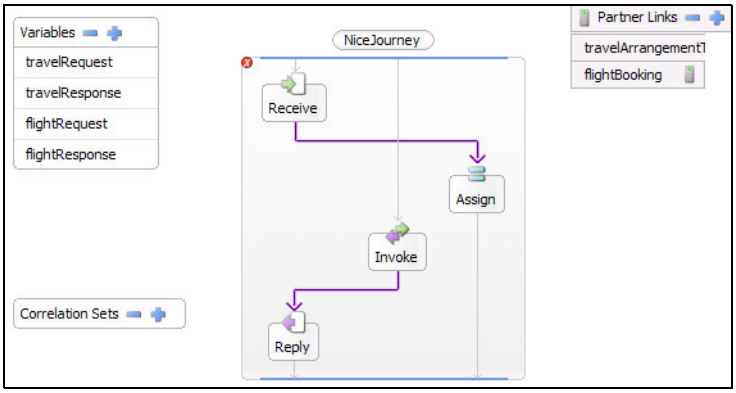


Figure 7-36 BPEL Editor

6. Save the .bpel file.

Adding a Java snippet activity

Due to a complication whereby complex xsd types are not always converted to appropriate Java classes, we need to define a Java snippet where we will add some code to perform this conversion manually.

From the variable travelRequest, we will assign the parts dateDepature and dateReturn, both of type xsd:dateTime, to the variable flightRequest parts dateDeparture and dateReturn, both of type java.util.Calendar.

Note: The variable flightRequest contains the parts dateDeparture and dateReturn. Both are defined in the file FlightBookingSystem.wsdl of type xsd1:Calendar, as follows:

```
...
<types>
 <schema ... targetNamespace="http://util.java/" ...
  <complexType name="Calendar">
  <all>
   <element name="firstDayOfWeek" type="int"/>
   <element name="time" nillable="true" type="dateTime"/>
   <element name="lenient" type="boolean"/>
   <element name="minimalDaysInFirstWeek" type="int"/>
   <element name="timeZone" nillable="true" type="xsd2:TimeZone"/>
  </all>
  </complexType>
 </schema>
</types>
<message name="getReservationRequest">
 <part name="dateDeparture" type="xsd1:Calendar"/>
 <part name="dateReturn" type="xsd1:Calendar"/>
</message>
...
```

The final mapping to java.util.Calendar is defined in the file FlightBookingSystemJavaBinding.wsdl, as follows:

```
<format:typeMap formatType="java.util.Calendar"
typeName="xsd1:Calendar"/>
```

This cannot be done using the Assign activity, because the xsd:dateTime is converted to a java.util.Date by WebSphere Studio Application Developer Integration Edition, which cannot be directly assigned to java.util.Calendar. The following example shows how to perform this conversion within a Java snippet for the dateDeparture parts of travelRequest and flightRequest.

Example 7-6 Conversion java.util.Date (results from xsd:dateTime) to java.util.Calendar

```
//assign travelRequest part to filghtRequest part
//to do this a conversion is necessary
Calendar calDeparture = Calendar.getInstance();
calDeparture.setTime( getTravelRequest().getTravelData().getDateDeparture() );
getFlightRequest(true).setDateDeparture( calDeparture  );
```

Note: It would make more sense if the xsd:dateTime were converted to a java.util.Calendar by WebSphere Studio Application Developer Integration Edition, because of the support of java.util.TimeZone and java.util.Locale.

Here are the steps to add the Java snippet to the NiceJourney process.

1. Select a Java snippet activity from the Palette and drop it onto the flow activity.
2. Create a link from the Assign activity to the Java snippet activity (right-click and select **Set flow between activities**).
3. Create a link from the Java snippet activity to the Invoke activity (right-click and select **Set flow between activities**).
4. Since this is now the final flow structure, right-click the flow activity and select **Arrange Flow Contents** to order the activities as shown below.

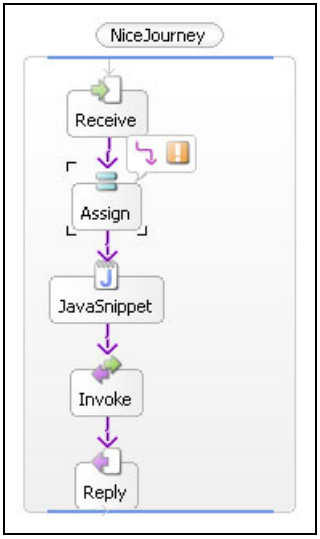


Figure 7-37 Current process design

5. Select the Java snippet activity and select the **Implementation** tab in the Detail Area. Remove all the code from the implementation, if there is any.
6. Copy and paste the following code into the Java snippet.

Example 7-7 Implementation of the Java snippet of simple process

```
//assign travelRequest parts to filghtRequest parts
//to do this a conversion is necessary
Calendar calDeparture = Calendar.getInstance();
```

```
calDeparture.setTime( getTravelRequest().getTravelData().getDateDeparture() );
getFlightRequest(true).setDateDeparture( calDeparture  );

Calendar calReturn = Calendar.getInstance();
calReturn.setTime( getTravelRequest().getTravelData().getDateReturn() );
getFlightRequest(true).setDateReturn( calReturn );

//only for controlling purpose
System.out.println("travelRequest.customer.firstname:
"+getTravelRequest().getCustomerData().getFirstName());
System.out.println("travelRequest.customer.lastname:
"+getTravelRequest().getCustomerData().getLastName());
System.out.println("travelRequest.customer.adress:
"+getTravelRequest().getCustomerData().getAddress());
System.out.println("travelRequest.customer.city:
"+getTravelRequest().getCustomerData().getCity());
System.out.println("travelRequest.customer.state:
"+getTravelRequest().getCustomerData().getState());
System.out.println("travelRequest.travel.cityFrom:
"+getTravelRequest().getTravelData().getCityFrom());
System.out.println("travelRequest.travel.cityTo:
"+getTravelRequest().getTravelData().getCityTo());
System.out.println("travelRequest.travel.dateDeparture:
"+getTravelRequest().getTravelData().getDateDeparture().toString());
System.out.println("travelRequest.travel.dateReturn:
"+getTravelRequest().getTravelData().getDateReturn().toString() );
```

7. If you save your file and look at this code, the Calendar object will have a red line beneath it indicating an error in the Java snippet. To fix this error, we need to import the Calendar class.

   Select the **NiceJourney** heading on the canvas, then select the **Imports** tab and type:

   ```
   import java.util.Calendar;
   ```

   If you refer to the Java snippet now, the error will have disappeared.

Changing the condition of a link

The last step is to assign the response from the FlightBookingSystem to be the response of the NiceJourney Web service. For this, the travelResponse will be taken from the flightResponse.

Note: The response of the FlightBookingSystem cannot be directly passed to be the response of the NiceJourney Web Service, because of different namespaces; see NiceJourneyPublicInterface.wsdl and FlightBookingSystem.wsdl.

To do this, there are a number of options, including using an assign or Java snippet activity. To demonstrate another way to perform this task, we will add Java code on the condition of a link.

1. Click the link between the Invoke and Reply activities.

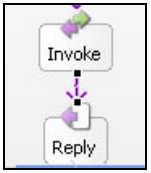


Figure 7-38 Selecting the link

2. Click the **Condition** tab in the Detail Area and select **Expression** from the drop-down list.
3. Type the following line above `return true;` in the text box:

   ```
   getTravelResponse(true).setReservationID(getFlightResponse().getResult());
   ```

   This will set the ReservationId in the travelResponse variable to the value of the result from the FlightReponse variable and is the final step in the design process.
4. Save and close the file. Make sure there are no errors in the process.

7.2.8 Deploying and testing a process in the IBM WebSphere Test Environment

A process can run in WebSphere Test Environment within the WebSphere Studio Application Developer Integration Edition V5.1 environment if these tasks are peformed:

1. Generate deploy code for the process to create an enterprise application.
2. Create a test server, configure it and add the process to the server.
3. Start the server and use the process Web client to start and stop instances of the process.

Generating deploy code

You must generate deploy code for a process so that it can be run on an application server. The Generate Deploy Code wizard creates an enterprise application that can be deployed to WebSphere Business Integration Server Foundation V5.1. With enterprise services, you can generate deploy code with one of three inbound bindings:

- EJB binding
- SOAP binding
- JMS binding

To generate deploy code for NiceJourney process, perform the following steps:

1. In the Services view of the Business Integration perspective, expand **WPC_Simple_Process** → **com.nicejourney** → **NiceJourney.bpel**.
2. Right-click **NiceJourney.bpel** and select **Enterprise Services** → **Generate Deploy Code**.
3. Select the **flightBooking** Partner Link in this dialog and click the **Browse** button. Select **Referenced_Partner** → **com** → **bookyourflight** → **FlightBookingJavaService.wsdl**. Your screen should look like Figure 7-39. Click the **OK** button to generate the deploy code.

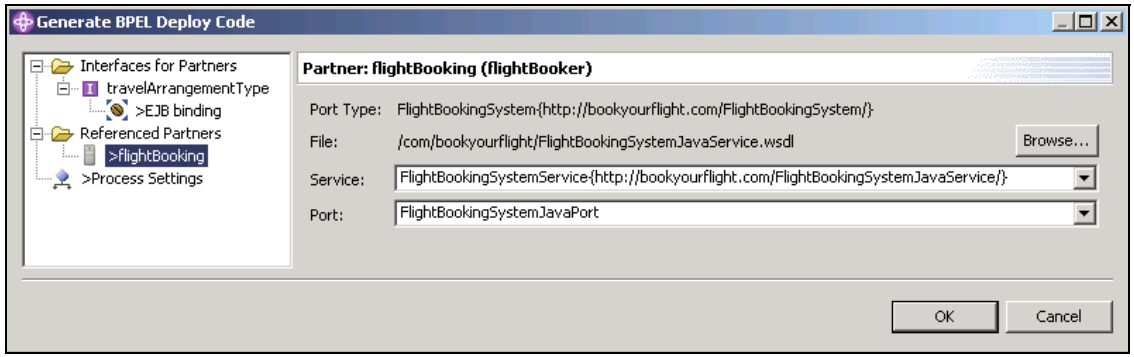


Figure 7-39 Generating the BPEL deployed code

The following projects are generated, containing the deploy code.

- WPC_Simple_ProcessEAR
- WPC_Simple_ProcessEJB
- WPC_Simple_ProcessWeb

Regenerating deploy code

If you have made changes in the BPEL Editor on your process, it is necessary to regenerate the deploy code.

Before you generate the deploy code again, it is recommended that you completely delete the previously generated deploy code. In this way, you can make sure that everything is generated anew and that no artifacts from the previous generated code are left, causing problems when the code is regenerated.

To delete the previously generated deploy code of the NiceJourney Web Service, perform the follow steps:

1. Select the generated projects **WPC_Simple_ProcessEAR**, **WPC_Simple_ProcessEJB** and **WPC_Simple_ProcessWeb**.

   This can be done by holding down the **Ctrl** key and left-clicking each of the above mentioned projects.
2. Press the **Delete** key.
3. In the upcoming confirmation dialog, select **Also delete contents in the file system** and click **Yes**.

Now you can perform the steps described in “Generating deploy code” on page 177 to built the deploy code again.

Deploying a process to the WebSphere Test Environment

To run a process, you need to:

- Create a server and service instance of the WebSphere Application Server (or reuse an existing one).
- Add the process (and any other enterprise application used by the process) to the server configuration.
- Deploy the process to the server.

To deploy the NiceJourney process to a server, perform the following steps:

1. Select the **Server** perspective. Right-click **Servers** and select **New** → **Server and Server Configuration**.
2. Name the server `NiceJourneyServer`, verify that **Integration Test Environment** is checked and click the **Finish** button.

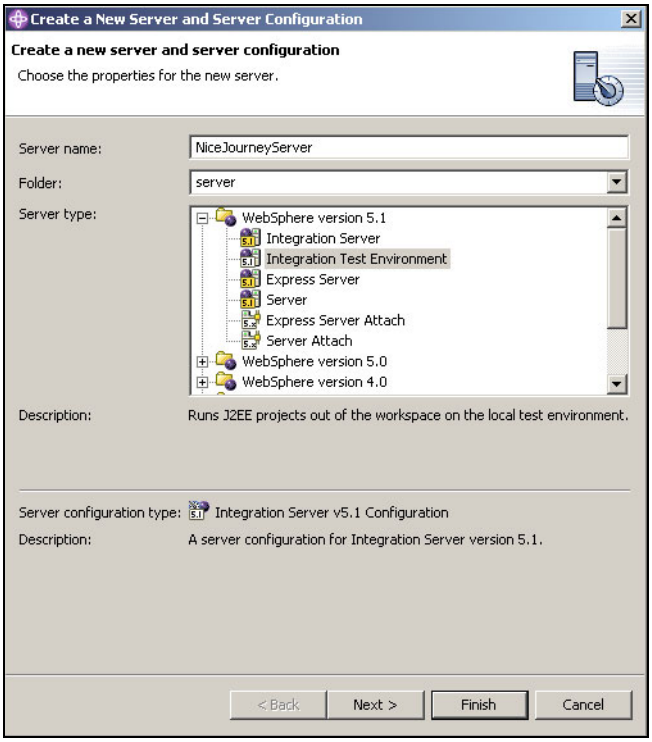


Figure 7-40 Creating the WebSphere Test Environment

3. Right-click **NiceJourneyServer** and select **Add or Remove Projects**. Select **WPC_Simple_ProcessEAR** and click the **Add** button. Click the **Finish** button.
4. The Server Configuration should now look like Figure 7-41.

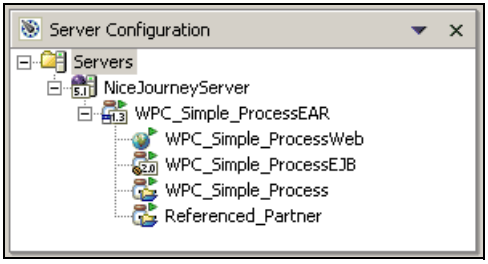


Figure 7-41 Server configuration view

Using the process Web client

The following steps show you how to use the process Web client with your new process.

1. Right-click **NiceJourneyServer** and select **Start** to start the server.
2. When the server has started (the text on the Console has stopped scrolling), switch to the Servers view, right-click the **NiceJourneyServer** and select **Launch Business Process Web Client**.
3. Click the **My Template** link in the Web client and select the check box next to **NiceJourney**. Click the **Start Instance** button.

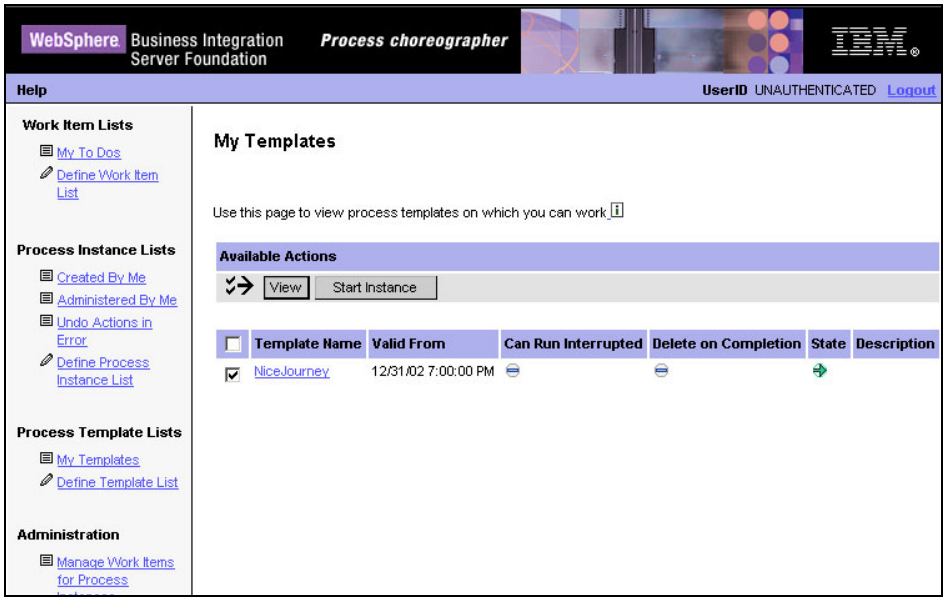


Figure 7-42 Process Web client - My Templates view

4. Fill out the details in the Process Input Message section.

> **Important:** Ensure the integer and dateTime fields have valid values; see the example in Figure 7-43.

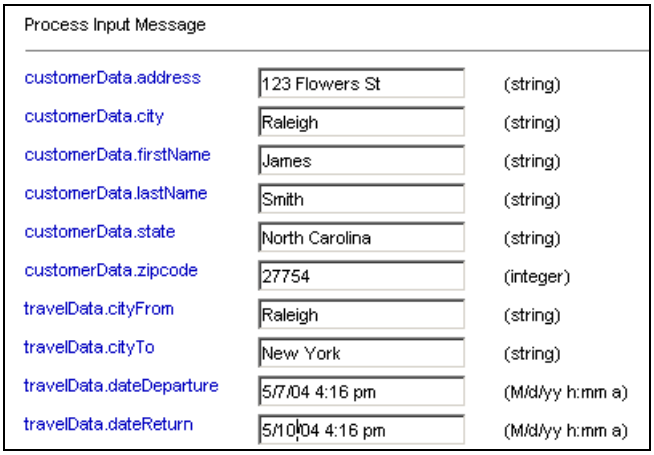


Figure 7-43 Process input messages

5. Click the **Start Instance** button to run the NiceJourney Web service.

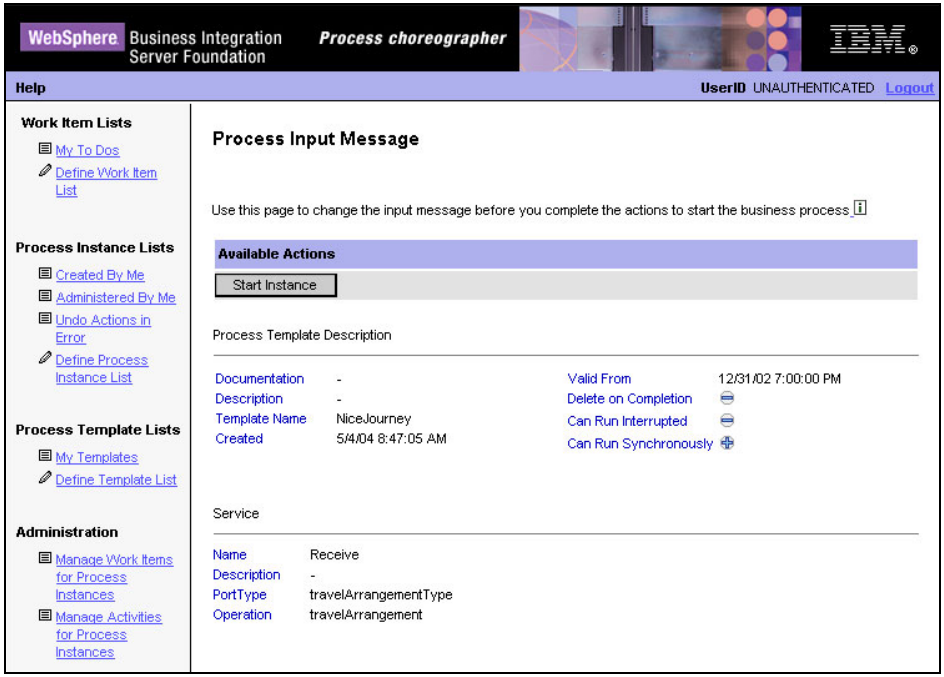


Figure 7-44 Starting the instance for the process

6. You will receive output from the NiceJourney Web service, a reservation ID. This was generated by the FlightBookingSystem which is referenced via partner link flightBooking.

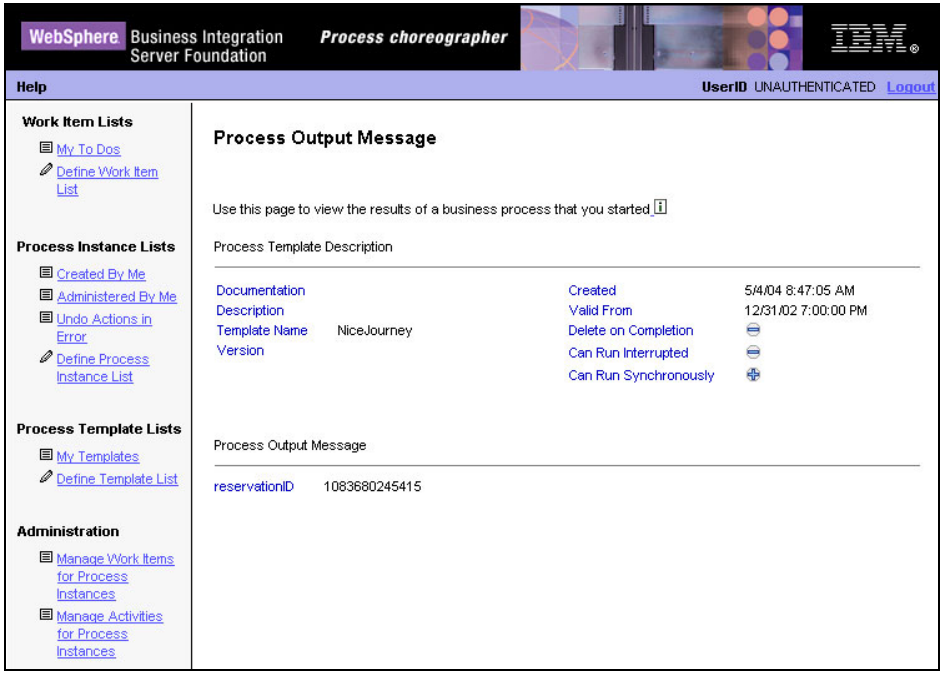


Figure 7-45 Output messages from the process

The simple NiceJourney Web service has now run successfully.

Refer to the Console of the WebSphere Test Environment to see the trace messages from the NiceJourney Web Service and from its reference partner, the FlightBookingSystem.

7.2.9 Debugging a process in WebSphere Test Environment

Once a business process is developed and deployed, it might be necessary to debug it to solve certain problems. WebSphere Studio Application Developer Integration Edition provides a Process Debugger to debug a business process at its logical level, which is more practical than to debug at the code level.

Process debugging components

For debugging purposes, there are two different components, the Process Debugger and the Process Engine.

- Process Debugger

  The Process Debugger is a component of WebSphere Studio Application Developer Integration Edition. It is used to set breakpoints in a business process on activities and on links. Once an instance of a process template with defined breakpoints is created, the Process Debugger comes up if the process flow reaches the first breakpoint. It can be used for several tasks on a process instance such as stepping through the process, viewing the value of process variables, stepping into source code, terminatng the process instance and so on. The Process Debugger is available in the Debug view of WebSphere Studio Application Developer Integration Edition.

- Process Engine

  The Process Engine always runs in an application server. This can be a WebSphere Test Environment of WebSphere Studio Application Developer Integration Edition, or WebSphere Business Integration Server Foundation. It executes the activities of the business process.

The Process Debugger and Process Engine can work together for debugging purposes, even if they reside on different machines; for details, refer to 7.2.11, “Debugging a process on WebSphere Business Integration Server Foundation” on page 194. For debugging purposes, at development time, they may reside on the same machine. This is detailed in the rest of this chapter.

Setting and removing breakpoints

Setting breakpoints on the activities of a business process can be done in the BPEL Editor. A breakpoint is indicated by one of the following symbols.

Symbol	Description
	The breakpoint is uninstalled, enabled, and restricted to one or more instances of the process
	The breakpoint is installed, enabled, and restricted to one or more instances of the process
	The breakpoint is uninstalled and enabled in all instances of the process
	The breakpoint is installed and enabled in all instances of the process
	The breakpoint is uninstalled and disabled.
	The breakpoint is installed and disabled.

Figure 7-46 Symbols to indicate different states of a breakpoint

On each activity, a breakpoint can be set at the entry of the activity and/or at the exit of the activity, which means giving the control flow to the Process Debugger

just before the activity is executed by the Process Engine, or just after the activity has been executed.

Setting breakpoints for a certain activity can be done by right-clicking it and selecting the according breakpoint to be set from the pop-up menu, as shown in Figure 7-47.

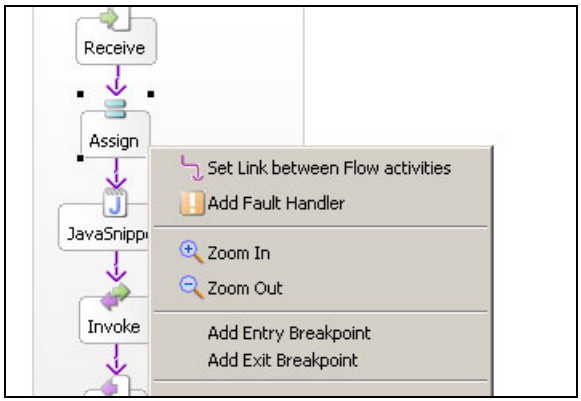


Figure 7-47 Setting a breakpoint for an activity

In the case of a Java snippet, breakpoints can additionally be set for every line of code of the snippet, by double-clicking the grey area in front of the corresponding line of code, as shown in Figure 7-48.

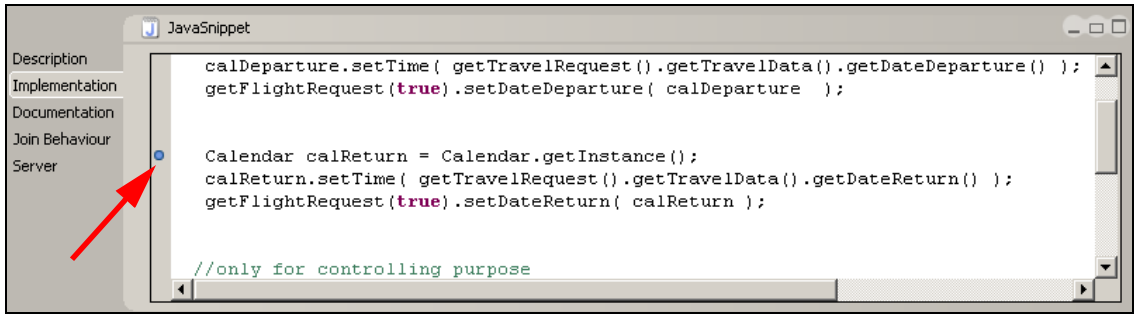


Figure 7-48 Adding a breakpoint to a line of code in a Java snippet

On a link, breakpoints can only be set if the condition of the link is an expression type. Setting breakpoints to a link can be done in the same way as for a Java snippet.

Removing breakpoints from a certain activity can be done by right-clicking it and selecting the according breakpoint to be removed from the pop-up menu.

Breakpoints in Java snippets or in links can be removed by double-clicking the breakpoint to be removed.

Process debugging views

For the purpose of debugging a business process, WebSphere Studio Application Developer Integration Edition provides the Debug Perspective, which can be added to your workspace by clicking **Window** → **Open Perspective** → **Other...**, then selecting **Debug** from the dialog and clicking **OK**.

The Debug Perspective contains the following views:

- Breakpoints view

  The Breakpoints view shows all the breakpoints which have been assigned to the business process. When you right-click a breakpoint, a pop-up menu appears which can be used to change certain properties of the breakpoint or to disable/enable it. Furthermore, it is possible in this view to restrict a breakpoint so that it only applies to a specific process instance and is ignored by other process instances.

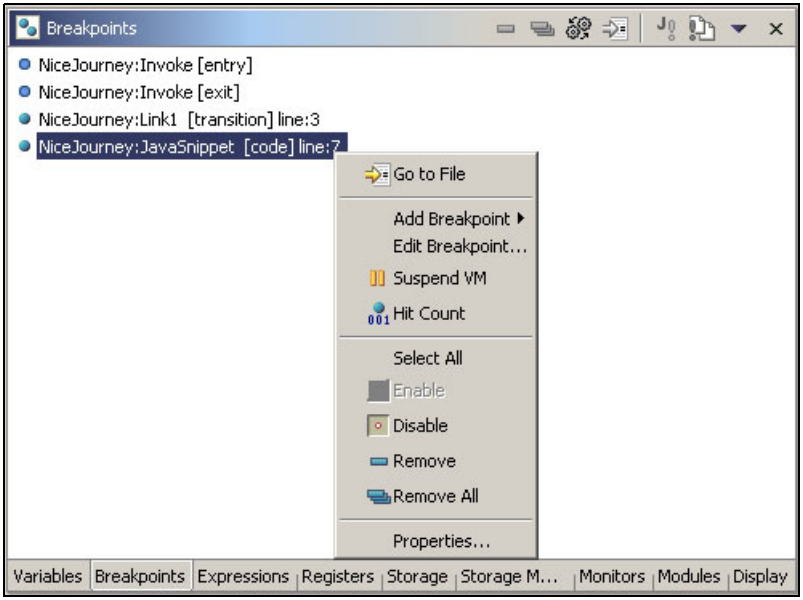


Figure 7-49 Breakpoint view in Debug Perspective

- Debug view

  The Debug view displays information about the currently running threads and process instances on the application server to which it is attached. It also shows the name of the activity which causes a suspension on a thread,

because of a breakpoint. From this view, the debugging tasks (see also “Debugging tasks for a process” on page 188) can be performed by using the tool bar shown in Figure 7-50.

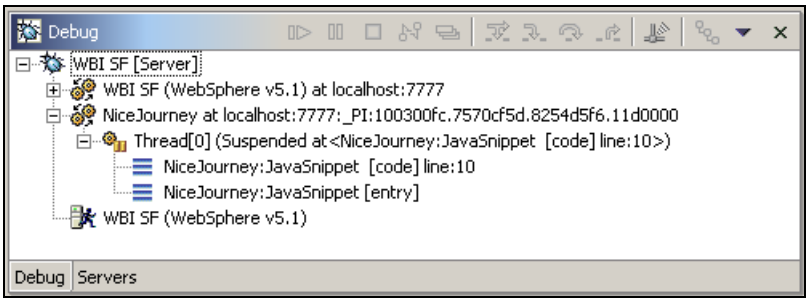


Figure 7-50 Debug view in Debug Perspective

- Variable view

  The Variable view shows all process variables, partner links, and their current values. The content shown in the Variable view is always associated with the selected activity in the Debug view and the activity which causes the block of the thread. It is also possible to change the value of a variable at debugging time by right-clicking the according value.

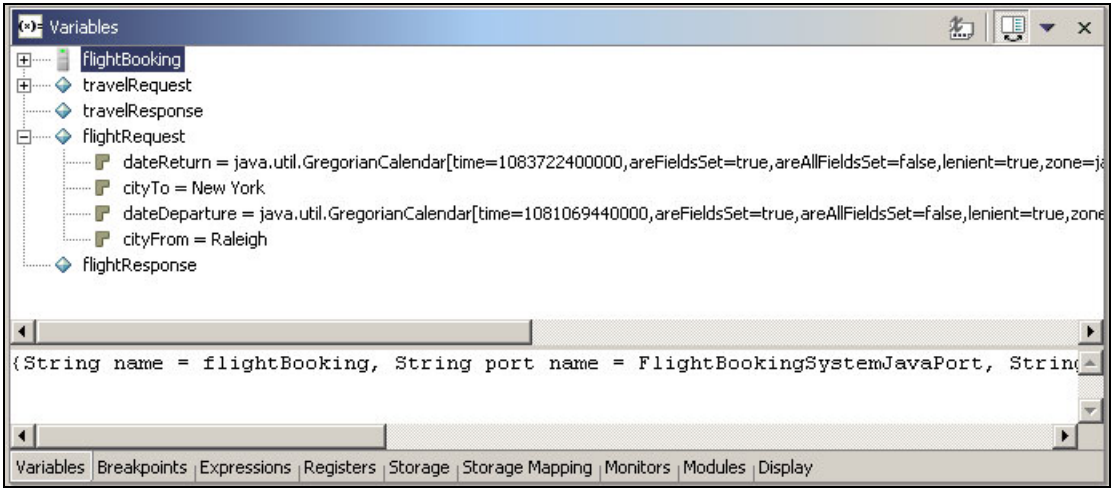


Figure 7-51 Variables view in Debug Perspective

The three views are refreshed when a debugging task is performed; this is described in the next section.

Debugging tasks for a process

Once a breakpoint is reached in a process instance, the Debug view of the Debug Perspective can be used to step through the process instance. The Debug view may be used in combination with the Variable view to learn about the flow in the process instance and the values of the variables as the flow continues.

The debugging tasks are available from the Run menu of the Debug Perspective as well as in the toolbar of the Debug view, shown below.

Figure 7-52 Debug view with toolbar to perform debugging tasks

The following table shows a subset of the available debugging tasks, the most frequently used to debug a process instance.

Table 7-1 Debugging Tasks available for a process instance

Symbol	Name	function key	Description
	Resume	F8	Resume the execution of the processes instance, after it has been interrupted by a breakpoint. It can be used to jump to the next breakpoint.
	Terminate	n/a	Terminate the process instance. To do this, select the activity which causes the block, because of the breakpoint, and click the terminate button.
	Step Into	F5	Step into a Java snippet or into a link which has an expression condition type.
	Step Over	F6	Step over an activity, to the next.
	Step Return	F7	Step to the return of a Java snippet or to return of a link which has an expression condition type.

At this point, we have everything prepared to start debugging a process instance.

Debugging a local process instance

In section 7.2.7, “Developing a new process” on page 156, we developed a new process which then was deployed to a local server in section 7.2.8, “Deploying and testing a process in the IBM WebSphere Test Environment” on page 177. In

this section, we discovered how to add breakpoints and how to work with the Debug Perspective of WebSphere Studio Application Developer Integration Edition.

In the NiceJourney process, we have set the following breakpoints.

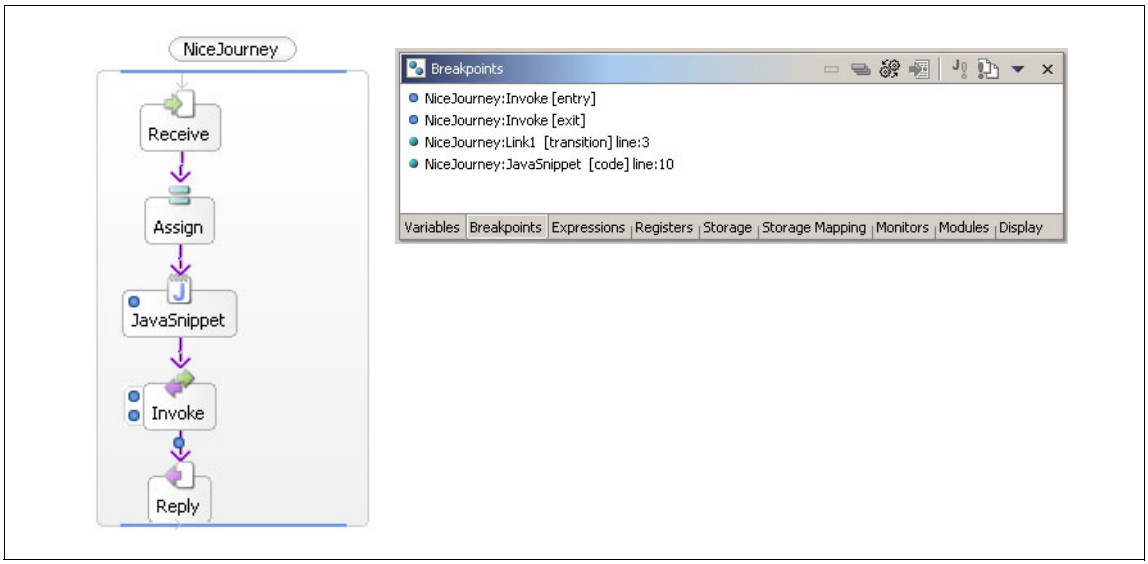


Figure 7-53 Breakpoints set to NiceJourney process, shown in BPEL Editor and in Breakpoint view

All we have to do now to debug a process in the WebSphere Test Environment is to start NiceJourneyServer in Debug mode.

1. If the NiceJourneyServer is still running, stop it.
2. Start the NiceJourneyServer in Debug mode. Right-click it and select **Debug** from the pop-up menu.
3. If the following messages appears, click **Yes**.

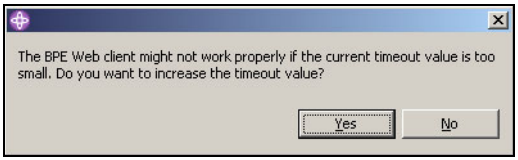


Figure 7-54 Message that occurs when debugging is done first time in workspace

4. Watch the Servers view (either in Debug Perspective or in Business Integration Perspective) and wait until the NiceJourneyServer has started (the Status column will show `Started in debug mode`).

5. Right-click **NiceJourneyServer** and select **Launch Business Process Web Client**.
6. Use the NiceJourney Template to create a process instance; this is shown in “Using the process Web client” on page 181.
7. When a new instance is created using the **Start Instance** button in the Process Web client, the Debug Perspective should come to the front; if not, switch manually to it by clicking **Window** → **Open Perspective** → **Debug**.
8. When the first breakpoint is reached in the process instance, the Debug view shows the activity (in our case the Java snippet) on which this breakpoint is set. Switch to the Debug view and scroll down to the Java snippet.
9. Switch to the Variable view to see the current content.
10. Click the **Step Into** button of the Debug view’s toolbar to step into the Java snippet.
11. Click the **Step Over** button of the Debug view’s toolbar to step over some lines of code in the Java snippet.
12. Click the **Step Return** button of the Debug view’s toolbar to step to the return of the Java snippet.
13. Use the **Resume** button of the Debug view’s toolbar to get to the next breakpoint, the Invoke activity.

After resizing and closing some views, your window should be similar to the following figure.

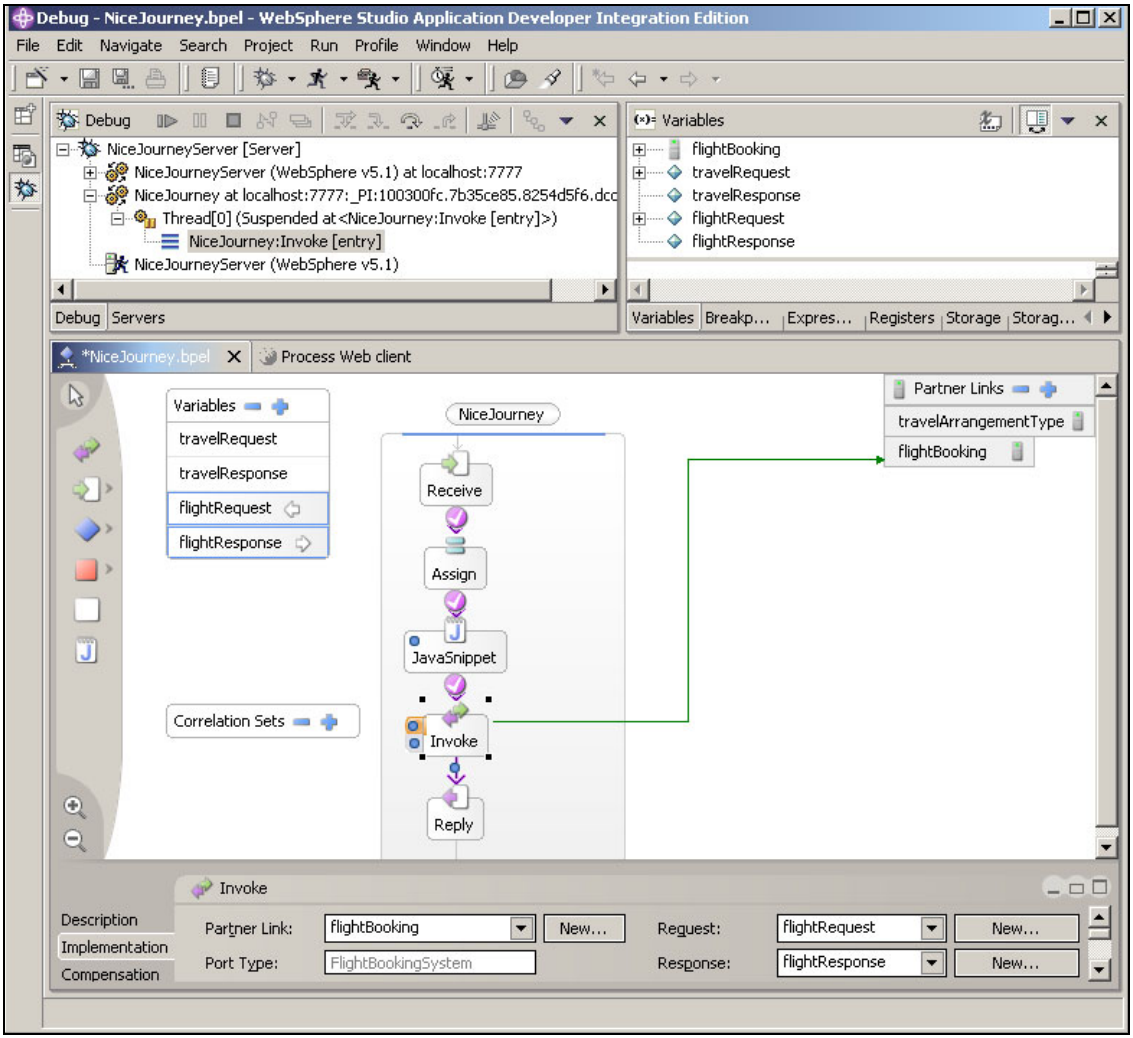


Figure 7-55 Debugging a process instance in the Debug Perspective

While stepping through the process instance, the current position of the flow is indicated in the BPEL Editor with the symbol . This symbol is also used to indicate that the process is paused at a breakpoint that is installed and enabled.

The symbol indicates that the link on which the symbol resides was followed, since the transition condition evaluated as `true`.

Note: You may get errors on the Console view during debugging. These errors do not show when you run the test in normal mode.

The reason is that the debugger tries to evaluate every variable and partner link at the breakpoints, even before they are available.

Refer to the WebSphere Studio Application Developer Integration Edition help to read about several other symbols that are used to indicate the status of a breakpoint and whether a link was followed.

7.2.10 Deploying a process to WebSphere Business Integration Server Foundation

For the deployment task on the WebSphere Business Integration Server Foundation, we will need the deployable package of the application.

1. Start the WebSphere Business Integration Server Foundation application server and bring up the Administrative Console (`http://<server_name>:9090/admin`).
2. Navigate to **Servers** → **Application Servers**, then select **server1**.
3. Select **Business Process Container** at the end of the Additional Properties (right-hand pane).
4. Make a note of the data source. This is the JNDI name for the data source that the Business Process Container is using. The default name is jdbc/BPEDB, if you are using the default container.
5. Navigate to **Applications** → **Install New Application**.
6. Select **Browse** then select the **WPC_Simple_Process.ear**; this is the enterprise application that holds the processes. Click **Next**.
7. Step through the installation steps of the process application, as if it were a normal enterprise archive (EAR).
8. When you get to the step where you must provide a default data source mapping for modules containing 2.0 entity beans, you will have to make some changes.

   Important: The Resource JNDI name and the Business Process Container Resource JNDI name *must* be the same.

   a. Open the Apply Multiple Mappings section with the + (plus) symbol next to it.

b. Select the correct Resource JNDI name from the drop-down list. There are multiple items on the list; the one you need has a name very similar to the data source JNDI name. It should have the eis/ prefix and the _CMP extension, so the full name should look like: eis/*datasourceJNDI*_CMP. The resource JNDI name for the default data source is eis/jdbc/BPEDB_CMP.

   In a Network Deployment environment, the JNDI name looks like eis/*datasourceJNDI_clustername*_CMP.

c. Check the box next to the EJB module, where the processes are stored.

d. Click the **Apply** button below `Specify existing Resource JNDI name`.

9. When you deploy an interruptible process, you get the following question (see Figure 7-56). Check the **Enable** box, then click **Next**.

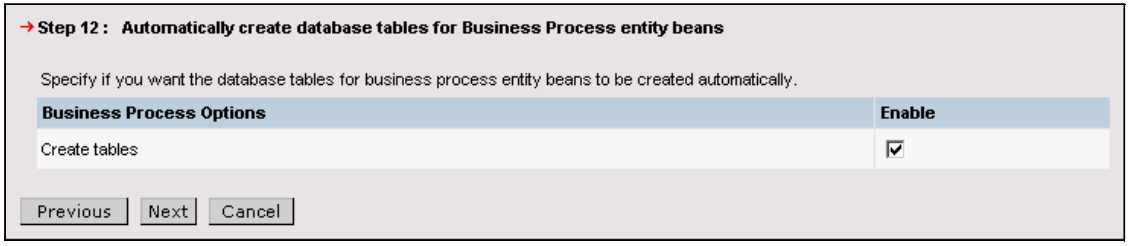


Figure 7-56 Automatically generate database tables

> **Note:** Interruptible processes need a database for persistence. The process state is persisted in a database by entity beans (EJB).
>
> By selecting this option, the necessary tables will be generated during deployment in the database configured for the business process container.

10. At the end of the deployment steps, click **Finish**.
11. Save the configuration for the application server.

Once the process has been deployed, you can start the application.

1. Navigate to **Applications** → **Enterprise Applications**.
2. Check the box next to the recently installed application, the one that includes the business processes; then click **Start** at the top of the page.

   Wait until the application starts; you should get a message at the top of the page, stating that the application was successfully started.
3. Log out from the Administrative Console.

You can run a quick test to make sure that the process has been deployed and started.

1. Open a Web browser at the `http://<server_name>:9080/bpe` location.
2. Select a template, one that you have just installed with the application.
3. Run the process to see if it works.

For more information, refer to the WebSphere Business Integration Server Foundation InfoCenter at `http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp`, then navigate to **WebSphere Business Integration Server Foundation** → **Deploying** → **Applications** → **Process Choreographer**; you can find more details under this node in several documents.

7.2.11 Debugging a process on WebSphere Business Integration Server Foundation

To debug a process instance on WebSphere Business Integration Server Foundation, the Process Debugger can be attached to the Process Engine; see also “Process debugging components” on page 183.

It does not matter whether or not both components reside on the same machine or on different machines. In this chapter, we will show how to attach a Process Debugger component of WebSphere Studio Application Developer Integration Edition to a Process Engine, component of WebSphere Business Integration Server Foundation residing on a different machine.

For this chapter, we assume that we have a BPEL process with defined breakpoints (refer to 7.2.9, “Debugging a process in WebSphere Test Environment” on page 183), and that the EAR file of this process is already deployed to the WebSphere Business Integration Server Foundation (refer to 7.2.8, “Deploying and testing a process in the IBM WebSphere Test Environment” on page 177).

Preparing for remote debugging

Before we can debug a process instance on WebSphere Business Integration Server Foundation, we have to prepare in the following way.

1. Prepare WebSphere Studio Application Developer Integration Edition

   To attach the Process Engine, we do not need a server configuration in WebSphere Studio Application Developer Integration Edition, but we have to enable the server targeting support.

   a. Select **Window** → **Preferences**.

b. In the navigation tree on the left, select **J2EE**.

c. In the pane on the right, select the radio button **Enable server targeting support**.

d. Click **Apply**.

Since debugging a process instance on a remote server takes time, the Debugger time-out and the Launch time-out should be increased appropriately.

e. Expand **Java** in the navigation tree of the still open preference dialog.

f. Select **Debug**.

g. In the pane on the right, change the Debugger time-out to `300000` and the Launch time-out to `200000`.

h. In the preference dialog, click **OK**.

2. Prepare WebSphere Business Integration Server Foundation.

To debug a business process on WebSphere Business Integration Server Foundation, the Debug mode must be enabled. The default port for debugging purposes is 7777. To speed up debugging, the JVM debug argument `-Xj9` should be used. Since debugging may take extra time, for looking at variables, it is necessary to disable the transaction time-out, which is done by setting a value of `0` for the client and server. All of this can be done by performing the following steps.

a. Start the application server.

b. Open the Administrative Console of the WebSphere Business Integration Server Foundation on which the deployed EAR file of our business process resides and which you are going to debug.

c. Navigate to **Servers** → **Application Servers**, then select **server1**.

d. Select **Debugging Service** in the upper half of the Additional Properties (right-hand pane).

e. Check the **Startup** check box and verify that the JVM debug port is set to `7777`.

f. Add the argument `-Xj9` to the JVM debug arguments.

g. Click **OK**.

h. Back in the Additional Properties, select **Transaction Service**.

i. Change the Total transaction lifetime time-out to `0` (zero).

j. Change the Client inactivity time-out to `0` (zero).

k. Click **OK**.

l. Log out of the Administrative Console.

m. Restart the application server.

Now we are ready to debug a process instance remotely.

Debugging a remote process instance

From the WebSphere Studio Application Developer Integration Edition where the process with defined breakpoints resides, it is now possible to debug a remote process instance on WebSphere Business Integration Server Foundation. To do this, perform the following steps.

1. Switch to the Debug perspective.
2. Select **Run** → **Debug...**.
3. In the navigation tree of the just opened Debug dialog, select **WebSphere Application Server Debug** and click **New.**
4. In the Name field (right hand side), type `RemoteDebugNiceJourney`.
5. Click the **Browse** button and select the service project **WPC_Simple_Process**.
6. In the input field *Host name*, type in the Name or IP of the host where the WebSphere Business Integration Server Foundation is running in Debug mode.
7. Verify that the JVM Debug Port is `7777`.
8. Click **Apply**, then click **Debug.**

The Process Debugger will attach to the Process Engine, waiting for a process instance to debug. To generate such a process instance, open an external Web browser and type in the following URL:

```
http://<IP_of_RemoteHost>:9080/bpe/webclient
```

where `<IP_of_RemoteHost>` is the IP or name of your remote host.

From here on out, you can perform the same steps as those described in 7.2.9, “Debugging a process in WebSphere Test Environment” on page 183 to debug a local process instance.

Finishing remote debugging

Since only one Process Debugger can be attached at the same time to the Process Engine, this should be detached once debugging is complete. Therefore, all debugged process instances should be terminated, whether by finishing them normally, or by clicking the **Terminate** button of the Debug view. To detach the Process Debugger from the Process Engine, switch to the Debug view and right-click the connection instance **j9[<IPofRemoteHost>:7777]**. In the pop-up menu, select **Disconnect**.

7.2.12 Process versioning

WebSphere Process Choreographer allows for multiple versions of your business process to exist in your application server. You may install a newer version of your process while preserving any running instances of the previous template.

Versions of your process are determined by the `validFrom` timestamp in the BPEL code, for example:

```
wpc:validFrom="2003-01-01T00:00:00"
```

This can be specified in WebSphere Studio Application Developer Integration Edition on the Server tab of the process properties, as shown in Figure 7-57.

Figure 7-57 Valid From fields in the BPEL Editor

If multiple versions of a process are installed and running, starting an instance of that process will always result in the latest valid template being executed.

In order for multiple versions of a process to exist on your application server, they must occupy separate install roots on the server. To accomplish this, you will need to ensure that your process retains the same name across versions, but has a different enterprise application name for each. There are a few ways that the application name can be specified:

- In WebSphere Studio Application Developer Integration Edition, your enterprise application name is set to the name of the Service Project where your business process exists.
- In your project's application.xml file, you may change the <display-name> field, which defines your application's name.
- Upon installing the application to your server, you can specify the application name. This is under Step 1 of the application installation procedure if you are using the Administrative Console, or the appname parameter if you are using the wsadmin script client.

7.2.13 Uninstalling deployed processes

As you have already seen, business processes (BPEL4WS) are deployed as simple enterprise applications (EAR). The process itself is part of the EJB module, therefore it is managed under the EJB module.

There are differences between interruptible and non-interruptible processes from the management point of view. Interruptible processes leave footprints as they run and once they finish. Each time a process starts, a new instance is created from the process template. The instances are persisted in a database configured for the business process engine. A process cannot be removed until instances of the process appear in the database, be they running or finished instances.

Before you can remove a business process from the business process container, you have to:

1. Finish or terminate the running processes.
2. Delete the finished processes.

Once all the instances are removed, you can start removing the process application. This entails two separate steps:

1. Stop the running process template.
2. Uninstall the process application.

There could be problems during the process described above. If a process application cannot be uninstalled, you can be sure that there are instances of the process running somewhere.

Terminating a process

Terminating a process works as a hard stop for the process. When terminating the process, the instance stops running and never returns to the running state. You can consider it a lost instance.

Terminating a process is not recommended under any circumstances. The process does not perform any compensation or any rollback when terminated. A terminated process can have a harmful impact on the business. The recommended way of stopping a process is to finish the process as expected. Administrators can manage the process instances and perform the necessary activities to stop the processes.

If you really need to terminate a process, follow the steps below:

1. If you are running the process in a Network Deployment environment, make sure that all the application servers are running.

2. Start the business process Web client and open a Web browser with the following URL: `http://<server_name>/bpe` or `http://<application_server_name>:9080/bpe` (to access the application server directly).
3. Select **Process Instance Lists** → **Administered By Me**.
4. Check the box next to the process(es) you want to terminate.

> **Note:** You can terminate more than one process at a time.
>
> You can select all the processes at the same time by checking the box at the head of the process list.

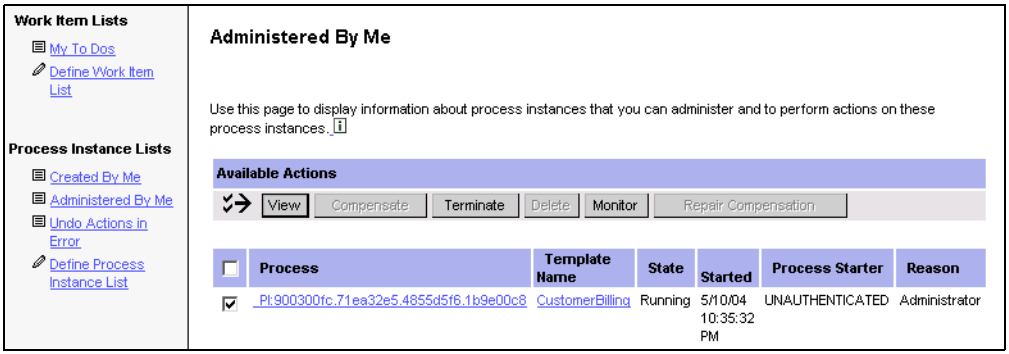


Figure 7-58 Terminating a process

5. Click **Terminate**.
6. Once the process is terminated, the state changes to `Terminated`.

Deleting a process instance

Before you can remove the business process applications, you have to remove all existing process instances.

1. Start the business process Web client if it is not running.
2. Select **Process Instance Lists** → **Administered By Me**.
3. Check the box next to the process(es) you want to delete; you may want to select all of them by checking the box in the header.

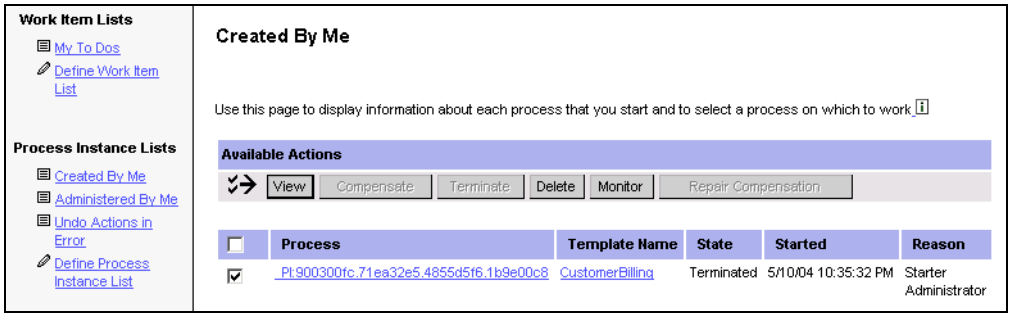


Figure 7-59 Deleting the process

4. Click **Delete** to delete the process(es).

   The process(es) should disappear from the list.

5. You can close the business process Web client.

Stopping the process template

Before you can remove the business process applications, you have to stop the process template(s).

1. If you are running the process in a Network Deployment environment, make sure that all application servers are running.
2. Open the Administrative Console in a Web browser at `http://<application_server_name>:9090/admin`.
3. Navigate to **Applications** → **Enterprise Applications**.
4. Select the application link that runs the business process.
5. Select **EJB Modules** at the bottom of the page, under Related Items.
6. Select the JAR file that holds the business process.
7. Select **Business Processes** at the very bottom of the page.
8. Check the box at the head of the business process templates list. It will select all the processes; then click **Stop**. It stops all the business process templates for the application.

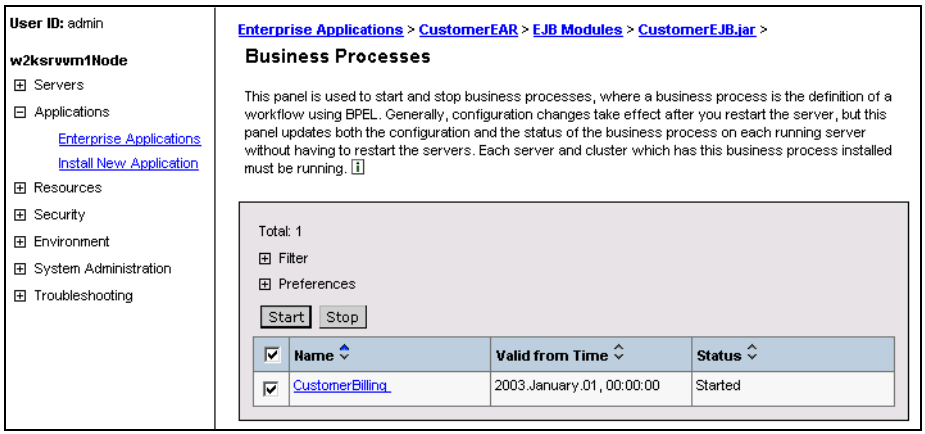


Figure 7-60 Stopping a Business Process

9. Save the configuration for WebSphere.

Uninstalling the business process application

Uninstalling a business process application is the same as uninstalling any other EAR file.

1. Check the box next to the enterprise application.
2. Click **Stop** to stop the running application.

> **Important:** This is a critical step; you will only be able to uninstall the application if it is stopped. It will not stop unless all the process instances are removed.

3. Once the application is stopped, check the box next to the enterprise application you want to remove, then click **Uninstall**.
4. When the application is uninstalled, save the configuration for WebSphere.
5. Log off of the Administrative Console.

At this point, you are ready to redeploy your new or revised business process application.

8

Chapter 8.

Process choreographer: developing a complex process

In this chapter, we show how you can create a process that utilizes more of the advanced capabilities provided by WebSphere Process Choreographer. We use a travel agency scenario and gradually extend the capability of our travel agent by making use of additional functionality in our process. You will see how to model complex interactions with partners, how to implement advanced sequencing logic, detailed fault handling and more. Each section can be read individually in order to understand how to use a specific capability, or the chapter can be read as a whole to understand how to model and build a complex process. As a prerequisite, it is recommended that you read Chapter 7, “Process choreographer: developing a simple process” on page 135, which demonstrates how to build a simple process. This will familiarize you with the development environment; this knowledge is assumed throughout this chapter.

8.1 Introduction

This chapter demonstrates the more advanced features of WebSphere Process Choreographer which you did not see in the previous chapter when creating a basic process. WebSphere Process Choreographer is capable of modeling and implementing complex business processes that contain features such as:

- Alternate execution paths based upon business factors
- Human Interaction
- Advanced partner interactions, both synchronous and asynchronous
- Error handling
- Complex sequencing
- Interruptible processes potentially lasting for days, weeks or months

We show you how WebSphere Process Choreographer can be used to achieve all of these advanced behaviors.

Our chosen scenario is that of a travel agency which is responsible for taking a customer's travel request and making the relevant arrangements. We begin with a basic outline of the business model so that you can become familiar with the overall requirements for the example business process. Each section of the chapter then adds detail to a specific part of the process to extend the functionality and to improve its robustness. Where possible, sections are created separately in order to avoid prerequisites between them. This will allow you to look at individual sections or combine them to gain more of the full process function.

The overall intention of the business process is to model a fictional travel agency's booking process, which we call the NiceJourney process. The basic capability of the process is shown in Figure 8-1, which illustrates the functionality we want to achieve but includes none of the implementation details at this point.

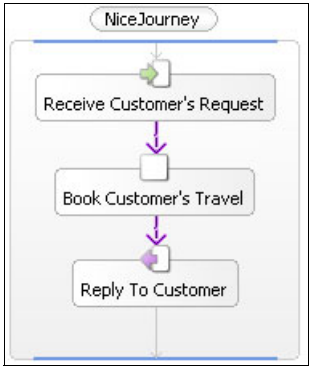


Figure 8-1 Basic function of the NiceJourney process

The process must receive some input from the customer, book some travel arrangements for them and then reply back to the customer. In this outline, the Book Customer's Travel activity is an Empty type activity; this is a useful way of putting an activity into a process without knowing what its implementation will be. By the end of this chapter, we will have replaced the empty Book Customer's Travel with many activities that will perform the required actions for booking the customer's travel.

The basic outline shown in Figure 8-2 provides an additional level of detail about the process that we will create. It shows how the booking breaks down into booking three separate reservations that together comprise a customer's trip - a car, a flight and a hotel. We will be implementing each part using WebSphere Process Choreographer activities in our process.

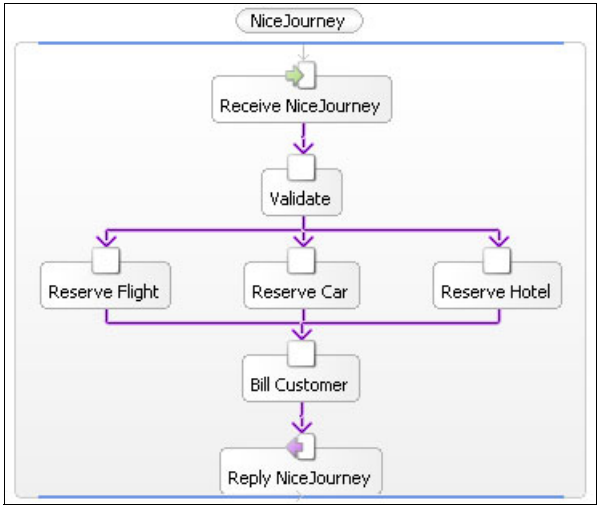


Figure 8-2 Basic Outline of the NiceJourney process implementation

8.2 Preparation

Before beginning to implement the individual parts of the complex process, you need to perform some basic steps to prepare the workspace. These will create the initial outline of the process and any one-time steps that are required to configure the execution of the NiceJourney process.

If you want to skip the step-by-step instructions in this section, you can simply import the completed outcome by using the Project Interchange feature described in "Project Interchange archive import/export" on page 562. The steps to import the solution are described in the following section.

Note that if you do not import the completed preparation project, however, you will need to follow *each* of the manual steps.

8.2.1 Importing the prepared NiceJourney

The following instructions will enable you to create a workspace with the basic outline of NiceJourney already completed.

1. Ensure that you have installed the Project Interchange plug-in as described in “Project Interchange archive import/export” on page 562.
2. Start WebSphere Studio Application Developer Integration Edition with a new workspace location.
3. Choose **File** → **Import...** and select **Project Interchange** from the menu system. If you do not see this option then the Project Interchange feature needs to be installed.
4. Browse to the additional directory and select the **ComplexNiceJourneyPreparationSolution.ic.zip** file from the ComplexProcess subdirectory.
5. Select the **NiceJourney** box to import this project into your workspace.
6. Click **Finish**.

8.2.2 Creating the prepared NiceJourney step-by-step

If you want to see the steps required to create the basic outline of the NiceJourney process then follow the procedure described in this section.

Creating the NiceJourney business process

1. Create a new service project called **NiceJourney**.
2. Create a new flow-based BPEL business process called *NiceJourney* in the package com.nicejourney.
3. Open the NiceJourney.bpel editor if it is not already open and find the top point of the process, labeled NiceJourney. Click this and then open the server properties at the bottom of the BPEL Editor.

   This is where you set properties relating to the process itself as opposed to individual activities.
4. Check the **Process is long running** option.

Configuring namespaces

WSDL and BPEL are both XML-based description languages. XML uses a concept called *namespaces* to give a degree of organization to the many tags that are used. A detailed understanding of XML namespaces is not a requirement for using WebSphere Process Choreographer or WebSphere Studio Application Developer Integration Edition.

Note: You can think of namespaces as a folder structure with each namespace value being a different folder. Any given XML tag used in the BPEL and WSDL documents can come from one of the namespaces. Because a business process will likely consume many services, there will be many XML documents and tags that are referenced directly or indirectly by the process. The namespace structure helps simplify this. It is similar to Java packaging.

One further point to be aware of is that although a namespace value looks like a URL, it is not actually intended to point to a real location and is just a unique identifier string.

The default namespaces are generated names of the form http://www.example.com/processXXXXXX. We want to edit these for our process because it is good practice to use your own naming systems. In some cases, the automatically created names can be used for speed and convenience.

1. Open the NiceJourney.bpel editor if it is not already open and find the top point of the process, labeled `NiceJourney`. Click this and then open the Server properties at the bottom of the BPEL Editor.

   This is where you set properties relating to the process itself as opposed to individual activities.

2. Change the target namespace to `http://nicejourney.com/NiceJourney`. Save and close the file.

3. Switch to the Package Explorer view and open NiceJourney.wsdl from the NiceJourney/com/nicejourney folder. This will open the file in the WSDL editor. Switch to the Graph view, rather than the Source code view, if it did not open automatically.

4. At the bottom left of the editor is the Definitions section. Click the **Edit Namespaces...** button.

Tip: If you do not see the Edit Namespaces... button then you may have selected one of the imports, types, services, bindings, port types or messages. You may have to click an area of white space in the top section to deselect all of these, at which point the Edit Namespaces... button should appear.

5. Change the Target Namespace value to `http://nicejourney.com/NiceJourney` and click **OK**.
6. In the top half of the editor, right-click the import for NiceJourneyInterface.wsdl and click **Delete**. This removes the import for the default namespace. This import will be correctly regenerated later with the updated namespace when we update the process partner link.
7. Save and close the file. Ignore the errors that may appear in the Tasks view at this point.
8. Now open the NiceJourneyInterface.wsdl file and click the **Edit Namespaces...** button. Change the Target Namespace value, this time using `http://nicejourney.com/NiceJourney/interface` to indicate that this is the namespace for the process *interface*.
9. Optionally, delete the NiceJourney/com/example folder and its contents from the Navigator view in the Resource perspective (this view allows you to work with the underlying file system directly). This will remove the remaining unwanted entries, produced by default from the old namespace values that were set by creating a new business process.

Defining the NiceJourney Process Interface

Now we will define the interface for our process by specifying the operations and messages that it will use. We will modify the default operation and messages to create the interface we want before building the process to implement this interface. This is a top-down development style where the interface is defined before the implementation is done.

1. If it is not already open, open NiceJourneyInterface.wsdl in the WSDL editor.
2. Rename the default port type from `ProcessPortType` to `NiceJourney`.
3. Rename the default operation from `InputOperation` to `NiceJourneyOperation`.
4. Rename the default message from `InputMessage` to `NiceJourneyInput`.
5. Right-click the single part that belongs to the NiceJourneyInput message and choose **Delete** to remove it.
6. Right-click the **NiceJourneyInput** message and select **Add Child** → **part.** Name the part `firstName` and click **OK**.

7. Add the following additional parts to NiceJourneyInput in the same way:
   - lastName
   - address
   - city
   - zipcode
   - state
   - cityFrom
   - cityTo
   - dateTimeDeparture
   - dateTimeReturn
   - cardType
   - cardNumber
8. Select the **zipcode** part and change its type to `xsd:int`.
9. Create a new message by right-clicking in the messages section of the editor and selecting **Add Child** → **message** from the pop-up menu. Name the message `NiceJourneyOutput` and click **OK**.
10. Add a part to the message called `reservationID` and set its type to `xsd:long`.
11. Expand **NiceJourneyOperation** and click the **input** node. Change the value of the associated message from `tns:InputMessage` to `tns:NiceJourneyInput`.
12. Do the same for the output node, this time choosing `tns:NiceJourneyOutput`.
13. Right-click the **NiceJourneyOperation** and choose **Add Child** → **Fault**. Name the fault `NiceJourneyFault`.
14. Right-click **NiceJourneyFault** and select **Set Message...**.
15. Choose to **Create a new message** and name it `NiceJourneyException`.
16. Right-click the **NiceJourneyException** message and select **Add Child** → **part.** Add the following parts, each of type `xsd:string`:
    - firstName
    - lastName
    - reason

    The completed definition of the interface is shown in Figure 8-3 on page 210.

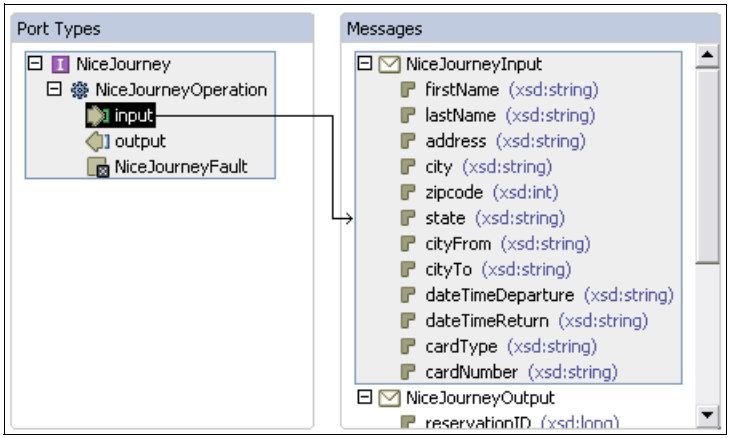


Figure 8-3 Defining the NiceJourney interface in NiceJourneyInteface.wsdl

17. Save and close the file.

Creating the process outline

1. Return to the NiceJourney.bpel editor, select the partner link **PartnerLink** and rename it to `NiceJourney`.
2. Open the implementation properties for the partner link and click **New** to create a new partner link type called `NiceJourneyPLT`. Make sure that the file is set to `NiceJourneyInterface.wsdl` and that there is only one role.

   Click **Browse...** to locate the Port Type File and locate *NiceJourney/com/nicejourney/NiceJourneyInterface.wsdl*. Choose the **NiceJourney** port type. This is the port type we just defined when creating our interface. Click **OK** to accept this.
3. The completed Partner Link Type dialog should look as shown in Figure 8-4 on page 211. Click **OK**.

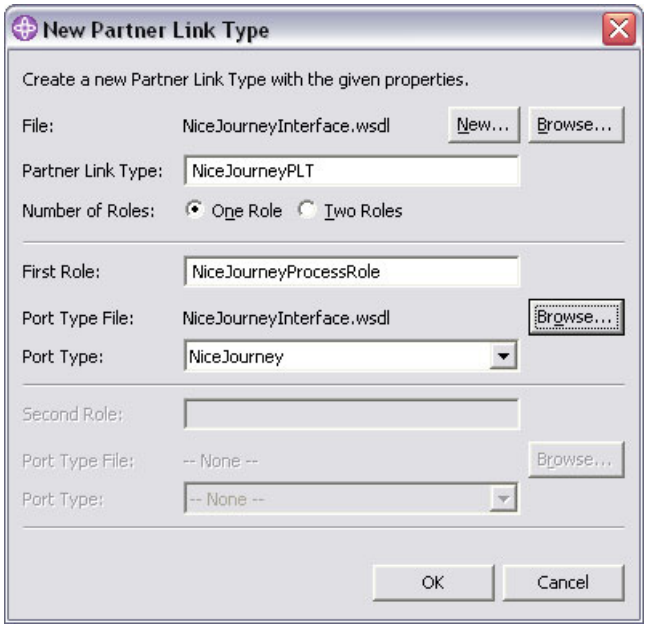


Figure 8-4 Completed Partner Link Type

4. The partner link type we just created contains the role that our process interface is to fulfill. On the NiceJourney partner link implementation properties, click the **<-->** arrow to switch the NiceJourneyProcessRole to the Process Role Name to the right of the editor. This indicates that the partner link role is being played by the process, not by a partner that the process will invoke.
5. Delete the default process variable named **InputVariable**.
6. Select the **Receive** activity, rename it to `Receive NiceJourney` and change the implementation settings to:
   - Partner Link: `NiceJourney`
   - Operation: `NiceJourneyOperation`
7. Click **New...** to the right of Request to create a new variable and name it `NiceJourneyInput`. Because you set the Partner Link and Operation already, this variable will be created with the correct message type automatically.
8. Select the **Reply** activity, rename it to `Reply NiceJourney` and change the implementation settings to:
   - Partner Link: `NiceJourney`
   - Operation: `NiceJourneyOperation`

9. Click **New...** to the right of Response to create a new variable and name it `NiceJourneyOutput`. Because you set the Partner Link and Operation already, this variable will be created with the correct message type automatically.
10. Save the NiceJourney.bpel file. There should be no errors in the Tasks view and one warning that the deployment code must be generated.
11. Optionally, delete the NiceJourney/com/example folder and its contents from the Navigator view in the Resource perspective (this view allows you to work with the underlying file system directly). This will remove the remaining unwanted entries that were created by default when you created a new business process.
12. Switch back to the Business Integration perspective.
13. Add empty activities and create control links so that your process is as shown in Figure 8-2 on page 205. Be sure to rename all the activities using the same names we used.

> **Tip:** You may find it easier to arrange items on the process if you make use of the auto-arrange functionality. Right-click the process or the flow and select whether to arrange or align the process or flow. Selecting the automatic option causes the editor to re-arrange the flow as you work.

14. Save the NiceJourney.bpel process and close the editor.

At this point, the process skeleton is in place and is ready for you to begin implementing one of its pieces.

8.3 Validation implementation

This part of the process is used to validate the input data that initiated the process. We will check the content of some of the data and terminate the process if any of the values are invalid.

We could create a more advanced validation that would look at the supplied values and reply back to the calling partner requesting additional input instead of terminating. However, this would depend upon the intelligence of the client to receive our response asking for additional information. The client to our process could easily be another computer system rather than a person operator. In many cases, a process will be invoked automatically from elsewhere and we will not want to reply when the data is invalid.

This also gives us an ideal opportunity to demonstrate one of the simplest activity types, the Terminate activity. This is used to end a process abruptly when a

failure scenario has occurred. In our case, the validation will determine that we do not have the necessary information to complete the process and therefore we will terminate.

Our implementation demonstrates the following activity types:

- Sequence
- Assign
- Invoke
- Java snippet
- Terminate

In addition, it shows the following features of BPEL processes:

- Synchronous Invocation of a Java service
- Fault Handlers

We will start with the outline process shown in Figure 8-2 on page 205 and then replace the empty Validate activity with the set of activities shown in Figure 8-5.

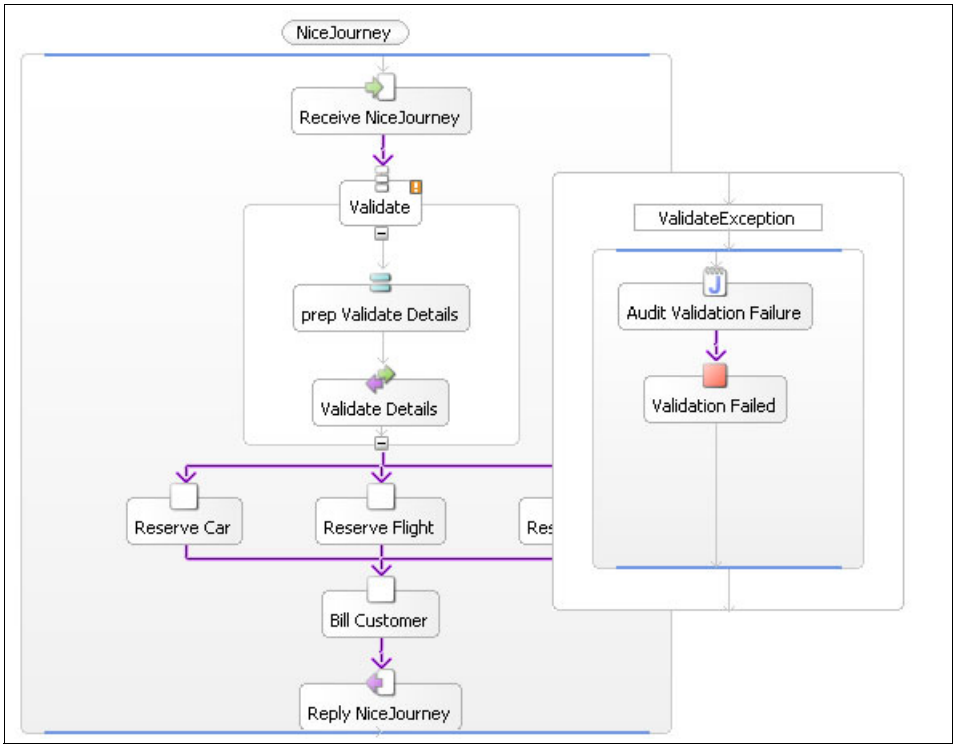


Figure 8-5 Completed Validation section of the NiceJourney process

A completed solution for this section can also be imported into a fresh workspace by using the Project Interchange Zip format to import *ComplexNiceJourneyProcessValidationSolution.ic.zip*. See “Project Interchange archive import/export” on page 562 for details about how to import this zip file.

8.3.1 Preparation

We show how to implement this part of the process in isolation. Therefore, you do not need to have completed any of the other implementation sections to be able to follow these steps.

However, as a minimum requirement, you do need to have the workspace prepared with the basic outline of the process. You can do this by either following the steps described in 8.2, “Preparation” on page 205 or by building upon your solution from another section. If you follow the instructions in 8.2, “Preparation” on page 205, you can choose to import a prepared solution and begin working on this section immediately.

If you intend to build upon another section then you can continue to use the same workspace that you used then. In this case, we suggest that you back up your projects at this point by exporting the NiceJourney project as a Project Interchange Zip file. For more information about Project Interchange Zips, see “Project Interchange archive import/export” on page 562.

Once you have either prepared the workspace with the basic outline or taken your own workspace from the previous section, you are ready to complete the Validation implementation.

8.3.2 Sequence activity

The validation will be implemented by multiple steps to be performed sequentially. We will not need any conditional flow logic to model this so we will use a sequence activity to contain this set of steps. Although we could continue to use a flow activity, with a simple control link between each step, a sequence is simpler to work with in the editor because the flow logic is provided for us by the structured activity. We start the implementation of the validation part of the process by creating this sequence:

1. Open the NiceJourney.bpel process in the editor.
2. Right-click the **Validate** activity and select **Change Type** → **Sequence**.

This will create a sequence within which we will place the necessary activities to complete the validation.

8.3.3 Invoke - Java Class synchronous invocation

Invoke activities are used to invoke some kind of service that a process needs to fulfill its requirements. The invoked service can be implemented in many different ways and the process does not need to know the details of this. As long as the service can be described using WSDL then we can invoke it from our process. Section 7.2.6, “Preparing to develop the process” on page 153showed how to use WebSphere Studio Application Developer Integration Edition to automatically create a WSDL-described service from a Java program.

In this section, we used a slightly different technique for invoking a service that is implemented as a Java program. In the previous chapter, a service interface was generated for the Java program and then this service was added to the process. This time, we will add the Java program directly to the process.

Note: When you add a Java class to the process editor directly, WebSphere Studio Application Developer Integration Edition will automatically create the service interface. This makes it very similar to creating the service interface yourself manually and then adding the service to the process.

This can be useful when it does not make sense to expose the Java code as a reusable service for other clients, perhaps because it is only used by other Java applications running locally and does not need a full service interface.

The following steps describe how to create and configure the Invoke activity.

1. Add an Invoke activity inside the *Validate* sequence and rename it to `Invoke Validation Partner`.

   This activity will call the external partner to our process that will perform the validation for us. The definition of *external* can mean anything from another piece of Java code executing within the same server environment as our process, to a service implemented outside our fictional NiceJourney company.

2. Create a new Service project called `ValidatorPartner` that will contain the validation implementation.

3. Import Validator.java and ValidatorException.java from the *ComplexProcess* directory from the additional material into the folder ValidatorPartner/com/nicejourney/partners.

   These are the Java classes that implement the Validation service. These could just as well be any implementation that is callable as a service (describable using WSDL).

4. Drag and drop the Validator.java file from the services view onto the NiceJourney.bpel editor canvas.

Notice that a special type of partner link is created with the Java symbol used as the icon. This indicates that this is a partner implemented as a Java program. Select this partner link and note that the implementation is the Java class, in this case Validator. Locate the generated WSDL definition of the partner in NiceJourney/com/nicejourney/comnicejourneypartnersValidatorNiceJourneybpel.wsdl.

5. Select the **Invoke Validation Partner** activity and go to the implementation editor. Set the Partner Link to `Validator` and the operation to `validate`.
6. Create new Request and Response variables using the **New...** buttons to ensure that the variable types are set automatically. Name them `ValidateDetailsRequest` and `ValidateDetailsResponse` respectively.

Note: Although the validate method of the Java class is defined as returning void, we found that it was necessary to supply a response variable.

When the Validator.java file was dropped onto the canvas, a new service definition of the validate method was automatically created. The WSDL file that describes this is found in the NiceJourney/com/nicejourney package and is called *comnicejourneypartnersValidatorNiceJourneybpel.wsdl*.

Exploring this file will reveal that the generated operation is defined as having both input and output. For this reason, we must supply a variable for both request and response when configuring the Invoke activity. We will not use the empty response variable.

The Invoke Validation Partner activity has now been successfully configured to call the validate method of the Validator class, using the process variables ValidateDetailsRequest and ValidateDetailsResponse.

8.3.4 Assign

Refer to 7.2.4, “Assign activity” on page 148 for additional instructions on how to use the Assign activity.

We need to use an Assign activity to set the values on the ValidateDetailsRequest variable before we invoke the Validator Java service. This assign will take the relevant values that were passed in to the process in the process request variable NiceJourneyInput. The following steps show how to copy these values from NiceJourneyInput to ValidateDetailsRequest.

1. Add an Assign activity inside the Validate sequence, before the Validate Details Invoke activity. Rename it to `prep Validate Details`.

2. Open the implementation for the Assign and configure copies between the following parts, using NiceJourneyInput in all cases as the From variable and ValidateDetailsRequest as the To variable for each copy. See Figure 8-6 for an example.

   - `firstName` to `firstName`
   - `lastName` to `lastName`
   - `address` to `address`
   - `city` to `city`
   - `zipcode` to `zipcode`
   - `state` to `state`
   - `cityFrom` to `from`
   - `cityTo` to `to`
   - `dateTimeDeparture` to `departDate`
   - `dateTimeReturn` to `returnDate`

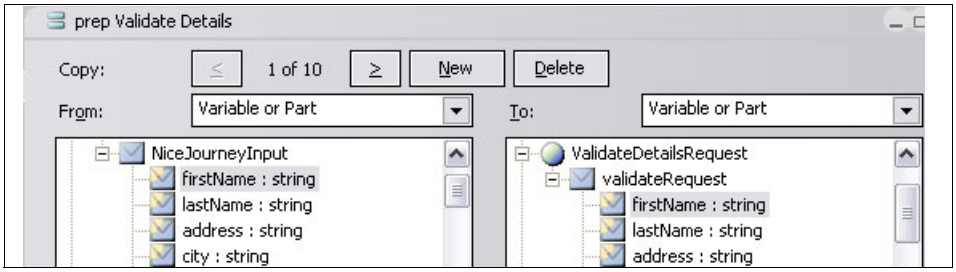


Figure 8-6 prep Validate Details Assign activity

8.3.5 Fault Handler

The Validate Details service that we invoke will fail when the required details are not provided (if any of the values is an empty string, or if the zip code is negative). In these cases, an exception will be thrown. We want to capture that error in our business process and terminate the process.

We can see that an error can occur by either inspecting the implementation class, or better still, by looking at the generated service interface for it. The implementation of the com.nicejourney.partners.Validator class shows that it throws a ValidateException if invalid data is passed to it. You can see this if you look at the imported class.

The generated service definition of this Java class is in *NiceJourney/com/nicejourney/comnicejourneypartnersValidatorNiceJourney bpel.wsdl* and this interface is shown in Figure 8-7 on page 218. You can see the ValidateException fault that is defined on the *validate* operation.

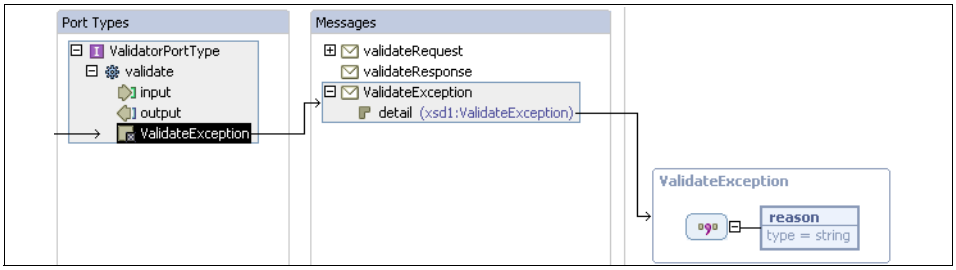


Figure 8-7 Generated service interface to the Validator class

This is the fault that we want to handle in our process. The following steps show how we add a Fault Handler that will catch this fault.

1. Open NiceJourney.bpel and click the icon for the sequence activity called **Validate**.
2. Right-click the activity and click **Add Fault Handler**.

> **Tip:** Instead of right-clicking, you can hover the pointer over the activity. Doing this with any activity causes options to appear in a speech bubble above the activity. With a sequence, one of the shortcut options is to add the Fault Handler, as shown in Figure 8-8.

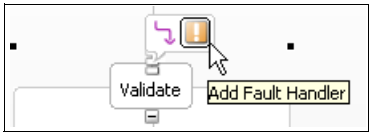


Figure 8-8 Adding Fault Handler graphically

At this point, there is a Fault Handler associated with this activity. You can see whether there is a Fault Handler associated with an activity by looking for the orange exclamation mark at the top right of the activity icon.

Fault handlers do not appear on the canvas by default. You have to choose to show the Fault Handler for any given activity. In addition, only one Fault Handler is shown at a time. Opening one will close all other Fault Handlers.

A Fault Handler can be added at any level of the process. For example, we could have specified a Fault Handler on the individual activity of the Invoke. Instead, we added it to the Validate sequence because logically it is associated with a failure at that level. You can add at whatever level is appropriate to your design. When an exception occurs, the runtime will search for a Fault Handler on the failing activity. If none is found then it will look for a Fault Handler at the next containing level. If none is found here then

the next level is tried. This continues until the final level of the process. If no Fault Handler exists on the process as a whole then the process will end in failure.

It is important to try and keep Fault Handlers free from implementation details. At the business process level, failures should be modeled as `Reservation failure` rather than `Failed to get DB2 database connection`. A technical failure should ideally be wrappered into a business level failure. This means that a future change in database implementation, for example, would not impact the business process.

3. View the Fault Handler that was just created by right-clicking the activity and choosing **Show Fault Handler** from the pop-up menu. This will open a small box to the right of the sequence; this is the empty Fault Handler. As an alternative, you can also double-click the small orange exclamation mark in the top right corner of the activity.

   Next, we need to add a catch to the Fault Handler so that we can begin to use it to catch exception scenarios. It is possible to catch all faults or to catch specific types of faults. In this situation, we want to catch the type of fault that is thrown by the Validator service.

4. Click the Fault Handler and then hover the pointer over it until the speech bubble of options appears, then click the **Add Catch** icon, as shown in Figure 8-9. Alternatively, right-click the Fault Handler and select **Add Catch** from the pop-up menu.

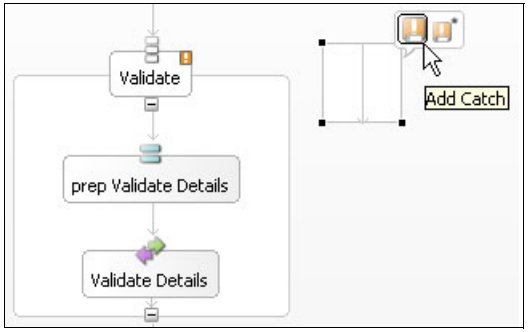


Figure 8-9 Adding a catch to a Fault Handler

5. Select the Catch and change the fault details to:
   - Fault type: `User-defined`

     Namespace:

     `http://nicejourney.com/comnicejourneypartnersValidatorNiceJourneybpel/`
   - Fault Name: `ValidateException`

Tip: The Namespace and Fault Name values here are important and must match those that define the service fault. When the Validator class was dropped onto the canvas, the comnicejourneypartnersValidatorNiceJourneybpel.wsdl file was created to describe the Validator class as a service. The namespace value specified in the fault here must match the namespace of that WSDL file. The Fault Name specified here must also match. You can check these values by looking at the WSDL file.

In general, a catch must specify the correct namespace and fault name in order to catch the fault. A common error is to specify these incorrectly, which will result in the catch not triggering. A catch all would normally trigger instead, if one was specified (we have not created a catch all for this Fault Handler).

6. Click **New...** to create a new Fault Variable and name it `ValidateException`.

   The variable is created with a default message type. We need to change this to indicate that it matches the type of the ValidateException fault, as defined in the WSDL definition of the Validator class.

7. Select the newly created **ValidateException** variable on the process editor and change the message properties by browsing to the file NiceJourney/com/nicejourney/comnicejourneypartnersValidatorNiceJourney bpel.wsdl and selecting the **validateException** message as shown in Figure 8-10.

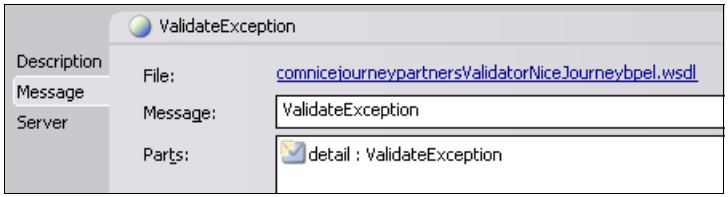


Figure 8-10 Specifying the message type for the ValidateException variable

8. Save the process file.

   This completes the creation of the Fault Handler. However, there are no actions inside the Fault Handler at the moment so we need to add some behavior.

8.3.6 Java snippet

We want to provide some basic auditing of processes that fail validation. In order to do this, we will make use of a Java snippet to simply write an informative

message to the server console. In a production application, you would likely have a much more sophisticated audit mechanism and not use the server standard output. However, this simple method demonstrates the type of behavior that may be modeled inside a Fault Handler.

We will use a Java snippet activity with some simple code to output the values of the data that was passed to the process. This Java snippet could be a place holder for a future upgrade where it would be replaced with a database write, for example.

1. Add a new Java snippet *inside the ValidateException Fault Handler* that you just created.
2. Rename it to `Audit Validation Failure`.
3. Add the following code for its implementation:

```
System.out.println("Validate failed because of :" +
getValidateException().getDetail().getReason());
```

> **Note:** This code first gets the ValidateException variable. It then gets the part of this exception object called Detail which is a complex type. Therefore, it is then necessary to get the Reason element of the complex type.
>
> You can work out the structure of the message by looking at the ValidateException message definition in the WSDL that was automatically generated to describe the Validator Java class. This file is: NiceJourney/com/nicejourney/comnicejourneypartnersValidatorNice Journeybpel.wsdl.
>
> You can also compare the message type with the Java equivalent by opening the ValidateException class that you imported earlier.

4. Save the file.

8.3.7 Terminate

The Terminate activity will halt execution immediately without triggering any further fault handling or compensation in the process. We will use this to end our process when invalid input is encountered.

1. Add a Terminate activity inside the Fault Handler and after the Java snippet. Rename it to `Validation Failed`.
2. At this point, it will be obvious that the default structure for the Fault Handler is a flow, not a sequence. Because of this, you will need to add a control link between the Java snippet and the Terminate activity.

Important: A Terminate activity is designed to stop processing but it does not necessarily stop the entire process. There are different contexts in which the activity might be used and these will influence the behavior. In this case, we have used the Terminate inside a Fault Handler. Our process design means that only this Fault Handler will be executing and no parallel activity is happening in the process. A Terminate activity inside a Fault Handler stops all activities that are currently active within it.

For further information about the different effects of a Terminate activity within different contexts, see *The terminate activity* in the Help for WebSphere Studio Application Developer Integration Edition.

At this point, the implementation of the Validation part of NiceJourney is complete and should look like Figure 8-5 on page 213.

8.4 Reserve Flight implementation

This part of the process is used to reserve the flight in accordance with the request that initiated the NiceJourney process. We are not so interested in the implementation of the service so this will be a simple Java class that reports a successful booking. The focus will be on how to model this interaction with a service partner within the business process.

We will invoke the flight booking partner as a synchronous interaction where our business process will block and wait for a reply. This can be compared with the asynchronous style interaction with a partner that we modeled in 8.5, “Reserve Car implementation” on page 226.

Our implementation demonstrates the following activity types:

- Sequence
- Assign
- Invoke

In addition, it will show the following feature of BPEL processes:

- Synchronous invocation of a Java service

We will start with the outline process shown in Figure 8-2 on page 205 and then replace the empty Reserve Flight activity with the set of activities shown in Figure 8-11 on page 223.

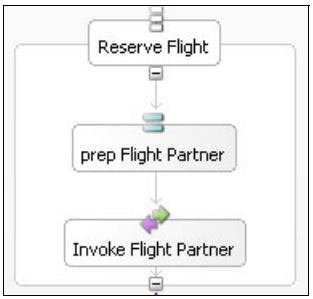


Figure 8-11 Completed Reserve Flight section of the NiceJourney process

A completed solution to this section can also be imported into a fresh workspace by using the Project Interchange Zip format to import ComplexNiceJourneyReserveFlightSolution.ic.zip. See “Project Interchange archive import/export” on page 562 for details about how to import this zip file.

8.4.1 Preparation

We show how to implement this part of the process in isolation. Therefore, you do not need to have completed any of the other implementation sections to be able to follow these steps.

However, as a minimum requirement, you do need to have the workspace prepared with the basic outline of the process. You can do this by either following the steps described in 8.2, “Preparation” on page 205 or by building upon your solution from another section. If you follow the instructions in 8.2, “Preparation” on page 205, you can choose to import a prepared solution and begin working on this section immediately.

If you intend to build upon another section then you can continue to use the same workspace that you used then. In this case, we suggest that you back up your projects at this point by exporting the NiceJourney project as a Project Interchange Zip file. For more information about Project Interchange Zips, see “Project Interchange archive import/export” on page 562.

Once you have either prepared the workspace with the basic outline, or taken your own workspace from the previous section, you are ready to complete the Reserve Flight implementation.

8.4.2 Sequence activity

The flight reservation will be implemented by multiple steps to be performed sequentially. We will not need any conditional flow logic to model this so we will

use a sequence activity to contain this set of steps. Although we could continue to use a flow activity, with a simple control link between each step, a sequence is simpler to work with in the editor because the flow logic is provided for us by the structured activity. We start the implementation of the Reserve Flight part of the process by creating this sequence:

1. Open the NiceJourney.bpel process in the editor.
2. Right-click the **Reserve Flight** activity and select **Change Type** → **Sequence**.

This will create a sequence within which we will place the necessary activities to complete the flight reservation.

8.4.3 Invoke - Java class synchronous invocation

For further information about how to add a Java class to a process and invoke it as a service, refer to 8.3.3, “Invoke - Java Class synchronous invocation” on page 215.

The following steps describe how to create and configure the Invoke activity.

1. Add an Invoke activity inside the Reserve Flight sequence and rename it to `Invoke Flight Partner`.

   This activity will call the external partner to our process that will perform the flight booking for us. The definition of *external* can be anything from another piece of Java code executing withing the same server environment as our process to a service implemented outside our fictional NiceJourney company on a non-Java platform.
2. Create a new Service project called *FlightPartne***r** that will contain the flight booking implementation.
3. Import FlightBookingSystem.java and FlightReservation.java into the folder FlightPartner/com/bookyourflight.

   These are the Java classes that implements the Flight Booking service. Instead of a Java class, the implementation could be any implementation that is callable as a service (describable using WSDL).
4. Drag and drop the FlightBookingSystem.java file onto the NiceJourney.bpel editor canvas.

   Notice that a special type of partner link is created with the Java symbol used as the icon to indicate that this is a partner implemented as a Java program. Select this partner link and note that the implementation is a Java class.
5. Select the **Invoke Flight Partner** activity and go to the implementation properties. Set the Partner Link to **FlightBookingSystem** and the operation to **makeReservation**.

6. Create new Request and Response variables using the **New...** buttons to ensure that the variable types are set automatically. Name them `FlightBookingRequest` and `FlightBookingResponse` respectively.
7. Save the file.

   Look at the Message properties for the two variables to see how they are typed by the auto-generated WSDL file that defines the FlightBookingSystem Java class.

Tip: We found that you need to close and re-open the NiceJourney.bpel editor to see the correct values in the message properties part of the new variables.

The Invoke Flight Partner activity has now been successfully configured to call the makeReservation method of the FlightBookingSystem class, using the process variables FlightBookingRequest and FlightBookingResponse.

8.4.4 Assign

Refer to 7.2.4, “Assign activity” on page 148 for additional instructions on how to use the Assign activity.

We need to use an Assign activity to set the values on the FlightBookingRequest variable before we invoke the FlightBooking Java service. This assign will take the relevant values that were passed in to the process in the process request variable, NiceJourneyInput. The following steps show how to copy these values from NiceJourneyInput to FlightBookingRequest.

1. Add an Assign activity inside the Reserve Flight sequence, before the Invoke Flight Partner activity. Rename it to `prep Flight Partner`.
2. Open the implementation for the Assign and configure copies between the following parts, using NiceJourneyInput in all cases as the From variable and FlightBookingRequest as the To variable for each copy. See Figure 8-12 on page 226 for an example.

- `cityFrom` to `cityFrom`
- `cityTo` to `cityTo`
- `dateTimeDeparture` to `dateTimeDeparture`
- `dateTimeReturn` to `dateTimeReturn`

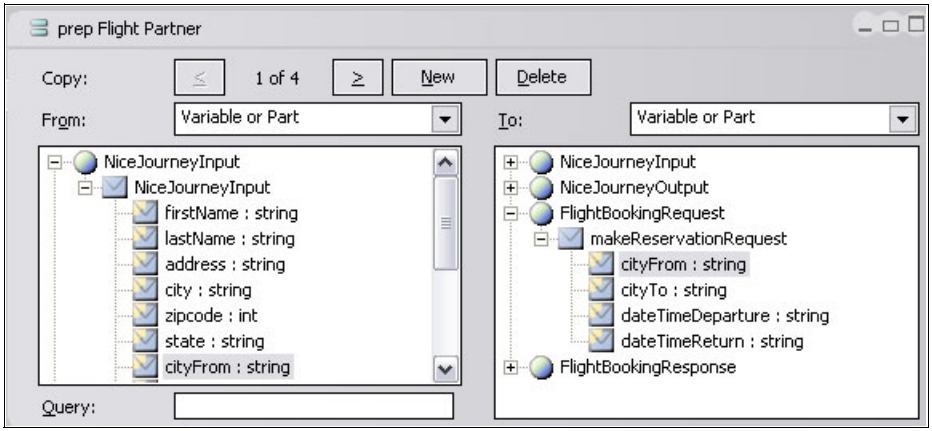


Figure 8-12 prep Flight Partner Assign activity

At this point, the implementation of the Reserve Flight part of NiceJourney is complete and should look like Figure 8-11 on page 223.

8.5 Reserve Car implementation

This section describes how we implemented the Reserve Car section of the NiceJourney process.

In order to reserve a car, we will use a fictional brokerage company called *BookCar* that takes a customer request and finds a suitable car. The BookCar broker will then reply with a car booking that can be from either of two rental companies. The interaction will be modeled as an asynchronous interaction with a business partner that can reply in two possible ways.

In order to do this, we will use the Invoke activity to make a one-way call to the brokerage company. We will then use a Pick activity to wait for one of the two possible replies from the brokerage.

In addition, we will implement the BookCar broker using another BPEL process. This allows us to show you how to invoke a business process from another business process.

Our implementation demonstrates the following activity types:

- Sequence
- Invoke
- Pick
- Assign

In addition, it will show the following features of BPEL processes:

- Asynchronous invocation of another BPEL process
- Correlation
- Conditional links
- Asynchronous interaction pattern

We will start with the outline process shown in Figure 8-2 on page 205 and then replace the empty Reserve Car activity with the set of activities shown in Figure 8-13.

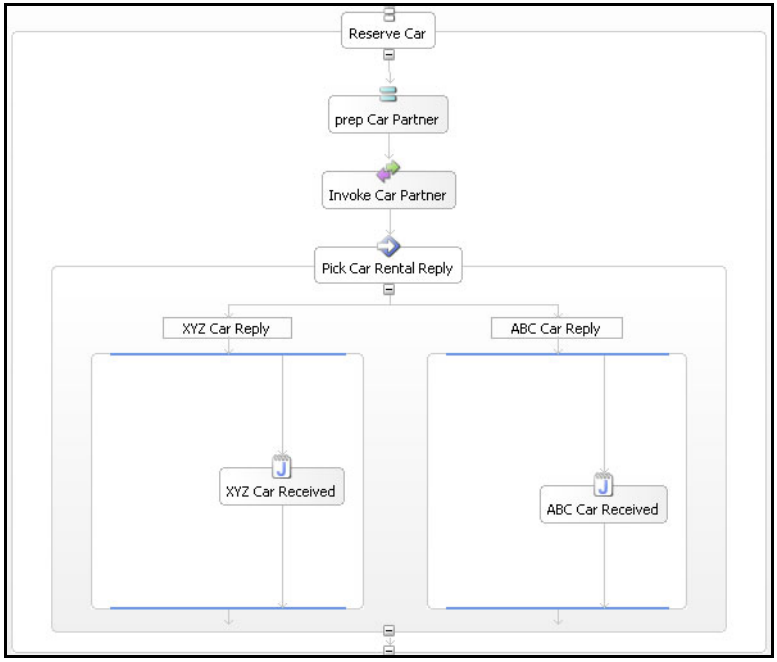


Figure 8-13 Completed Reserve Car section of the NiceJourney process

In addition, the BookCar broker implementation is shown in Figure 8-14 on page 228.

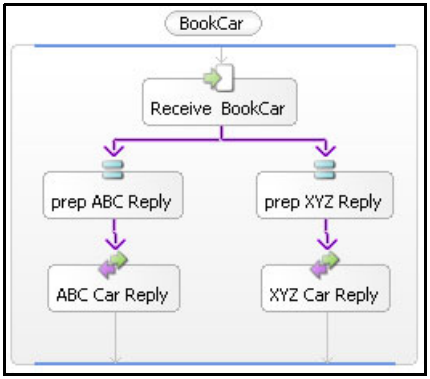


Figure 8-14 Completed BookCar process

A completed solution to this section can also be imported into a fresh workspace by using the Project Interchange Zip format to import ComplexNiceJourneyReserveCarSolution.ic.zip. See “Project Interchange archive import/export” on page 562 for details about how to import this zip file.

8.5.1 Preparation

We show how to implement this part of the process in isolation. Therefore, you do not need to have completed any of the other implementation sections to be able to follow these steps.

However, as a minimum requirement, you do need to have the workspace prepared with the basic outline of the process. You can do this either by following the steps described in “Preparation” on page 205 or by building upon your solution from another section. If you follow the Preparation instructions you can choose to import a prepared solution and begin working on this section immediately.

If you intend to build upon another section then you can continue to use the same workspace that you used then. In this case, we suggest that you back up your projects at this point by exporting the NiceJourney project as a Project Interchange Zip file. For more information about Project Interchange Zips see , “Project Interchange archive import/export” on page 562.

Once you have either prepared the workspace with the basic outline, or taken your own workspace from the previous section, you are ready to complete the Reserve Car implementation.

8.5.2 BPEL process partner

We will create the secondary process that simulates the function of a car rental brokerage company. Once this process is complete then we can then invoke it from the NiceJourney process in an asynchronous interaction.

Creating the BookCar process

1. Create a new BPEL flow-based process called `BookCar` in package com.bookyourcar in the NiceJourney project.

> **Tip:** We found that it was necessary to have the partner BPEL process in the same service project as the client BPEL process. This meant that we had to create BookCar.bpel in the same project as NiceJourney.bpel.

2. Open the BookCar.bpel editor if it is not already open and find the top point of the process, labeled BookCar. Click this and then open the server properties at the bottom of the BPEL Editor.

   This is where you set properties relating to the process itself as opposed to individual activities.

3. Check the **process is long running** option.
4. Save and close the process.

Configuring Namespaces

WebSphere Process Choreographer creates default namespaces to speed up the development of a process and for convenience. In many cases it is worth creating your own specific namespaces as this makes it easier to keep track of them by using memorable, meaningful names. We will change the namespaces to make them more relevant:

1. Open the BookCar.bpel editor if it is not already open and find the top point of the process, labeled BookCar. Click this and then open the server properties at the bottom of the BPEL Editor.

   This is where you set properties relating to the process itself as opposed to individual activities.

2. Change the target namespace to `http://bookyourcar.com/BookCar`. Save and close the file.
3. Switch to the Package Explorer view and open *BookCar.wsdl* from the *NiceJourney/com/bookyourcar* folder. This will open the file in the WSDL editor. Switch to the **Graph** view, rather than the source code view, if it did not open automatically.

4. In the bottom left of the editor is the Definitions section. Click the **Edit Namespaces...** button.

   Tip: If you do not see the Edit Namespaces... button then you may have selected one of the imports, types, services, bindings, port types or messages. You may have to click an area of whitespace in the top section to deselect all of these, at which point the Edit Namespaces... button should appear.

5. Change the Target Namespace value to `http://bookyourcar.com/BookCar` and click **OK**.
6. In the top half of the editor, right-click the import for BookCarInterface.wsdl and click **Delete**. This removes the import for the default namespace. This import will be correctly re-generated later with the updated namespace when we update the process partner link.
7. Save and close the file. Ignore the errors that may appear in the Tasks view at this point.
8. Now open the BookCarInterface.wsdl file and click the **Edit Namespaces...** button in the same way. Change the Target Namespace value, this time using `http://bookyourcar.com/BookCar/interface` to indicate that this is the namespace for the process *interface*.
9. Save and close the file.
10. Optionally, delete the NiceJourney/com/example folder and its contents from the Navigator view in the Resource perspective (this view allows you to work with the underlying file system directly). This will remove the remaining unwanted entries, produced by default from the old namespace values that were set by creating a new business process.

Defining the BookCar interface

1. Open BookCarInterface.wsdl using the WSDL editor and switch to the Graph view.
2. Change the name of the port type from *ProcessPortType* to `BookCar`.
3. Change the operation name from *InputOperation* to `BookCarOperation`.
4. Change the name of the message from *InputMessage* to `BookCarRequest`.
5. Remove the *contents* part of the BookCarRequest message by right-clicking it and selecting **Delete**.
6. Right-click **BookCarRequest** to add new parts by clicking **Add Child** → **part**. Add the following parts:

- startDate
- endDate
- firstName
- lastName
- location

7. Right-click the **input** node of the **BookCarOperation** and select **Set Message...**. Choose **Select an existing message** and select the **tns:BookCarRequest** entry. Click **Finish**.
8. Right-click the **output** node under the BookCarOperation and select **Delete**.

> **Note:** There is no output operation because this process will be invoked in a fire and forget style.

9. The result should look as shown in Figure 8-15. Save the file and close the editor.

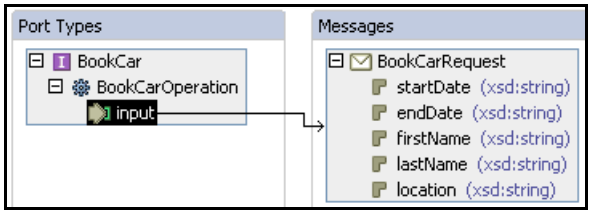


Figure 8-15 Completed BookCar WSDL interface

Using the BookCar process interface

1. Return to the BookCar.bpel editor and select the partner link **PartnerLink** and rename it to `BookCar`.
2. Open the implementation properties for the partner link and click **New ...** to create a new partner link type called *BookCarPLT*. Make sure that the File is set to BookCarInterface.wsdl and that there is only one role.
3. Type `BookCarProcessRole` for the name of the first role.
4. Click **Browse...** to locate the Port Type file and locate NiceJourney/com/bookyourcar/BookCarInterface.wsdl. Choose the **BookCar** port type. This is the port type we just defined when creating our interface. Click **OK** to accept this. Click **OK** again to complete the Partner Link Type definition.
5. The partner link type we just created contains the role that our process interface is to fulfill. On the BookCar partner link implementation properties, click the **<-->** arrow to switch the BookCarProcessRole into the Process Role

Name to the right of the editor. This indicates that the partner link role is being played by the process, not by a partner that the process will invoke.

6. Delete the default process variable named *InputVariable*.
7. Select the **Receive** activity, rename it to `Receive BookCar` and change the implementation settings to:
   - Partner Link: `BookCar`
   - Operation: `BookCarOperation`
8. Click **New...** to the right of Request to create a new variable and name it `BookCarInput`. Because you set the Partner Link and Operation already this variable will be created with the correct message type automatically.
9. Delete the Reply activity.

   Because this process is invoked asynchronously with a fire-and-forget there is no reply. Instead we will reply back to the calling NiceJourney process by sending a message back to the instance of the NiceJourney process that called us.
10. Save the BookCar.bpel file and verify that the Tasks view contains no errors related to the BookCar process and only two warnings about the need to generate deployment code.

Implementing the BookCar process

1. Add activities to the process, rename them and then link them together as shown in Figure 8-16.

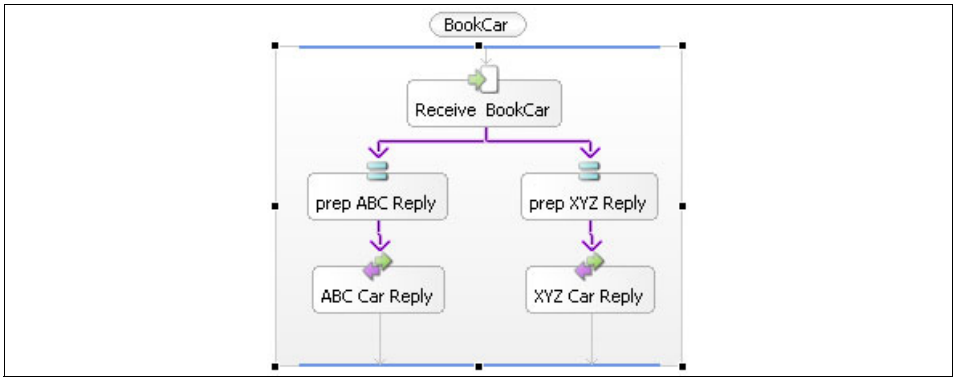


Figure 8-16 BookCar process activities

Tip: You may find it easier to arrange items on the process if you make use of the auto-arrange functionality. Right-click the process or the flow and select whether to arrange or align the process or flow. Selecting the automatic option causes the editor to re-arrange the flow as you work.

2. Save the file. You will see eight errors because the two invoke activities are still waiting to have their input variables, operations, port type and partners defined. The two warnings about generating deployment code from earlier will still be in the Tasks view.

The initial preparation of the BookCar process is now complete but you will notice that the ABC and XYZ Car Reply activities have no implementation yet. These reply activities will be sending a response back to the NiceJourney process that originally called them. However, the NiceJourney process is currently not able to accept these replies so we cannot configure the BookCar to send its replies yet. We first need to extend the NiceJourney process to accept these replies. Then we can complete the BookCar process as described in “Reply - BPEL Asynchronous invocation” on page 244.

8.5.3 Sequence activity

Our car reservation will be implemented by multiple steps to be performed sequentially within the NiceJourney process. We will not need any conditional flow logic to model this so we use a sequence activity to contain this set of steps. Although we could continue to use a flow activity with a simple control link between each step, a sequence is simpler to work with in the editor because the flow logic is provided for us by the sequence structure. We start the implementation of the Reserve Car part of the process by creating this sequence:

1. Open the **NiceJourney.bpel** process in the editor.
2. Right-click the **Reserve Car** empty activity and select **Change Type** → **Sequence**.

This will create a sequence within which we will place the necessary activities to complete the car reservation.

8.5.4 Invoke - BPEL Asynchronous invocation

Now that the second process, BookCar, is available to be called as a partner, we will add a call to it from the NiceJourney process. We also need to set up points in the NiceJourney process where a reply can be received from the BookCar process.

1. Drag and drop the **BookCar.bpel** process onto the **NiceJourney.bpel** canvas in the editor. This will create the special partner link to a BPEL process.

   Tip: When adding a BPEL partner link to your process the secondary BPEL process must exist within the same project as the primary one.

2. Click the **BookCar** partner link and notice how the implementation is the BookCar.bpel file and specifically the BookCar partner link.
3. Add an Invoke activity inside the Reserve Car sequence and name it `Invoke Car Partner`.
4. Open the implementation properties for this activity and set the values to:
   - Partner Link: `BookCar`
   - Port Type: `BookCar`
   - Operation: `BookCarOperation`

   Tip: Instead of opening the implementation properties to set the partner link, try hovering the pointer over the activity until the 'speech bubble' appears at the top right. Click the first icon to set the partner link as shown in Figure 8-17 and you will then be able to connect the arrow that appears to the BookCar partner link in the top right of the process editor. Note that you will still have to set the operation manually in this case.

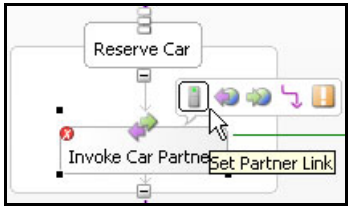


Figure 8-17 Setting a partner link using the graphical editor

5. Click **New...** to the right of Request to create a new variable and name it `BookCarRequest`. Because you set the Partner Link and Operation already this variable will be created with the correct message type automatically.

8.5.5 Assign

Refer to "Assign activity" on page 148 for additional instructions on how to use the Assign activity.

1. Add an Assign activity in front of the Invoke Car Partner activity, inside the Reserve Car sequence, and rename it to `prep Car Partner`. This will be used to set the input variables for the Book Car Request.
2. Open the implementation for the Assign and configure five copies between the following parts, using NiceJourneyInput as the 'From' variable and BookCarRequest as the 'To' variable for each copy. See Figure 8-18 for an example.
   - `firstName` to `firstName`
   - `lastName` to `lastName`
   - `cityTo` to `location`
   - `datetimeDeparture` to `startDate`
   - `dateTimeReturn` to `endDate`

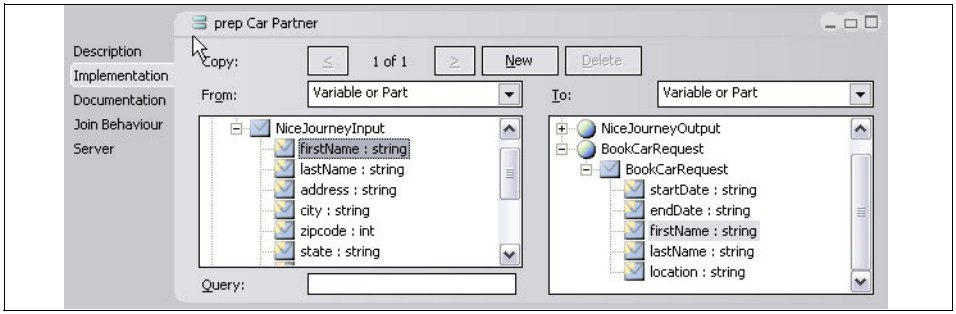


Figure 8-18 Creating the Assign activity

3. Save the process and close the editor.

There should be no errors related to the NiceJourney process and one warning about the deployment code. The eight errors for the BookCar process will still exist and we will fix them soon.

8.5.6 Pick activity

The BookCar partner is invoked asynchronously and will subsequently respond by sending a reply back to the NiceJourney process. It can reply in two possible ways - an 'ABC Car Reply' or an 'XYZ Car Reply' which represent two different types of car rental agreements that the BookCar broker could provide us. Our NiceJourney process needs to receive either one of these two replies and then pick how to proceed based upon which reply is received. We can model this using the Pick activity.

A Pick activity will wait for one of a number of possible replies to occur and then proceed based upon which one was received. This is exactly the behavior we need.

In addition to creating the Pick activity we need to create the extra receive capability so that our process can be replied to from the BookCar process. We will do this by adding the additional operations to our process interface. These receive operations will be used to accept replies from BookCar.

Adding Reply operations to the NiceJourney interface

The BookCar process will reply to the NiceJourney process by invoking some operation that is available on the NiceJourney process interface. We need to add operations that allow BookCar to reply with either an ABC or an XYZ car rental.

1. Open NiceJourneyInterface.wsdl from the folder NiceJourney/com/nicejourney.
2. Use the graphical WSDL editor to add the following items. The steps are similar to those used when editing the BookCarInterface WSDL in “Defining the BookCar interface” on page 230.
   - Port Type: XYZCarRental with Operation: XYZCarRental
   - Port Type: ABCCarRental with Operation: ABCCarRental
   - Message: XYZCarRentalReply
   - Message: ABCCarRentalReply
3. Add an input node to the XYZCarRental and ABCCarRental operations.
4. Set the Message on each input node to either XYZCarRentalReply or ABCCarRentalReply as appropriate.
5. Add the following parts to the XYZCarRentalReply message, ensuring that you change the type from xsd:string where appropriate:
   - firstName (xsd:string)
   - lastName (xsd:string)
   - price (xsd:long)
   - carType (xsd:long)
6. Now add these parts to the ABCCarRentalReply message, again ensuring that you change the type from xsd:string where appropriate:
   - class (xsd:string)
   - price (xsd:int)
   - name1 (xsd:string)
   - name2 (xsd:string)
   - termsandconditions (xsd:string)
7. The final configuration should look as shown in Figure 8-19.

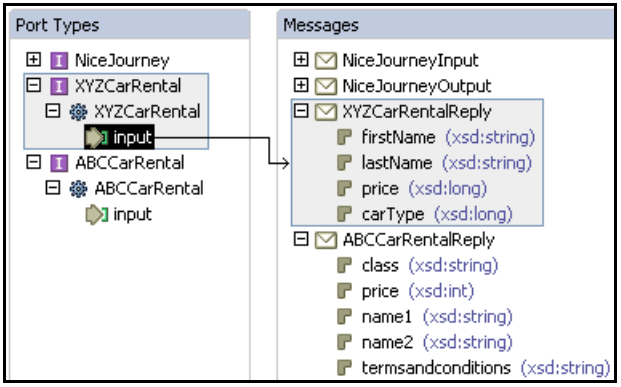


Figure 8-19 Adding the car rental reply operations to the NiceJourney interface

Notice that there are two different messages, one for each reply type. You will see these used shortly in the Pick activity.

8. Save the file and close it.

Adding the Pick activity to the NiceJourney process

Now that the process interface changes are complete we have two extra operations that can be performed on our process. We will use these operations to receive replies from the BookCar process. It will reply by invoking these one way operations on our NiceJourney process. It could invoke either one of XYZCarRental or ABCCarRental. A Pick activity will handle what to do if either one of these operations is called.

1. Open **NiceJourney.bpel** in the BPEL Editor.
2. Add a Pick activity after the *Invoke Car Partner* activity and rename it to `Pick Car Rental Reply`.
3. Right-click the Pick activity and choose **Add OnMessage.**
4. Select the OnMessage and set its Display Name to `XYZ Car Reply` on the Description tab.
5. Add another OnMessage in the same way and name it `ABC Car Reply`.
6. Drag and drop **NiceJourneyInterface.wsdl** onto the **NiceJourney.bpel** editor canvas. Select the **XYZCarRental** operation and click **OK**.

   NiceJourneyInterface.wsdl is where we defined the operations so we need to add it as a Partner Link.
7. Repeat the drag and drop but this time select **ABCCarRental** for the operation.

This creates additional partner links for the XYZCarRental and ABCCarRental operations. We need these when configuring the Pick activity.

8. Select the **XYZCarRental** partner link and switch to the implementation tab. Click the **<-->** arrow to switch the roles and indicate that this is an operation available on the process, not an operation that the process will call.
9. Do the same for the *ABCCarRental* partner link
10. Select the **XYZ Car Reply onMessage** section of the Pick activity and set the following implementation values:
    - Partner Link: `XYZCarRental`
    - Port Type: `XYZCarRental`
    - Operation: `XYZCarRental`
11. Click **New...** to the right of **Request** to create a new variable and name it **XYZCarRentalReply**. Because you set the Partner Link and Operation already this variable will be created with the correct message type automatically.
12. Select the **ABC Car Reply onMessage** section of the Pick activity and set the following implementation values:
    - Partner Link: `ABCCarRental`
    - Port Type: `ABCCarRental`
    - Operation: `ABCCarRental`
13. Click **New...** to the right of **Request** to create a new variable and name it **ABCCarRentalReply**. Because you set the Partner Link and Operation already this variable will be created with the correct message type automatically.
14. Save the process.

    You should see three errors regarding the NiceJourney process (and still eight others regarding BookCar.bpel) and one warning to generate the deployment code. The errors relate to *correlation* and we will fix them soon.

Customizing the Pick behavior

The OnMessage structures allow you to add behavior that you want to execute upon receipt of a given incoming message. We will add different behavior to each one to indicate what kind of car rental we received.

1. Add a Java snippet to the XYZ Car Reply onMessage block and rename it to `XYZ Car Received`. Add the following code to its implementation:

```
System.out.println("XYZ Car Rental will be used");
```

2. Add another Java snippet, this time to the *ABC Car Reply* onMessage block and rename it to `ABC Car Received`. Add the following code to its implementation:

   ```
   System.out.println("ABC Car Rental will be used");
   ```

   The code in these Java snippets performs no business function but it allows us to illustrate the effect of the Pick activity more clearly by printing a message in the server console on receipt of one of the two Car Rental messages.

   In a real world scenario, it is likely that some additional business processing of the results from the car rental would be required.

3. Save the NiceJourney.bpel file

8.5.7 Correlation sets

There are two types of messages that a process can receive at runtime:

1. Initiation messages

   These messages start a new process. They are not passed to a process that is already executing but instead start a new one.

2. Input messages

   These messages are passed to a process *instance*. This means that they provide input to an already executing process instance, at some point within the process.

In our process, the initial Receive activity (named Receive NiceJourney) is an initiation point. When the WebSphere Process Choreographer runtime receives a message of the type defined on this activity (NiceJourneyInput) then it will initiate an instance of the NiceJourney process.

In contrast to this Receive activity, the onMessage sections for the Pick activity are designed to accept messages to the process *while it is running*. These are points within the process where messages arrive that target a particular instance of NiceJourney that is already running. There could be many NiceJourney processes executing within the runtime - how do we match up an incoming message with a specific active instance of NiceJourney?

The answer is to use correlation sets. A correlation set enables us to match incoming messages with specific process instances. We want to correlate an incoming XYZCarReply or ABCCarReply (from BookCar process) with the particular NiceJourney instance that originally invoked the BookCar instance that is replying. We need some piece of identifier information that will be used between the two processes to provide the link.

In our scenario we will use a correlation set that uses the customer name as the unique identifier information. We will use both the first name and second name parts to build our unique identifier. The correlation set will then be initialized when the first request comes in to initiate a NiceJourney process. After this, the correlation set will be passed between the NiceJourney and BookCar process instances so that they can specifically communicate with each other.

Note: Using the customer name alone is not a perfect correlation set choice because there is no guarantee of uniqueness. It is conceivable that two customers could share the same name. A more sophisticated choice could involve adding a timestamp value or some other unique identifier.

Configure the correlation set as follows:

1. Locate the CorrelationSet list on the NiceJourney.bpel canvas. Typically this is to the left of the process. Click the **+** symbol to add a new correlation set.
2. Rename the new correlation set to `CustomerName`.
3. Click the **Properties** tab of the correlation set.

   A correlation set can have multiple properties, each defined by either a primitive type or as a type or element in an XSD file. In our example we want a firstName part and a lastName part of primitive type xsd:string.
4. Click **New...** to create a new property named `firstname` and of built-in type `xsd:string`. Refer to Figure 8-20 on page 241. Ignore the aliases section for now - we will come back and fill in this information later.

   Tip: Notice that the New property dialog asks you for a file name to create the correlation property in. This always defaults to <ProcessName>Interface.wsdl (NiceJourneyInterface.wsdl) in our case. This is the default file that is used for partner links as well and we have used it to define our processes interface and operations. However, if you decide to use another filename then make sure that you alter the file name here to match your chosen interface filename before creating the property.
5. Repeat the above step to create a property called `lastname`, also of type `xsd:string`.

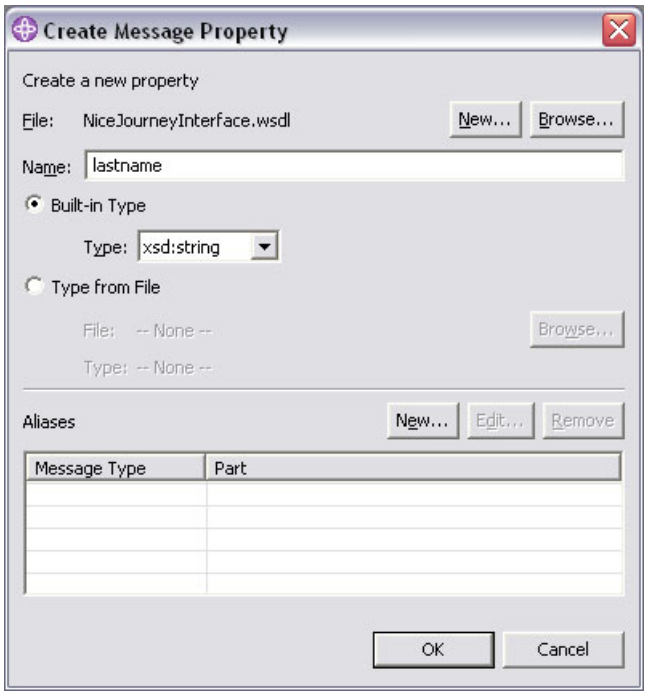


Figure 8-20 Adding properties to a correlation set

Now that we have the correlation set defined and we know the properties (a firstname and a lastname) of the set, we need to consider how those properties are set and where they are used. When operations are called on the process (for example an XYZ or ABC car reply will call one of the operations we defined in the WSDL earlier) they are called by passing a message. What we must do is associate each correlation set property with specific parts within each of the possible messages.

For example, the input to our process is a NiceJourneyInput message. It has many parts but one of these is to be associated with the *firstname* property of the correlation set. In the same way, another part of the NiceJourneyInput message must be associated with the lastName part of the correlation set.

In the Reserve Car activity we then need to invoke the BookCar process and at this point we need to pass this correlation set information. However, the message type (and associated parts) that we call BookCar with are different than we used to call NiceJourney. The message type is now a BookCarRequest instead of a NiceJourneyInput. That means we need to associate the correct parts of the BookCarRequest message with the correlation properties as well.

This is why there is the concept of aliases. They are used to indicate which parts of a given message matches which property of the correlation set. Different messages will have the correlation set property information held in different parts.

6. Select the **firstname** property and click **Edit...**
7. Click **New...** to create a new alias
8. Browse to *NiceJourney/com/nicejourney/NiceJourneyInterface.wsdl* and select the **NiceJourneyInput** nessage. This indicates that we will find an alias for the firstname property in the NiceJourneyInput message.
9. Select the **firstName** part of the message and click **OK.**
10. Click **New...** again to create another alias.
11. This time, browse to *NiceJourney/com/bookyourcar/BookCarInterface.wsdl* and this time select **BookCarRequest**. Select the **firstName** part and click **OK**.
12. Click **New...** to create another alias and this time select the **XYZCarRentalReply** message, found in file *NiceJourney/com/nicejourney/NiceJourneyInterface.wsdl*
13. Select the **firstName** part and click **OK**.
14. Finally, create another alias using the `name1` part from the ABCCarRentalReply message in NiceJourney/com/nicejourney/NiceJourneyInterface.wsdl.
15. Click **OK** to complete editing the firstName message property.
16. Repeat steps 6 through 15 but using the appropriate parts for the lastname correlation set property instead of firstName, taking them from the same messages, in the same WSDL files that the firstname parts were obtained from.
17. You should end up with a correlation set called CustomerName with two properties, each with four aliases taken from the messages NiceJourneyInput, BookCarRequest, XYZCarRentalReply and ABCCarRentalReply.
18. Save the process. There should be one extra item in the Tasks view - an information entry to tell us that our correlation set is not used at present which causes a validation failure.

    Now that all the aliases are in place we need to specify on the relevant individual activites that they need to use our correlation set.
19. Select the first activity, **Receive NiceJourney** and locate the **Correlation** settings tab.
20. Click **Add** to add a new correlation to this activity.

Tip: When you have added the correlation it can appear as if the values are permanently set and that they have been chosen correctly by the product. Be wary that this is not the case. In this example there is only one correlation set so we have been presented with the correct one but this would often not be the case. Also note that each value can be changed by selecting the item and opening the drop-down selection box.

21. Set the direction to **Receive**, Initiation to **Yes** and Correlation Set to **CustomerName**. You can change values by clicking them and then using the drop-down box.

    Here we are indicating that on the receive direction (the only direction for a Receive activity, unlike an invoke) this correlation set should be initiated. The correlation set properties will then remain set for the lifetime of the process (they cannot be changed). They will be set from the parts of the *NiceJourneyInput* message that is received by this activity, in accordance with the aliases we defined for the correlation set properties.

22. Now select the activity **Invoke Car Partner**.

23. On the correlation tab, click **Add** to add the correlation set and check that it is set to:

    – Direction: `Send`
    – Initiation: `No`
    – Correlation Set: `CustomerName`

    We are not initiating the correlation set this time, instead we are using the values that we initiated when the process was started by a NiceJourneyInput message.

24. Select the **XYZ Car Reply** onMessage section for the Pick activity and add a new correlation with these properties:

    – Direction: `Receive`
    – Initiation: `No`
    – Correlation Set: `CustomerName`

25. Repeat the previous step for the ABC Car Reply onMessage section of the Pick activity.

26. Save the process.

    There should be no errors associated with the NiceJourney process and only one warning to generate the deploy code. The BookCar process still has eight errors that we will now resolve.

27. Close the file.

8.5.8 Reply - BPEL Asynchronous invocation

We need to return to the BookCar process and complete its implementation now that the NiceJourney process is ready to accept replies through its Pick activity.

1. Open the **BookCar.bpel** file in the BPEL Editor
2. Drag and drop the **NiceJourney.bpel** file onto the canvas to create a partner link to the NiceJourney process.
3. Select the partner link, which defaulted to being called NiceJourney, and rename it to `ABCCarReply`.
4. Select its implementation tab and change the partner link to `ABCCarRental`.
5. Drag and drop the **NiceJourney.bpel** file onto the **BookCar** canvas once again.
6. Select the partner link which again defaulted to being called NiceJourney and rename it `XYZCarReply`.
7. Set its implementation partner link value to `XYZCarRental`.

   We now have partner links back to the NiceJourney process that invoke its two receive points that the pick chooses between. Next we must associate the invoke activities with the correct partner link.
8. Select the ABC Car Reply Invoke activity and change its implementation values to:
   - Partner Link: `ABCCarReply`
   - Port Type: `ABCCarRental`
   - Operation: `ABCCarRental`
9. Click **New...** to the right of **Request** to create a new variable and name it `ABCCarReply`. Because you set the Partner Link and Operation already this variable will be created with the correct message type automatically.
10. Select the **XYZ Car Reply** Invoke activity and change its implementation values to:
    - Partner Link: `XYZCarReply`
    - Port Type: `XYZCarRental`
    - Operation: `XYZCarRental`
11. Click **New...** to the right of Request to create a new variable and name it `XYZCarReply`. Because you set the Partner Link and Operation already this variable will be created with the correct message type automatically.
12. Save the file.

    There should be no errors in the Tasks view at this point and only two warnings that we still haven't generated any deploy code.

8.5.9 Assign

We want to provide some output data from the broker, BookCar. Again we are not too concerned with the implementation business logic and are focussing instead on the integration logic. Therefore, we will use simple assigns to determine the output data from the brokerage. In reality there would likely be some kind of complex algorithm for determining an appropriate response.

Refer to “Assign activity” on page 148 for additional instructions on how to use the Assign activity.

1. Select the **prep ABC Reply** Assign activity and configure it to copy the following parts between the BookCarInput and ABCCarReply variables
   - firstName and name1
   - lastName and name2
2. In addition, use the Assign implementation editor to set these fixed values to variables in ABCCarReply. See Figure 8-21 on page 246 for an example.
   - From: `Fixed Value`

     Type: `xsd:string`

     Value: `Driver must be over 25`

     To: `termsandconditions`
   - From: `Fixed Value`

     Type: `xsd:string`

     Value: `B`

     To: `class`
   - From: `Fixed Value`

     Type: `xsd:int`

     Value: `2000`

     To: `price`

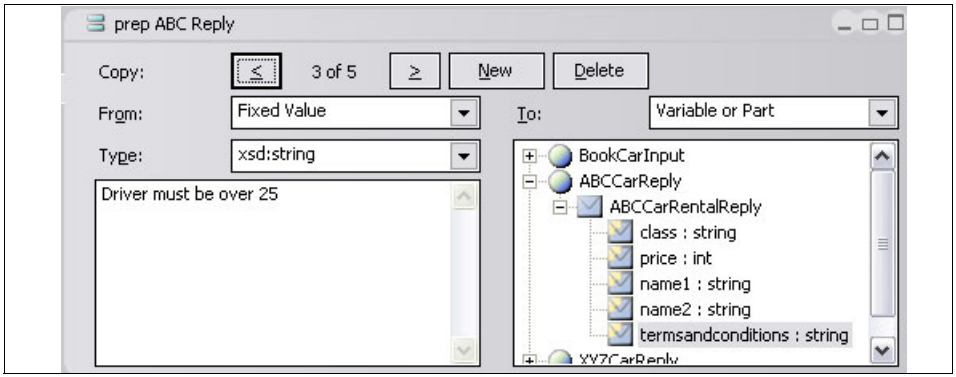


Figure 8-21 Assigning fixed values to BookCar reply parts

3. Select the prep XYZ Reply Assign activity and configure it to copy the following parts between the BookCarInput and XYZCarReply variables.
   - firstName and firstName
   - lastName and lastName
4. In addition, use the Assign implementation editor to set these fixed values to variables in ABCCarReply. See Figure 8-21 for an example.
   - From: `Fixed Value`

     Type: `xsd:long`

     Value: `2`

     To: `carType`
   - From: `Fixed Value`

     Type: `xsd:long`

     Value: `500`

     To: `price`
5. Save and close the file.

8.5.10 Conditional link

Although the process is now free from errors, the BookCar process is not quite complete. We want this process to have some kind of intelligence and to choose which of the two car type replies to send, based upon the input that this simulated brokerage received.

In this chapter, we are focused on the choreography of individual services, not the implementation of each service itself. Although BookCar is a special type of

service in that it is a process itself, we still do not need to put much functionality into it in order to demonstrate the points. Therefore, we will demonstrate the use of a conditional link to hold some simple business logic only.

1. Open the BookCar.bpel process in the BPEL Editor.
2. Select the control link that joins the Receive BookCar Receive activity and the prep ABC Reply Assign activity.
3. Switch to the **Condition** tab for this activity and choose **Visual Expression** for the Value. This will open the Visual Expression editor as shown in Figure 8-22.

> **Tip:** Visual Expression is a way of visually composing expressions for comparing variables and values, without writing Java code. If you like, you can optionally convert your visual expression to its pure Java representation if you like. To do this, first build the visual expression and then change the Value selection to `Expression` instead of `Visual Expression`.

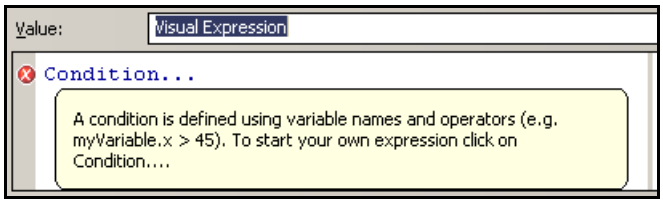


Figure 8-22 The Visual Expression editor

4. Click the **Condition ...** and you will see the list of variables to the right of the expression editor. Expand the **BookCarInput** and click the **location** part. This will start the expression with the phrase *BookCarInput.location*.
5. Click the **Method or Field** option to specify a method and click the **equalsIgnoreCase(anotherString)** method as shown in Figure 8-23 on page 248. This will allow us to make a case-insensitive comparison to a string value.

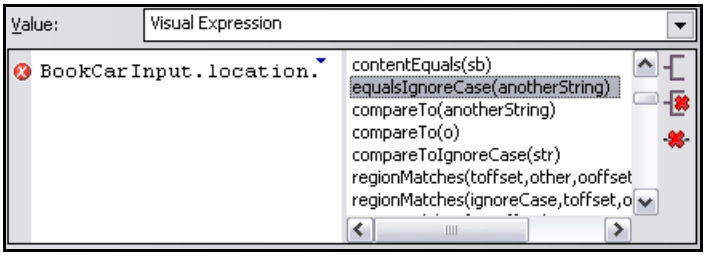


Figure 8-23 Creating the Visual Expression

> **Note:** You may be wondering why we did not choose the '==' operator and instead selected the equalsIgnoreCase(anotherString) method. This is because the comparison will be made in Java. Java uses the '==' operator to compare two *objects* to see if they are the same. Two strings that have the same content may not be the same object - they are just two objects with the same properties. In this case a '==' comparison will return false. What we want to do is compare the string contents, not the objects themselves. Therefore, we use the equalsIgnoreCase(anotherString) method instead of '==' to check this.

6. At this point you will see that the expression is

   ```
   BookCarInput.location.equalsIgnoreCase(anotherString)
   ```

   The cursor (blue arrow pointing down) will be positioned after this string. However, we want to enter a value for 'anotherString'.

7. Click the **anotherString** text and this will open up the list of variables and other items to choose from on the right. We are going to hardcode our comparison string as this is a simple piece of demonstration business logic. You could however compare against the contents of another variable for example.

8. Click **String** at the bottom of the right hand list and enter `New York` into the box as shown in Figure 8-24. Select **Return**.

Figure 8-24 Setting the value for the String comparison

9. This completes the Visual Condition which should now read:

   ```
   BookCarInput.location.equalsIgnoreCase( "New York" )
   ```

   A control link will only be followed if its condition evaluates to true. This Visual Expression will ensure that this link is only followed when the car booking location is New York. This means that our broker will only return an ABC Car Reply for New York rentals.

   We want to complete the business logic by saying that in all other cases the broker should send an XYZ Car Reply. We can do this by saying that the alternative control link path has a condition of 'otherwise'. This means that if all other control link options evaluate to false then this link should be followed. So if our check for a location of New York results in false then instead we follow the alternative link to an XYZ Car Reply.

10. Select the control link between Receive BookCar and prep XYZ Reply activities. Set the condition value to `Otherwise`.

11. Save the process.

The Book Car part of the process has now been completed. We modeled this as a partnership with a brokerage company that asynchronously finds a suitable car for our travel requirements and then sends it back to us some time later.

We used another BPEL process to provide an implementation of the brokerage and demonstrated how to invoke it from the first process. We used a one way fire and forget to invoke the second process then used a Pick activity to wait for possible responses. In the second process we chose which type of reply to send and then called back to the first process so that the Pick activity could proceed.

At this point the implementation of the Reserve Car part of NiceJourney is complete and should look like Figure 8-13 on page 227. The BookCar process should be as shown in Figure 8-14 on page 228.

8.6 Reserve Hotel implementation

This section describes how we implemented the Reserve Hotel section of the NiceJourney process.

In order to reserve a hotel we will use a manual step that requires a real person to organise the hotel arrangement. This is a powerful option provided by WebSphere Process Choreographer and there is extensive support for incorporating human interactions into a process. We will show how a team of agents could complete the hotel bookings as they arise and provide the details of the reservation back to the NiceJourney process in order that processing may continue.

In order to do this, we will use the Staff activity to create a work item that can be claimed by an agent. The agent can make the hotel booking, probably by calling or faxing a suitable hotel, and then complete the work item and pass the reservation details back to the process.

We will also make use of the Transformer activity which allows us to do more advanced message mappings and variable manipulation that is possible with an assign.

Our implementation demonstrates the following activity types:

- Sequence
- Staff
- Transformer

We will start with the outline process shown in Figure 8-2 on page 205 and then replace the empty Reserve Car activity with the set of activities shown in Figure 8-25.

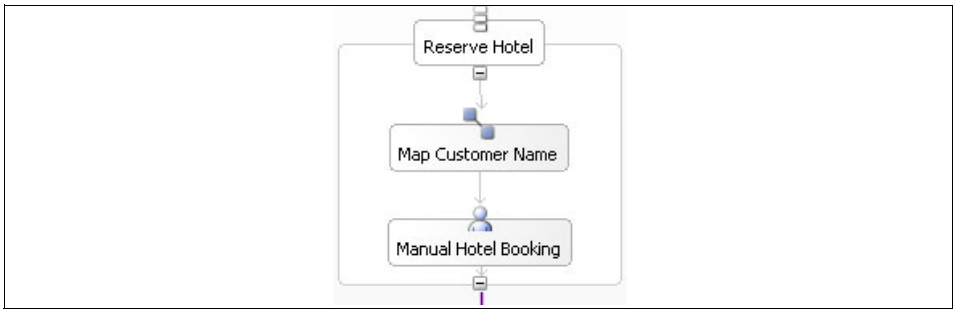


Figure 8-25 Completed Reserve Hotel section of the NiceJourney process

8.6.1 Preparation

As with the other implementations, we again show how to implement this part of the process in isolation. Therefore, you do not need to have completed any of the other implementation sections to be able to follow these steps.

However, as a minimum requirement you do need to have the workspace prepared with the basic outline of the process. You can do this by either following the steps described in “Preparation” on page 205 or by building upon your solution from another section. If you follow the Preparation instructions you can choose to import a prepared solution and begin working on this section immediately.

If you intend to build upon another section then you can continue to use the same workspace that you used then. In this case, we suggest that you back up

your projects at this point by exporting the NiceJourney project as a Project Interchange Zip file. For more information about Project Interchange Zips see , “Project Interchange archive import/export” on page 562.

Once you have either prepared the workspace with the basic outline, or taken your own workspace from the previous section, you are ready to complete the Reserve Hotel implementation.

8.6.2 Sequence activity

The hotel reservation will be implemented by multiple steps to be performed sequentially. We will not need any conditional flow logic to model this so we will use a sequence activity to contain this set of steps. Although we could continue to use a flow activity, with a simple control link between each step, a sequence is simpler to work with in the editor because the flow logic is provided for us by the structured activity. We start the implementation of the Reserve Hotel part of the process by creating this sequence:

1. Open the NiceJourney.bpel process in the editor.
2. Right-click the **Reserve Hotel** activity and select **Change Type** → **Sequence**.
3. Save the process.

This will create a sequence within which we will place the necessary activities to complete the hotel reservation.

8.6.3 Staff activity

The Staff activity is used to represent the point in the process at which human involvement is required to proceed. The Staff activity is defined by an operation that has associated input and output messages. This operation and these messages are defined in a WSDL file, typically on the interface to the process.

The process is then defined to have two variables with message types that match the input and output for the operation associated with the Staff activity. In addition, during development the persons who are allowed to perform the staff activities are defined in an abstract manner. At this point, although the developer indicates what kind of person can perform the activity, they do not know about the technology that will be used to implment the security enforcement.

At runtime, when the Staff activity is reached, the WebSphere Process Choreographer runtime engine creates a work item that can be claimed by any staff who are potential owners of the work item. The runtime is pre-configured with the necessary security registry implementation. The WebSphere Process Choreographer container maps the abstract security role that the developer specified to a real query against the security registry implementation. This

mechanism allows the runtime to determine who is a potential owner and then enforces this policy.

Once the work item is claimed by an eligible operator, the request variable and its contents are passed to that person who then uses this information to decide how to populate the response variable. The person enters this data and then completes the work item, passing this response variable and its contents back to the process which can then continue execution.

The most likely method of interaction between a person and their work items will be through the Process Web Client. This supplied application enables a person to log in and then interact with running processes in accordance with the configured security. For example, a person qualified to be a hotel booking operator would log in and see all processes that needed a hotel booking to be completed. They could then claim one of these processes which would mark that processes instance as claimed. It would then be unavailable for claim by other qualified hotel booking operators. Once claimed, the person then has responsibility to complete the Staff activity by providing the data to pass back to the process instance to allow execution to continue.

In our case, we will provide the hotel booking operators with the customer name, location required, checkin and checkout dates. The hotel operator would then claim this activity and make a manual hotel booking based upon this information. The reply from our Staff activity will be defined as requiring a reservation ID and a price so the hotel booking operator must provide these values and then complete the Staff activity. The operation that defines this request and response (input and output) must first be added to the NiceJourney process.

Defining the HotelBooking operation

First we must extend the process interface to create the operation that will be used by the Staff activity. This operation defines the input and output messages that will be used by the Staff activity. We will define the operation and messages in the same interface WSDL that we have used for the process interface already.

1. Open NiceJourneyInterface.wsdl from folder NiceJourney/com/nicejourney in the WSDL editor.
2. Add a new Port Type called `StaffHotelReservation`.
3. Add an operation to it called `StaffHotelReservationOperation`.
4. Add an input node to this operation.
5. Add an output node to this operation.
6. Right-click the **input** node and choose **Set Message...**
7. Select the **Create a new message** option and name it `StaffHotelRequest`.

8. Click **Finish**.
9. Right-click the **output** node and choose **Set Message...**.
10. Select the **Create a new message** option and name it `StaffHotelResponse`.
11. Click **Finish**.
12. Add the following parts to the *StaffHotelRequest* message (each should be of type xsd:string):
    - location
    - checkinDate
    - checkoutDate
    - customerName
13. Add the following parts to the StaffHotelResponse message:
    - reservationID
    - price
14. Change the type of both parts from xsd:string to xsd:long.
15. The completed interface changes should look as shown in Figure 8-26.

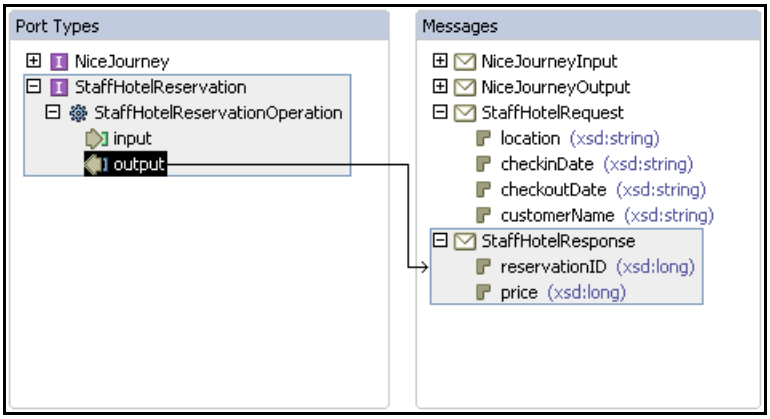


Figure 8-26 Adding StaffHotelReservation to the NiceJourneyInterface.wsdl

16. Save and close the NiceJourneyInterface.wsdl file.

Adding the Staff activity

1. Open NiceJourney.bpel in the BPEL Editor.
2. Add a Staff activity inside the Reserve Hotel sequence and rename it to `Manual Hotel Booking`.

3. Open the implementation properties and click **Browse...** to locate the NiceJourneyInterface.wsdl file in folder NiceJourney/com/nicejourney. Choose the **StaffHotelReservation** port type and click **OK**.

   Tip: If the StaffHotelReservation port type does not appear then you may find that closing the NiceJourney.bpel file and then re-opening it will refresh the list of Port Types.

4. The Port Type and Operation values should automaticallly be set. Check that they are StaffHotelReservation and StaffHotelReservationOperation.
5. Click **New...** to the right of **Request** to create a new variable and name it **StaffHotelRequest**. Because you set the Partner Link and Operation already this variable will be created with the correct message type automatically.
6. Click **New...** to the right of **Response** to create a new variable and name it **StaffHotelResponse**. Because you set the Partner Link and Operation already this variable will be created with the correct message type automatically.
7. Save the process.

   At this point there should be no errors in the workspace and one warning to remind you to generate the deployment code.

8.6.4 Transformer Service activity

You may have noticed that the interface to the Staff activity defined the StaffHotelRequest message and that this message has a part called customerName. This part needs to be populated with the customer's full name, in other words a concatenation of the first and seconds names.

In the NiceJourney process we have created a variable called StaffHotelRequest that we will pass as the request variable for the Staff activity. This variable must be populated with the correct data. The customer's name can be taken from the NiceJourneyInput variable but this has it split across two parts - firstName and lastName.

An Assign activity is frequently used for copying the value of variable parts between variables. In this case we want to copy two parts of one message into one part of another message. This requires a more advanced message transformation.

We can use a Transformer service to perform such advanced message manipulation. A Transformer service is also a re-usable service that we describe in WSDL and it can therefore be used in multiple places within a process or even

in another process. You should be careful to use Assign if possible however because it is likely to perform better than a Transformer.

A Transformer also allows you to do much more complicated mappings, for example combining multiple parts from multiple messages. Transformers use XPath and eXtensible Style Language (XSL) transforms. Because the data variables are defined by XML documents, XSL transformations allow you to make any conversion between data variables

We show how to create a new Transformer service and place a transformer activity in the process to call it:

1. Add a Transformer activity just before the Manual Hotel Booking Staff activity, inside the Reserve Hotel sequence, and rename it to `Map Customer Name`.
2. Switch to the implementation tab for the transformer activity and select the following values from the drop-down selection boxes:
   - Request: `NiceJourneyInput`
   - Response: `StaffHotelRequest`
3. Now that you have specified the input and output messages, note that the **New...** button becomes enabled. Click this to launch the new transform wizard, which will automatically know that you want to map between parts of NiceJourneyInput, in order to set the parts fo StaffHotelRequest. Complete the values as shown in Figure 8-27 on page 256. Click **OK** when you have completed the dialog.

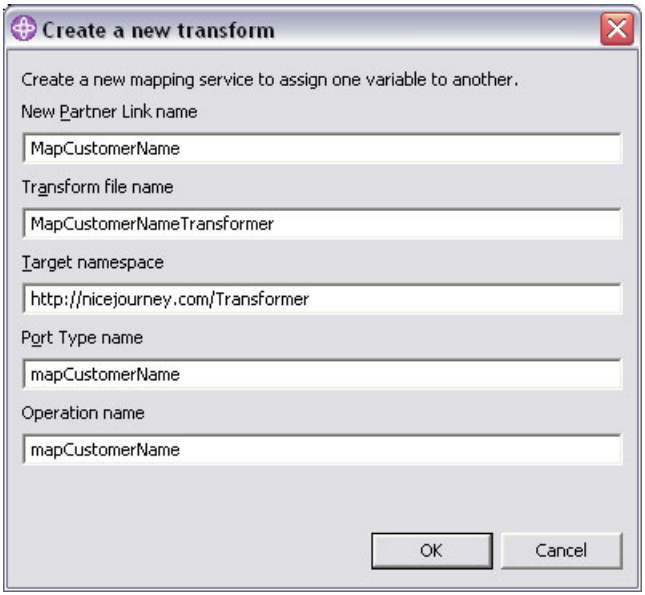


Figure 8-27 Creating a new transform

4. The new transformer file, MapCustomerNameTransformer.wsdl is opened in the transformer editor, a special editor for WSDL files that describe transform services.

   Take a look around the editor. See on the left hand side is the input message, in this case NiceJourneyInput, and on the right hand side is the output, in this case StaffHotelRequest. It is possible to add additional input messages that can be aggregated together to create the output message. You could add more input messages by using the **Transformer Editor** → **Add Input Message** option from the menu system. However, all of the input information that we need in this case can be found in the single message, NiceJourneyInput.

5. Drag the **firstName** part of NiceJourneyInput from the left of the editor and drop it onto the **customerName** part of StaffHotelRequest.

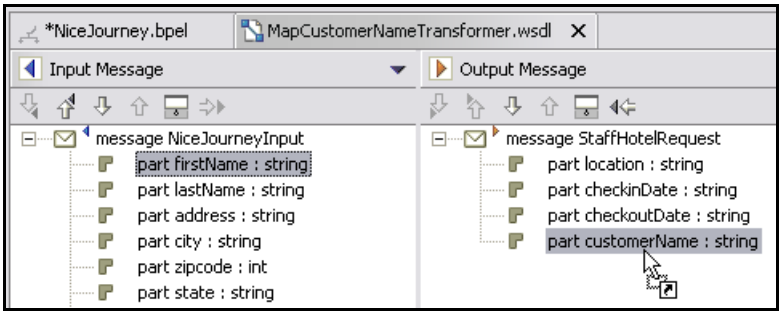


Figure 8-28 Creating the mapCustomerName transform

6. The editor will show the mapping in the bottom section as shown in Figure 8-29. Notice that the output message is on the left in this section with the input in the middle. On the right is an XPath Expression to show how the firstName part is located within the input message using XPath.

Figure 8-29 Basic XPath expression

7. Perform similar drag and drops to map:
   - NiceJourneyInput.city to StaffHotelRequest.location
   - NiceJourneyInput.dateTimeDeparture to StaffHotelRequest.checkinDate
   - NiceJourneyInput.dateTimeReturn to StaffHotelRequest.checkoutDate
8. At this point we still need to combine the NiceJourneyRequest.lastName part with the first name. Drag and Drop the **lastName** part onto **StaffHotelRequest.customerName**. Now, both first and last names are being dropped onto *customerName*.

   You will notice that the XPath Expression for the StaffHotelRequest.customerName is now empty. This is because the editor can no longer assume a simple copy of the data value between the input and output parts. How should the two input parts provided be combined to create this output part? We need to provide this information.
9. Click in the empty **XPath Expression** box for the part *customerName* until you see the **...** button appear as shown in Figure 8-30 on page 258. You may have to click at least twice.

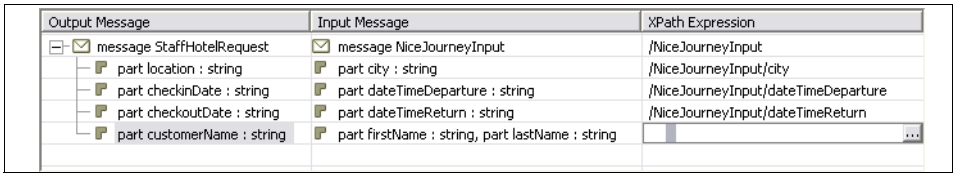


Figure 8-30 Customizing the MapCustomerName transformer

10. Click the **...** button to open the details of the transformation.

    Note that you can click any of the values in the Input Message or XPath Expression columns of the editor and then alter their settings from here.

    You will see the transformation details and notice that the default action has been to perform a join between the two input variables. We want to pass the customer name in the format “lastname, firstname”. This means we need to join them with a “, “ delimeter.

11. Click **/NiceJourneyInput/lastName** and then click the up arrow to move the lastName to the beginning.

12. Click the Delimeter field to the right of **/NiceJourneyInput/lastName** and enter **“, “** without the quotation marks. Then click away from the entry field. The results are shown in Figure 8-31. Notice that there is an Example which shows how the two parts are to be joined, using their XPath references. Click **Finish**.

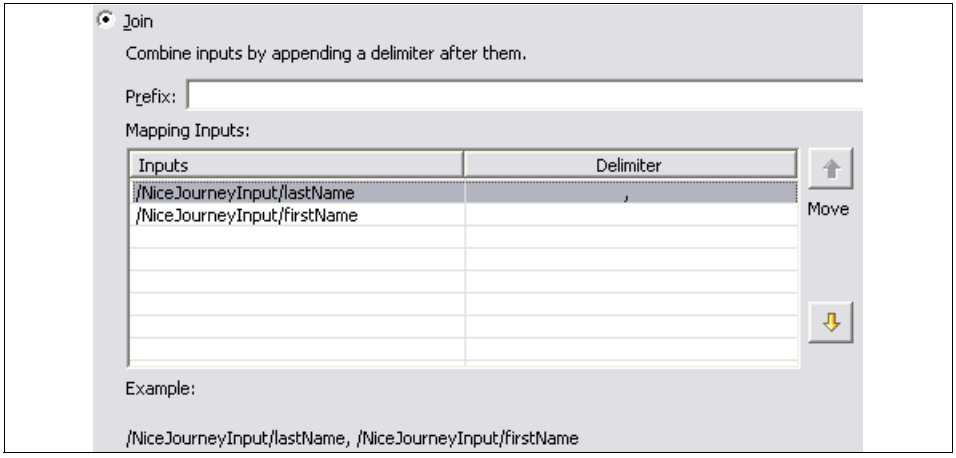


Figure 8-31 Creating the join transform

13. In the transformer editor you will now see the join expression. Review the other XPath Expressions to see examples of a ‘Move’ expression which makes a simple copy of a parts value.

Tip: We found that if you did not set every part of the output message when using the Transformer then an exception would be thrown at runtime. You should ensure that every part of the output message is set to some value by your transformer in order to avoid this problem.

14. Save the MapCustomerNameTransformer.wsdl file and close the transformer editor
15. Save the NiceJourney.bpel file and close the file.

The Reserve Hotel part of the process has now been completed. We modeled this as a manual step that required a person to take the hotel requirements, manually make a booking, and then respond to the business process. We saw how simple it was to add such an interaction to the process.

We also used a Transformer service to perform some message mapping intelligence above and beyond the capabilities offered by an Assign activity. This was important because the Staff activity interface used a different format for the customer name - a single part instead of two separate ones for first and last names.

8.7 Bill Customer implementation

This section describes how we implemented the Bill Customer section of the NiceJourney process.

In order to bill the customer, we will first determine the type of card that the customer supplied, allowing either a *credit* or *debit* card type. We will use a Switch activity to switch behavior depending upon the card type and will call a different payment service for each type of card that was used. The payment services will then validate the card number and debit the account, creating errors if there is a problem with this step.

In fact, there are a number of possible errors within the Bill Customer section so we will build more comprehensive error handling logic than that used in 8.3, “Validation implementation” on page 212.

As a result of an error, the process may be unable to bill the customer successfully. This would constitue a failure of the overall NiceJourney booking process and in this scenario, we will reverse the booking of the hotel, car and flight because they will not be required if payment fails.

In order to reverse the booking procedures we will use a feature of WebSphere Process Choreographer known as *Compensation*. A service invocation can be

defined to have an associated compensating service. This compensating service can then be used to reverse the work done when the service was originally invoked, In some scenarios, the compensating action might not be a reversal of the forward process but may be some alternative action that compensates for what was done. For this reason, the term Compensation is more appropriate and will be used instead of reverse in the remainder of this chapter. An example would be the mailing of a letter to a customer. This cannot be reversed but it can be compensated for by mailing a subsequent letter advising to ignore the first one.

Our implementation demonstrates the following activity types:

- Switch
- Assign
- Invoke
- Throw

We will replace the empty Bill Customer activity from Figure 8-2 on page 205, with the set of activities shown in Figure 8-32. However, unlike for the other sections, we cannot start implementing Bill Customer from this basic outline because it does not have Reserve Hotel, Reserve Flight or Reserve Car implemented and we need these to be complete so that we can configure compensation for them.

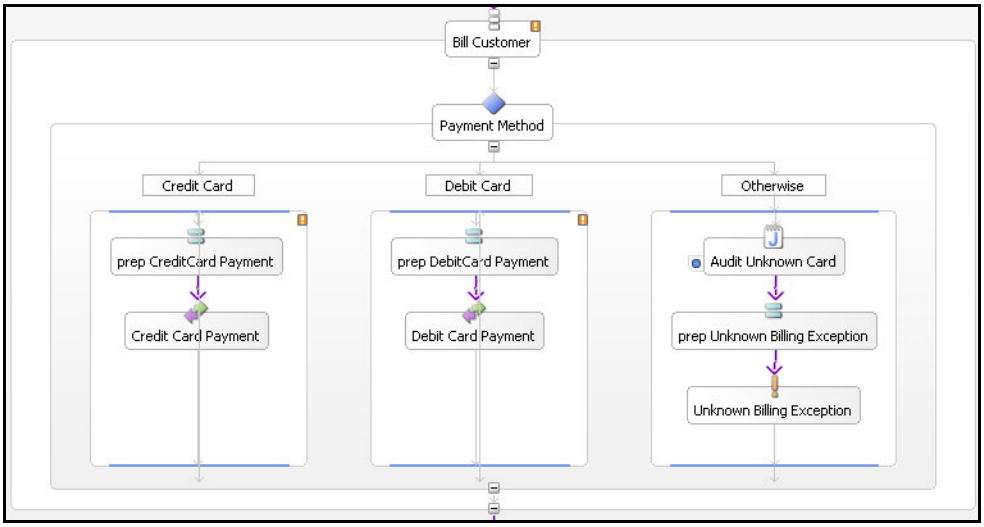


Figure 8-32 Completed Bill Customer section of the NiceJourney process

8.7.1 Preparation

Because we will be using compensation in this section of the chapter we need to have the forward processing implementations for Reserve Hotel, Reserve Car and Reserve Flight already completed. We will then configure the compensating actions necessary to reverse these reservations when a billing failure occurs.

Therefore, we did *not* start the Bill Customer implementation with the outline process shown in Figure 8-2 on page 205. Instead, we first implemented the three reservation sections as described in 8.4, “Reserve Flight implementation” on page 222, 8.5, “Reserve Car implementation” on page 226 and 8.6, “Reserve Hotel implementation” on page 249.

You can either:

- Follow each of these sections and implement them sequentially, building your NiceJourney process until all three implementations are added.
- Import our solution that contains all three implementations already.

If you intend to build each implementation section sequentially then we suggest that you back up your projects as you progress by exporting the NiceJourney project as a Project Interchange Zip file. For more information about Project Interchange Zips see , “Project Interchange archive import/export” on page 562.

If you choose to start with our prepared solution, import the Project Interchange zip, *ComplexNiceJourneyReserveFlightCarHotelSolution.ic.zip*

Once you have either prepared the workspace yourself, or imported our solution workspace, you are ready to complete the Bill Customer implementation.

8.7.2 Switch

The payment can be made by two types of card, credit or debit. We want to perform different steps in the process depending upon which card is used so that we can invoke a different payment system partner for each type of card. The ideal activity type is the *Switch*. This activity enables choosing one of a number of different possible paths in the process.

A Switch activity is similar to a case statement in many programming language. In fact, when you configure a Switch activity, you use the Add Case option to create different possible paths. A Switch activity will execute one, and only one, of these possible paths.

We want to choose different paths based upon the card type that was specified. The following steps show how to add the Switch to the process and then how to

specify the conditional logic that will determine which of the different cases is followed.

Note: A final subtlety with the Switch activity is that when more than one case evaluates to true, the first one will be chosen and the other(s) will not be executed.

1. Open the NiceJourney.bpel process in the editor.
2. Right-click the **Bill Customer** activity and select **Change Type** → **Switch**.
3. Right-click the **Switch activity** and choose **Add Case**

   This creates the first case. Each case has its own flow to specify what processing to perform if that case is the one that executes. Therefore, the container for the new case is a flow.
4. Select the new case and change its display name in the Description properties to `Credit Card`.
5. Add another case to Bill Customer switch and change its display name to `Debit Card`.
6. Right-click the **Bill Customer** Switch activity again but this time choose **Add Otherwise**.

   We are using the otherwise case for when the card type is neither credit or debit. We will treat this as an error case and configure it to reflect this.

   Now that we have the two cases for the different card types, we have to set the condition for when each case will evaluate to true.
7. Select the **Credit Card** case flow and change to the **Condition** properties. Select **Visual Expression**.
8. Use the Visual Expression editor in a similar way to that described in “Conditional link” on page 246 to build the following condition:

   ```
   NiceJourneyInput.cardType.equalsIgnoreCase( "credit" )
   ```
9. Select the **Debit Card** case flow and change its condition, again using the Visual Expression builder but this time specifying:

   ```
   NiceJourneyInput.cardType.equalsIgnoreCase( "debit" )
   ```
10. Save the process. The completed Switch should look as shown in Figure 8-33 on page 263.

Figure 8-33 Completed Bill Customer Switch activity

At this point we have set up the Switch activity framework for billing the customer but now we need to perform the correct processing for each different type of card. We need to invoke a different payment processing service for each card type and also handle any errors that may be thrown by those services.

Tip: We found it useful to collapse the sequences for Reserve Car, Reserve Flight and Reserve Hotel while implementing Bill Customer so as to reduce the amount of busy canvas space in the BPEL Editor at one time. You can do this by clicking the - (minus) sign at the root of any sequence structured activity.

8.7.3 Import the Payment Processing Services

We created some simple services that simulate the processing of a card payment. These services must be imported into the workspace before they can be configured as partner links and then invoked from within the NiceJourney process.

These services are supplied in a Project Interchange Zip file. For more information about Project Interchange Zips see , “Project Interchange archive import/export” on page 562.

1. Import the Project Interchange zip, BillCustomerPaymentPartners.ic.zip.
2. Check that the Payment Partners project now exists in the workspace.

Figure 8-34 Payment Partner Services Implemenations project

Explore the implementation of the credit and debit card processing services. They are simple Java classes but we also created them to return exceptions when the payment processing was invalid. We will need to deal with these exceptions from within the process when we invoke the services.

8.7.4 Creating the partner links

Now that the services are available in the workspace we will create partner links in NiceJournet.bpel to invoke them. We will add the Java files directly and have WebSphere Studio Application Developer Integration Edition therefore auto-generate the Web services definition of the them.

1. Drag and drop the **CreditCardPayment.java** file from PaymentPartners/com/nicejourney/payments/credit onto the NiceJourney.bpel canvas. After a few seconds of processing the CreditCardPayment partner link will appear.
2. Repeat the above step but this time drag and drop **DebitCardPayment.java** from its package location to create the DebitCardPayment partner link.
3. Save the process.

8.7.5 Credit Card case

Now we will create the processing logic for the Credit Card case of the Bill Customer Switch activity.

Invoke

1. Add an Invoke activity inside the Credit Card case flow and rename it to `Credit Card Payment`.
2. Set the Partner Link in its Implementation properties to `CreditCardPayment`. The Port Type and Operation will automatically be selected.
3. Click **New...** to the right of Request to create a new variable and name it `CreditCardPaymentRequest`. Because you set the Partner Link and Operation already this variable will be created with the correct message type automatically.

4. Click **New...** to the right of Response to create a new variable and name it `CreditCardPaymentResponse`. Because you set the Partner Link and Operation already this variable will be created with the correct message type automatically.

Assign

1. Insert an Assign activity before the new Invoke activity and rename it to `prep Credit Card Payment`.
2. Be sure to create the necessary control link from the Assign to the Invoke.
3. Use the Assign activity to assign the following parts from the NiceJourneyInput variable to the CreditCardPaymentRequest variable:
   - cardNumber to cardNumber
   - firstName to firstName
   - lastName to lastName
4. Save the process. This completes the Credit Card payment case which should look as shown in Figure 8-35.

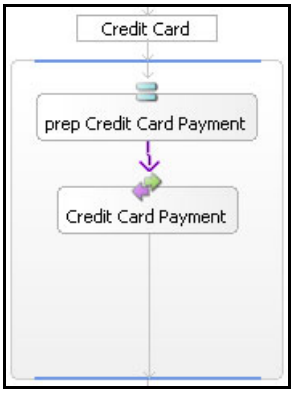


Figure 8-35 Completed Credit Card Case

8.7.6 Debit Card case

Now we will create the processing logic for the Debit Card case of the Bill Customer Switch activity. It is very similar to the Credit Card case.

Invoke

1. Add an Invoke activity inside the Debit Card case flow and rename it to `Debit Card Payment`.
2. Set the Partner Link in its Implementation properties to `DebitCardPayment`. The Port Type and Operation will automatically be selected.

3. Click **New...** to the right of Request to create a new variable and name it `DebitCardPaymentRequest`. Because you set the Partner Link and Operation already this variable will be created with the correct message type automatically.
4. Click **New...** to the right of Response to create a new variable and name it **`DebitCardPaymentResponse`**. Because you set the Partner Link and Operation already this variable will be created with the correct message type automatically.

Assign

1. Insert an Assign activity before the new Invoke activity and rename it to `prep Debit Card Payment`.
2. Be sure to create the necessary control link from the Assign to the Invoke.
3. Use the Assign activity to assign the following parts from the NiceJourneyInput variable to the DebitCardPaymentRequest variable:
   - cardNumber to cardNumber
   - firstName to firstName
   - lastName to lastName
4. Save the process. This completes the Debit Card payment case which should look as shown in Figure 8-36.

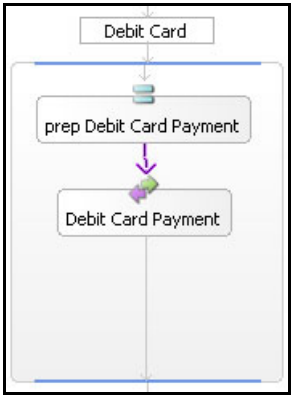


Figure 8-36 Completed Debit Card Case

8.7.7 Unknown Card Otherwise case

When an unknown card type is encountered we will not be able to process the payment for the NiceJourney booking that we have made and this will therefore result in a failure. It would be more realistic to have checked the card type prior to

making the reservations but we will use it here to illustrate a possible failure in payment processing.

We will configure an unknown card type to throw a fault within the process. Later we will configure the error handling to deal with this scenario.

We also want to audit any failures due to an unknown card type. We will use a simple Java snippet activity to do this but the auditing could be implemented as a separate invokable service if necessary.

Audit

1. Add a Java snippet inside the Otherwise case and rename it to `Audit Unknown Card`.
2. Add the following Java code to its Implementation properties:

```
System.out.println("Unknown Card Type Passed To Billing. Card Type: "+
getNiceJourneyInput().getCardType());
System.out.println("PAYMENT FAILED ");
```

3. Save and close the NiceJourney.bpel file.

Throw

We are going to throw a fault that will indicate that an exception has happened in the billing. We will define this fault ourself and must therefore define its message format.

In this way, you create your own user-defined faults and then throw and catch them as appropriate in your process.

1. Open NiceJourneyInterface.wsdl in the WSDL editor.
2. Use the Graph view of the WSDL editor to create a new message named BillingException with a single part named detail, of type xsd:string. See Figure 8-37.

Figure 8-37 Definition of the BillingException message

3. Save and close NiceJourneyInterface.wsdl.
4. Open NiceJourney.bpel in the BPEL Editor again.

   Now that the message type has been defined we will add a Throw activity and use our message type for the fault variable that the Throw will use.

5. Add a Throw activity to the Otherwise case of the Bill Customer Switch activity. Rename it to `Unknown Billing Exception`. Do not create any control links at this point.
6. Switch to the **Fault** properties for the Throw activity and select **User-defined**.
7. Enter `BillingException` for the Fault name.
8. Click **New...** to create a new variable called `UnknownBillingException`.

> **Note:** Any fault variable that you define in this way will automatically be created. When creating it the BPEL Editor needs to decide what message definition to use for the variable.
>
> Because you are free to define your own fault message definitions (as we have just done), the editor does not know what message definition to use. Therefore, a default supplied definition is used, called *bpelFault*. This fault is used for errors specific to the execution of a BPEL process, not your own processes. If you want to work with potential errors that might happen in the BPEL runtime environment, consider using the *bpelFault* message type.
>
> More typically, you will want to provide your own error information, often with business relevant information. In this example, we are going to alter the default message type that *UnknownBillingException* has been created with, and instead use our own definition.

9. Select the newly created **UnknownBillingException** variable at the bottom of the variables list (in the top left of the canvas).
10. Switch to its **Message** properties and notice that it is defined by a *bpelFault* message - a supplied built-in type that carries information about process execution errors. We want to use our own message definition instead.
11. Click **Browse ...** and locate NiceJourney/com/nicejourney/NiceJourneyInterface.wsdl and then choose the BillingException message in the drop-down selection. Click **OK**.
12. Finally add a control link between the Audit Java snippet activity and the Throw activity as shown in Figure 8-38 on page 269.

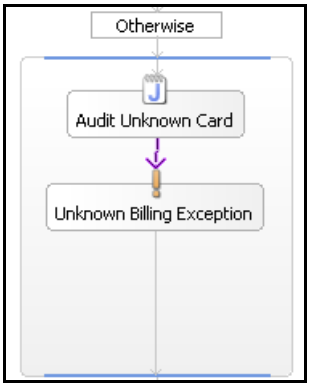


Figure 8-38 Completed Unknown Card Case

13. Save and close the NiceJourney.bpel process.

8.7.8 Fault handling

In this section we will be creating Fault Handlers that will control what happens in the events of faults occuring in the process execution. We have already deliberately created our own Throw activity and we will use a Fault Handler to catch this fault.

In addition, faults can come directly from an activity without being explicitly thrown by a deliberate Throw activity. For example, the credit and debit card payment partners can both throw a fault if payment processing fails.

Fault handlers are associated with activities within the process and are defined to handle certain faults. A given Fault Handler will be invoked whenever a matching fault type occurs within the activity it is associated with.

Because many activities are actually containers for other activities (for example a flow, sequence, pick or switch) then the Fault Handler applies to all activites contained within the container. The term *scope* is useful when describing what set of activities a Fault Handler applies to. For example, a Fault Handler defined on a Switch activity has a scope which contains all activities within that switch construct.

Scopes are also nested. Whenever a fault occurs in an activity, the WebSphere Process Choreographer runtime checks to see if there is a Fault Handler for that scope. If there isn't then it will pass it up to the next enclosing scope, which itself may or may not have a Fault Handler. This proceduce is then repeated all the way up to the scope of the process itself.

We will use Fault Handlers with different scopes to catch the faults we are interested in. We will even apply a Fault Handler to the entire process so that we can catch any fault condition that occurs in any of the enclosed scopes.

Credit card

The Java implementation of the Credit Card Payment system uses a method called processPayment. The method signature for this method shows that it can throw a Java exception called *CreditCardException*.

WebSphere Process Choreographer is based upon Web services and uses XML to define its variables. These variables are used for input and output when invoking services. In addition, services can return errors using *faults* - the name given to special return messages that represent an error scenario. The data type of a fault is defined in the same way as the input and output - by using XML message definitions within a WSDL file.

The Java credit card payment partner was added by dragging and dropping the Java class onto the canvas. This is a quick way of firstly exposing the Java class as a Web service and secondly adding it to the process canvas. WebSphere Studio Application Developer Integration Edition has automatically created a WSDL definition that is equivalent to the Java class but has made it easier by doing this automatically.

If we want we can inspect the Web service definition that was created by looking at the generated WSDL file that represents the service. For the credit card payment we would see that the service defines an input, an output, but also a fault. This fault is the WSDL representation for what is represented in Java as the CreditCardException class.

What this means is that when we invoke the credit card payment partner, we might receive a fault message reply instead of the successful output. We will add a Fault Handler that is designed to catch such a fault and to take action on it.

Adding the Fault Handler

1. Select the **Credit Card** section of the Bill Customer switch and click the **Add Fault Handler** icon as shown in Figure 8-39 on page 271. Be sure to select the complete flow and not one of the individual activities that are part of it.

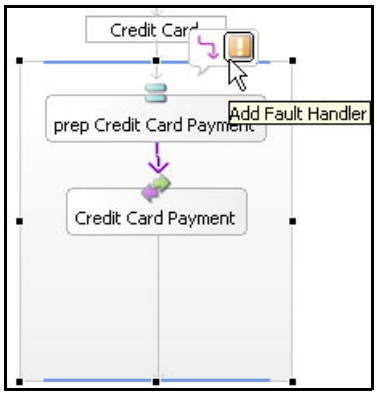


Figure 8-39 Adding a Fault Handler to the Credit Card section

This causes a Fault Handler to be associated with the Credit Card section and the Fault Handler is indicated by a small orange exclamation mark icon in the top right corner of the flow. By default, the Fault Handler itself is not actually shown on the canvas when it is first created.

Important: In fact, it is only possible to display one Fault Handler at a time on the canvas. If you try to show a Fault Handler when another one is already displayed, the first one will automatically close when the new one is displayed.

2. We want to edit the Fault Handler so first we need to display it. There are two ways of doing this. Either double-click the small icon in the top right corner of the Credit Card case or alternatively right-click the whole flow and select **Show Fault Handler**. The Fault Handler is displayed as an empty structure as shown in Figure 8-40.

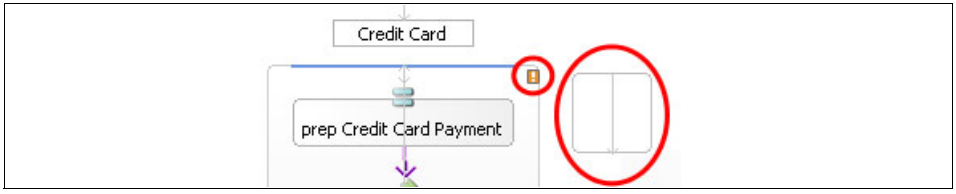


Figure 8-40 Fault Handler (circled to the right) and Fault Handler icon (circled left)

3. Select the **Fault Handler** and then use the speech bubble icons to select **Add Catch**. Alternatively, right-click and select it from the pop-up menu instead.

This creates a catch that can be configured to catch certain types of fault occuring within the Fault Handler's scope. Alternatively you can add Catch All to catch all faults within the scope, irrespective of the actual type of the fault.

Note: Once a Catch has been created it is necessary to define the Fault which you are trying to catch. There are lots of acceptable combinations for defining a catch:

- faultname and no variable
- typed faultVariable but no faultName
- fully specified with faultName and faultVariable

When a fault occurs, it will be matched by the Catch which most specifically matches it. The more qualified a Catch, the more you will know why it has been thrown. If no match is found then it will go to the Catch All of the Fault Handler.

Of particular importance is the Namespace definition which must be correctly set to the namespace of the fault message type. Incorrectly setting the Namespace is a frequently encountered problem that leads to the Catch All being executed, rather than the intended Catch that has been incorrectly specified.

4. Select the new Catch block and complete the information :

   Fault type: `User-defined`

   Namespace:
   `http://nicejourney.com/comnicejourneypaymentscreditCreditCardPayment NiceJourneybpel/`

   Fault Name: `CreditCardException`

5. Click **New ...** to create a fault variable and name it `CreditCardException`.

   Note: We want to catch the specfic fault that is thrown by the Credit Card payment system. Therefore we specify the exact same namespace that is used to define the fault on the credit card system. This was auto-generated when we dragged and dropped the Java class onto the canvas. We must also set the fault name to match that defined in the same WSDL file.

6. Select the new **CreditCardException** variable from the variables list at the top left of the canvas.
7. On its message properties page, click **Browse ...** and locate the WSDL file that was generated for the Java credit card payment system. This file is located at

`NiceJourney/com/nicejourney/comnicejourneypaymentscreditCreditCardPaymentNiceJourneybpel.wsdl` . Select the **CreditCardException** message definition shown in Figure 8-41. Click **OK**.

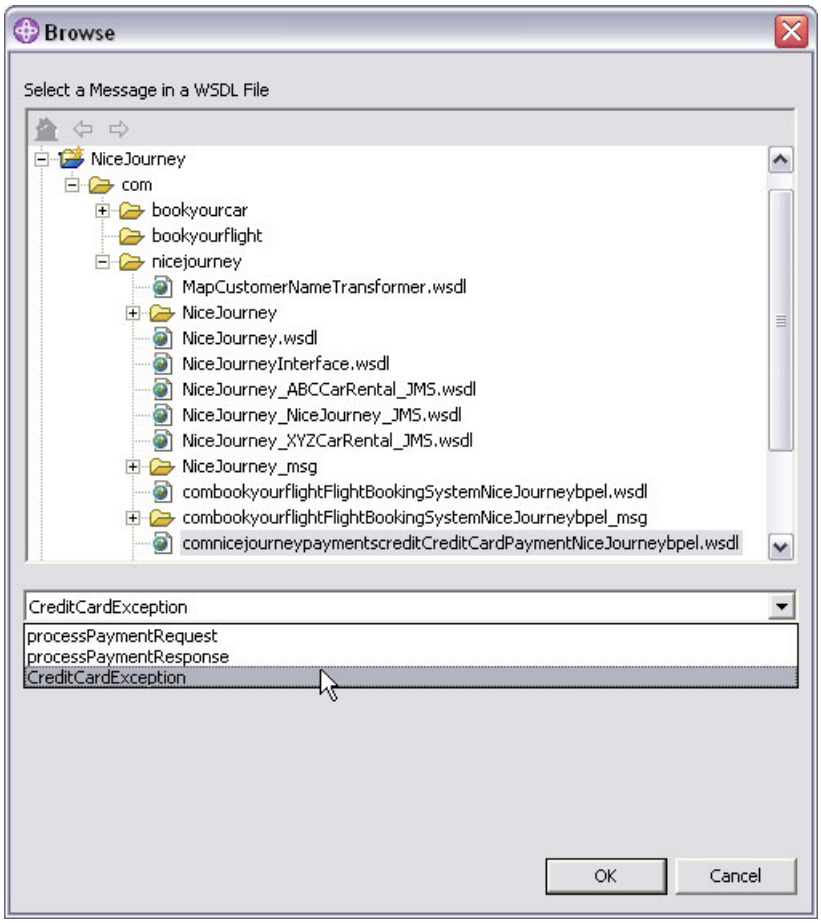


Figure 8-41 Changing the CreditCardException variable type

Completing the Fault Handler

We will use the Fault Handler to audit that a credit card payment has failed and then we will throw a new fault. This new fault will indicate that a more generic failure has occured - the billing process as a whole has failed. (We have already used this technique when dealing with the unknown card type). This fault will then be handled by another Fault Handler that we will create later on the enclosing Bill Customer scope.

1. Add a Java snippet inside the Fault Handler, naming it `Audit CreditCard Exception` and adding the following code for its Implementation:

```
System.out.println("Credit Card Payment Failure: " +
getNiceJourneyInput().getCardType() + " " +
getNiceJourneyInput().getCardNumber());
```

This will write the invalid card details to the system log for auditing purposes.

2. Add a Throw activity and rename it to `Credit Billing Exception`.
3. Change its Fault properties to:

   Fault type: `User-defined`

   Namespace: `http://nicejourney.com/NiceJourney`

   Fault Name: `BillingException`
4. Click **New ...** and name the new variable `CreditBillingException`.

   We can re-use the message type that we used when we threw an exception earlier for an unknown card. However, we must create another variable for this specific Throw activity, even though the variable will share the message definition with the UnknownBillingException variable we created earlier.
5. Select the new **CreditBillingException** variable from the variables list and Browse to change its Message definition to **BillingException** as defined in NiceJourney/com/nicejourney/NiceJourneyInterface.wsdl. Click **OK**.
6. Return to the Fault Handler and add an Assign activity and rename it to `prep CreditCard Exception`. Use it to copy the *reason* part of the `CreditCardException` variable to the *detail* part of the `CreditBillingException` as shown in Figure 8-42.

Figure 8-42 Assigning the data from CreditCardException to CreditBillingException

7. Finally add control links so that the Java snippet connects to the Assign which then connects to the Throw as shown in Figure 8-43 on page 275.

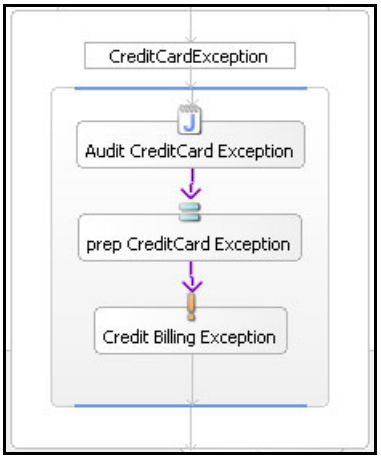


Figure 8-43 Completed Fault Handler for faults occuring in the Credit Card scope

8. Save and close the NiceJourney.bpel process.

We have now completed a Fault Handler that will catch any faults that are thrown by failures in the Credit Card payment partner. The Fault Handler will log the credit card failure and credit card details before throwing a more generic fault to indicate that billing has failed. At a higher scope we will handle all billing failures (credit card problem, debit card problem or unknown card) using a shared fault message type and Fault Handler.

Debit Card

We will create a very similar Fault Handler for the debit card payment as we used for the credit card payment. Therefore the description of the steps we took will be briefer in this section than the previous one.

Adding the Fault Handler

1. Open **NiceJourney.bpel** in the BPEL Editor.
2. Select the **Debit Card** section of the `Bill Customer` switch and click the **Add Fault Handler**. Be sure to select the complete flow and not one of the individual activities that are part of it.
3. Right-click the whole flow and select **Show Fault Handler**.
4. Select the Fault Handler and then use the speech bubble icons to select **Add Catch**. Alternatively, right-click and select it from the pop-up menu instead.
5. Select the new Catch block and complete the information :

   Fault type: `User-defined`

Namespace:
`http://nicejourney.com/comnicejourneypaymentsdebitDebitCardPaymentNiceJourneybpel/`

Fault Name: `DebitCardException`

6. Click **New ...** to create a fault variable and name it `DebitCardException`.
7. Select the new **DebitCardException** variable from the variables list at the top left of the canvas.
8. On its Message properties page, click **Browse ...** and locate the WSDL file that was generated for the Java debit card payment system. This file is located at NiceJourney/com/nicejourney/comnicejourneypaymentsdebitDebitCardPaymentNiceJourneybpel.wsdl . Select the **DebitCardException** message definition and click **OK**.

Completing the Fault Handler

The content of the Fault Handler will be very similar to that used for the Fault Handler on the Credit Card payments scope.

1. Add a Java snippet inside the Fault Handler, naming it `Audit DebitCard Exception` and adding the following code for its Implementation:

```
System.out.println("Debit Card Payment Failure: " +
getNiceJourneyInput().getCardType() + " " +
getNiceJourneyInput().getCardNumber());
```

   This will write the invalid card details to the system log for auditing purposes.

2. Add a Throw activity and rename it to `Debit Billing Exception`.
3. Change its Fault properties to:

   Fault type: `User-defined`

   Namespace: `http://nicejourney.com/NiceJourney`

   Fault Name: `BillingException`

4. Click **New ...** and name the new variable `DebitBillingException`.

   Again, we can reuse the message type that we used when we threw an exception earlier for an unknown card. However, we must create another variable for this specific Throw activity, even though the variable will share the message definition with the `UnknownBillingException` and `CreditBillingException` variables we created earlier.

5. Select the new **DebitBillingException** variable from the variables list and Browse to change its Message definition to **BillingException** as defined in NiceJourney/com/nicejourney/NiceJourneyInterface.wsdl. Click **OK**.

6. Return to the Fault Handler and add an Assign activity and rename it to `prep DebitCard Exception`. Use it to copy the *reason* part of the `DebitCardException` variable to the *detail* part of the `DebitBillingException`.
7. Finally add control links so that the Java snippet connects to the Assign which then connects to the Throw as shown in Figure 8-44.

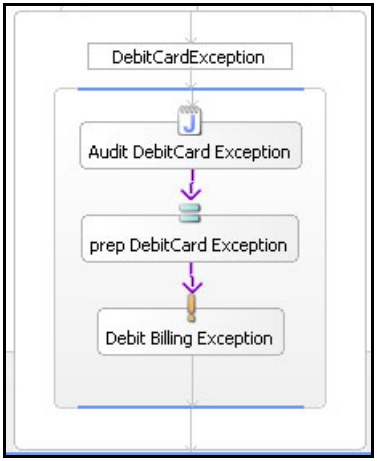


Figure 8-44 Completed Fault Handler for faults occuring in the Debit Card scope

8. Save and close the NiceJourney.bpel process.

Bill Customer

There are now Throw activities in three places within the Bill Customer Switch activity:

- Fault Handler for the Credit Card
- Fault Handler for the Debit Card
- Otherwise case of Bill Customer

Each one of these throw activites throws a fault of message type: `BillingException`, although each one has its own fault variable of this type. We will now create a Fault Handler on the Bill Customer activity to catch all of these faults and handle the scenario of a billing failure in general.

Adding the Fault Handler

1. Select the **Bill Customer** activity and add a Fault Handler to it.
2. Display the Fault Handler and select **Add Catch.**

Note: The Fault Handler appears to the right of the activity and may therefore be hidden from view. Scroll to the right of the canvas to see it.

3. Set the fault properties for the Catch as follows:

   Fault type: `User-defined`

   Namespace: `http://nicejourney.com/NiceJourney`

   Fault Name: `BillingException`

4. Click **New ...** and name the new variable `GeneralBillingException`.
5. Select the new **GeneralBillingException** variable from the variables list and Browse to change its Message definition to **BillingException** as defined in NiceJourney/com/nicejourney/NiceJourneyInterface.wsdl. Click **OK**.

Completing the Fault Handler

We will use a very similar style Fault Handler to the ones already created. Firstly we will audit the failure using a simple Java snippet and then we will throw the exception to a higher scope. It could be possible that the Fault Handler would have the ability to handle the error and allow the process to continue. However, in our process a Billing Exception is regarded as a fatal error and so must be re-thrown as a process failure fault.

We will then use this fault later in the process development to trigger automatic compensation which will undo all of the work that had already been committed when the failure occured.

1. Add a Java snippet inside the Fault Handler, naming it `Audit Billing Exception` and adding the following code for its Implementation:

   ```
   System.out.println("BILLING EXCEPTION");
   ```

   This will log the billing exception for auditing purposes.

2. Add a Throw activity and rename it to `NiceJourney Failure`.
3. Change its Fault properties to:

   Fault type: `User-defined`

   Namespace: `http://nicejourney.com/NiceJourney`

   Fault Name: `NiceJourneyFailure`

4. Click **New ...** and name the new variable `NiceJourneyException`.
5. Select the new **NiceJourneyException** variable from the variables list and Browse to change its Message definition to **NiceJourneyException** as defined in NiceJourney/com/nicejourney/NiceJourneyInterface.wsdl. Click **OK**.

6. Return to the Fault Handler and add an Assign activity and rename it to `prep NiceJourney Exception`. Use it to make the following three copies between parts of variables:

   GeneralBillingException: detail → NiceJourneyException: reason

   NiceJourneyInput: firstName → NiceJourneyException: firstName

   NiceJourneyInput: lastName → NiceJourneyException: lastName

7. Finally add control links so that the Java snippet connects to the Assign which then connects to the throw activity as shown in Figure 8-45.

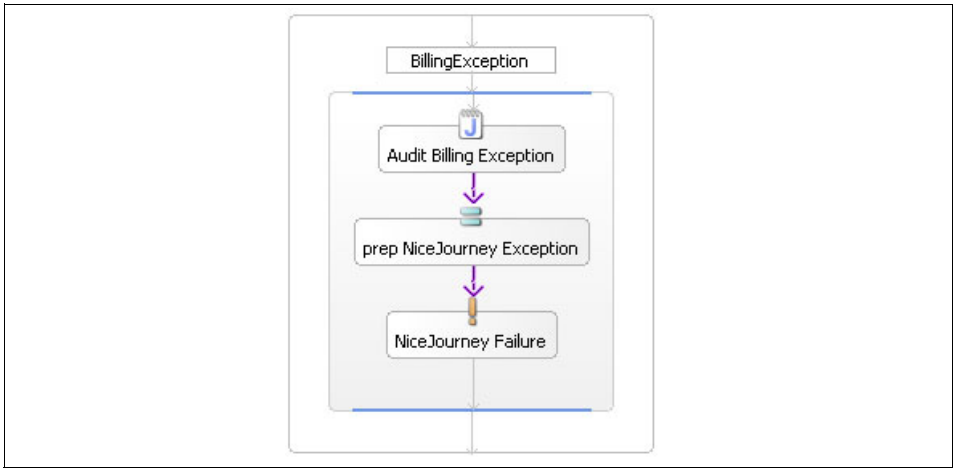


Figure 8-45 Completed Fault Handler for Bill Customer scope

8. Save and close the NiceJourney.bpel process.

NiceJourney process

There is one throw activity inside the Fault Handler for the Bill Customer Switch activity. We want to catch this fault when it is thrown. Currently there are no Fault Handlers defined that will catch it so we need to create one at a higher scope. We will use the top level Fault Handler for the entire process to do this.

Adding the Fault Handler

1. Select the root of the entire process which is labeled NiceJourney and add a Fault Handler to it, either using the icon or the pop-up menu as before.
2. Display the Fault Handler and select **Add Catch.**
3. Set the fault properties for the Catch as follows:

   Fault type: `User-defined`

   Namespace: `http://nicejourney.com/NiceJourney`

Fault Name: `NiceJourneyFailure`

4. Click **New ...** and name the new variable `NiceJourneyFailed`.
5. Select the new **NiceJourneyFailed** variable from the variables list and Browse to change its Message definition to **NiceJourneyException** as defined in NiceJourney/com/nicejourney/NiceJourneyInterface.wsdl. Click **OK**.

Completing the Fault Handler

If a fault reaches the top level of the process then we need to consider what the final action should be. Because the NiceJourney process is an interruptible one, if a failure occurs then it is likely that some work will already have been committed. In this scenario we want to use compensation to undo those changes.

We will configure compensation in the next section but before doing this we must add a manual compensation for the hotel booking. This is because the hotel booking was done through a Staff activity and compensation does not support staff activities.

We will use this Fault Handler to manually undo the hotel booking. The Fault Handler will not do anything else and therefore the fault will exit the handler and cause the complete process to end in a fault. This will be the trigger for the compensation of all the other steps.

1. First import the implementation of the compensating service. In our case we used a very simple Java class to demonstrate the functionality.
2. Import **Compensator.java** from the ComplexProcess directory of the additional material and into folder **NiceJourney/com/nicejourney/**
3. Drag and drop the **Compensator.java** file from the services view onto the NiceJourney.bpel editor canvas.

   Notice that a special type of partner link is created with the Java symbol used as the icon. This indicates that this is a partner implemented as a Java program. Select this partner link and note that the implementation is the Java class, in this case Compensator. Locate the generated WSDL definition of the partner in NiceJourney/com/nicejourney/comnicejourneyCompensatorNiceJourneybpel.wsdl.
4. Add an Invoke activity inside the new Fault Handler and rename it to `Cancel Hotel`.
5. Select the **Cancel Hotel** activity and go to the implementation editor. Set the Partner Link to **Compensator** and the operation to **cancelHotel.**

6. From the drop-down selection box, set the request variable to **StaffHotelReponse** - we will use the reply from the booking invocation as the input to the cancellation service. This variable contains the reservation ID so will be enough information to cancel the hotel.
7. Create a new Response variable using the **New...** button to ensure that the variable type is set automatically. Name it **CancelHotelResponse**. The Fault Handler is now complete and should look like that shown in Figure 8-46.

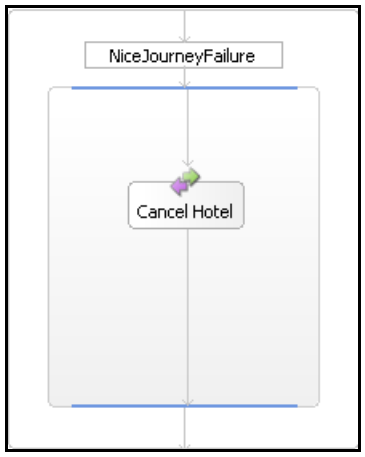


Figure 8-46 Completed Fault Handler for NiceJourney process scope

8. Save and close the NiceJourney.bpel process.

8.7.9 Compensation

The process is now configured so that a payment failure will be handled, logged and then re-thrown to the parent scope for the whole process. The Fault Handler at this point will cancel the hotel booking.

Inside this Fault Handler we do not do anything to end the process gracefully so the fault propogates through, resulting in failure of the overall process. This will trigger any compensation we have configured.

If the payment failure has caused us to end the process in failure then we will want to cancel both the flight and the car (as well as the hotel we already cancelled). In this section we show how to configure the compensation of these two services.

Compensation allows us to select another WSDL described service that is associated with an activity. If that activity has already completed when the process ends in failure, the associated compensation service will be called.

In our case, if the payment fails then the flight and car services will be called in the forward direction already. Compensation will automatically call the associated compensating services to cause a reversal of this work.

Flight compensation

We will configure a simple Java service to perform the flight cancellation that gets triggered by compensation.

1. Locate the `Reserve Flight` sequence and then find the `Invoke Flight Partner` activity inside the sequence.
2. Select the activity and switch to its **Compensation** properties.
3. Change the Partner Link to **FlightBookingSystem**, the operation to **cancelReservation**, and select **FlightBookingRequest** from the drop-down list of input variables.

The compensation is now set to call the service that is implemented by the Flight Booking System Java class, using method cancelReservation.

Car compensation

The cancellation of the car booking is also implemented by a simple Java class that writes to the system log. In a production system this would probably remove some entry from a database.

1. Locate the **Reserve Car** sequence and then find the Invoke Car Partner activity inside the sequence.
2. Select the activity and switch to its **Compensation** properties.
3. Change the Partner Link to **Compensator**, the operation to **cancelCar**, and select **BookCarRequest** from the drop-down list of input variables.
4. Save and close the process.

8.8 Testing

We have now completed a travel agency process that processes an incoming request and books a hotel, car and flight. It then calls payment services that simulate the payment from a debit or credit card. If a failure occurs in the process then this is audited and the compensation and fault handling ensures that all of the bookings are cancelled.

You can test the process by generating the deployment code for NiceJourney.bpel and BookCar.bpel. Then use the business process Web client to initiate the process and interact with the Book Hotel Staff activity.

You may wish to import the final solution directly by using our completed workspace, supplied as a project interchange zip file called NiceJourneyComplete.zip.

8.9 Problem determination and tips

While working with WebSphere Studio Application Developer Integration Edition you may find the following tips and problem determination aids useful.

8.9.1 How to delete generated deployment code

Sometimes you will want to completely delete the generated deployment code for a process. If you have made major changes since the previous generation of deployment code then this can be a good idea because some of old generated code will not be removed upon re-generation.

There are two useful techniques for achieving this quickly. Either

- Delete the entire generated EAR project and its modules, taking care to exclude your services project from the deletion
- Delete each individual generated module project except your services project

Deleting EAR project but preserve services projects

If the generated EAR project only contains deployment code in support of the process then you can delete this entire project and the supporting Web and EJB modules that it contains. You must be careful however to not delete the services projects that contain your business processes.

1. Switch to the J2EE Hierarchy view and expand **Enterprise Applications** to display the EAR project.

   It will be called <*ServiceProjectName*>EAR where <*ServiceProjectName*> is the name of the service project that was used to create this generated code.
2. Select this project and press **Delete**.
3. Select the **Also delete module and utility Java projects** option and make sure both of the options are checked.
4. Click the **Details >>** button to display the list of projects.
5. Carefully de-select the projects that contain your processes. Leave the generated projects checked.
6. Click **OK**.
7. Choose to also delete the contents from the file system to complete the deletion.

8. Re-generate your code as necessary.

Deleting each individual generated project

An alternative method is to delete each of the module projects and EAR project separately. In this method, you have to opt to include each project in your deletion, rather than to exclude each one from a deletion. The end result is the same:

1. Switch to the package explorer.
2. Select (multi-select if you like) each generated project and then click **Delete** for each one (or all in one go).

   The generated projects will be called <*ServiceProjectName*>Web, <*ServiceProjectName*>EJB, <*ServiceProjectName*>EAR where <*ServiceProjectName*> is the name of the service project that was used when generating the code.
3. For each deletion, also select to delete the contents from the file system.

8.9.2 Forgetting to create tables and datasources

When testing the process, it is important to remember to select the **Create Tables and Datasources** option before starting the server. Failure to do this will result in an exception when attempting to execute the process. The console log will show lots of errors about missing tables.

Fix this by stopping the server and choosing the Create Tables and Datasources option (it can only be done with the server stopped). This might fail the first time because the tables will be corrupted by the previous attempt to use the server. If the message is not free of errors, simply repeat the Create Tables and Datasources step. On the second attempt there should be no errors. You can now start the server and test your process.

8.9.3 Type mapping - primitive and complex types

Because WebSphere Process Choreographer uses XML to describe all variable types (either in a separate XSD file or embedded within a WSDL file) then it is important to consider mapping data to and from XML.

Web services development is easier when done ‘top-down’ which means creating the interface first, including the data type definitions in XML. The service is then implemented. When developed bottom up, the implementation is already done and a corresponding Web services interface must be generated to describe it. This means describing the XML data types to represent the implementation. For example, if the implementation is some kind of Java service then it will be

necessary to define the service's method signature in XML types that are equivalent to the Java types.

For a top-down development (XML first) it is necessary to map the XML data types you define into Java types. For a bottom-up approach to developing each service, the Java types must be mapped to XML types.

WebSphere Process Choreographer uses the Web Services Invocation Framework (WSIF) to invoke Web services at runtime. WSIF defines mappings between Java and XML types and these mappings should be used when developing the services.

We recommend to be aware of round-tripping issues. Round-tripping refers to mapping a Java type to its XML type and then back to Java (or XML to Java to XML). For some mappings, the end result is not the same as the start point and we recommend that you avoid using any XML or Java types that do not have round-tripping.

Note: For further information about round-tripping refer to the following series of developerWorks® articles:

http://www-106.ibm.com/developerworks/webservices/library/ws-tip-roundtrip1.html
http://www-106.ibm.com/developerworks/webservices/library/ws-tip-roundtrip2.html
http://www-106.ibm.com/developerworks/webservices/library/ws-tip-roundtrip3.html

We also recommend that you avoid mappings that are different in WSIF than in the emerging JAX-RPC technology. You can refer to the Help in WebSphere Studio Application Developer Integration Edition for information about type mapping issues with WSIF.

A complicated interface to a Web service can require XML complex types which may require particularly sophisticated mappings and care should be taken to try to minimise the complex type usage. As well as potential difficulties in mapping between types, some areas of WebSphere Studio Application Developer Integration Edition do not support complex types in full. For example, a variable that is defined by a message using complex types cannot be fully manipulated by an Assign activity. It also cannot be used within a Correlation Set alias.

9

Chapter 9.

Process choreographer: clients

There are several ways to start/access a process instance in WebSphere Process Choreographer. Depending on the type of client you want to interact with a process instance, you have two choices:

- Developing a standalone client which uses the BPE, J2EE- or SOA API to interact with the process instance.
- Using the business process Web client shipped with WebSphere Studio Application Developer Integration Edition and customizing its look and feel.

9.1 Standalone client

Implementing a standalone client is the most flexible way to access a process instance; furthermore, it provides the highest degree of independence regarding the GUI of your client.

Figure 9-1 shows the different layers involved in accessing/starting a process instance.

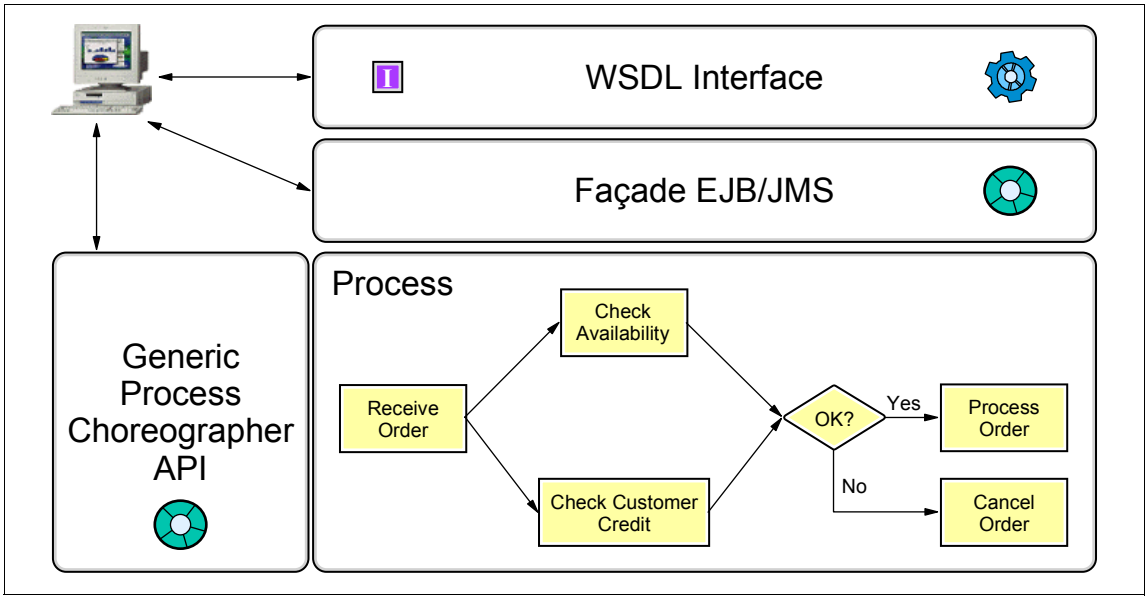


Figure 9-1 Interacting with the business process

Each layer has a different level of abstraction. The most abstract one is the WSDL layer; the Generic Process Choreographer API layer has the lowest level of abstraction. A layer with a higer level of abstraction uses a layer with a lower level of abstraction to interact with the process instance.

A standalone client can also directly use one of these layers to access a process instance.

9.1.1 Invoking a business process using the Process Choreographer API

The Process Choreographer API, also known as BPE API, is the most direct way to interact with a process. As a result of this, clients built on this API have to be changed if the business process is changed, even if the change is a minor one.

The API is public but propriatery, which means that it is not commonly known by J2EE developers; furthermore, once you start using the API, you also need to be familiar with the WSIF API.

The BPE API provides a session bean, called the *BusinessProcess* bean. This session bean exposes several methods that can be used to query the BPE database. You call methods on this session bean to invoke the process with the appropriate message. Note that you have to compose the *message* (instantiate the WSIFMessage object) programmatically prior to invoking the BPE API. A WSIFMessage object can consist of multiple parts. It is complicated to compose the message parts programmatically, but this method of invoking processes provides very tight coupling with the BPE API and much more control. This may or may not be necessary when invoking a Business Process.

Because the BPE API is the most direct way to interact with the process instance, it provides calls which cannot be performed from the more abstract layers above. The documentation (Javadoc) of the BPE API can be found at the WebSphere InfoCenter at:

`http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp`

From there, navigate to **WebSphere Business Integration Server Foundation** → **Reference** → **Javadoc** → **Enterprise Extensions API**.

9.1.2 Invoking a business process using the generated façade EJBs

When the deployed code is generated for the business process, as discussed in “Generating deploy code” on page 177, façade Enterprise Java Beans are generated depending on the chosen binding. These façade Enterprise Java Beans can be used to invoke a process in a common J2EE manner.

Similar to the BPE API, façade Enterprise Java Beans are also a very direct way to interact with a process instance, so the invoker and the business process are tightly coupled. It might be meaningful in some cases to choose this way of invocation to avoid the creation of an internal object model, which takes place using the WSDL layer for invocation. This internal object model, used by WSIF, may result in a performance overhead. Since there is no external access to the process instance, it might be better to call the façade Enterprise Java Beans directly, instead of exposing them as Web services and using the WSDL for the process invocation.

There is a relationship between the type of business process (short running or long running) and the available bindings when generating deploy code. According to the chosen binding, the façade Enterprise Java Beans are generated. These issues are covered in the following sections.

EJB binding for deployed business processes

The EJB binding is the default binding type for short running business processes, but it can also be used for long running business processes.

For this binding type, the façade is implemented as a Session Enterprise Java Bean, available in a J2EE EJB 2.0 project in the workspace of WebSphere Studio Application Developer Integration Edition.

After you have generated the deployment code for the business process using the EJB façade, you should find the Session EJB for your process as shown in Figure 9-2 on page 290, for the NiceJourney process.

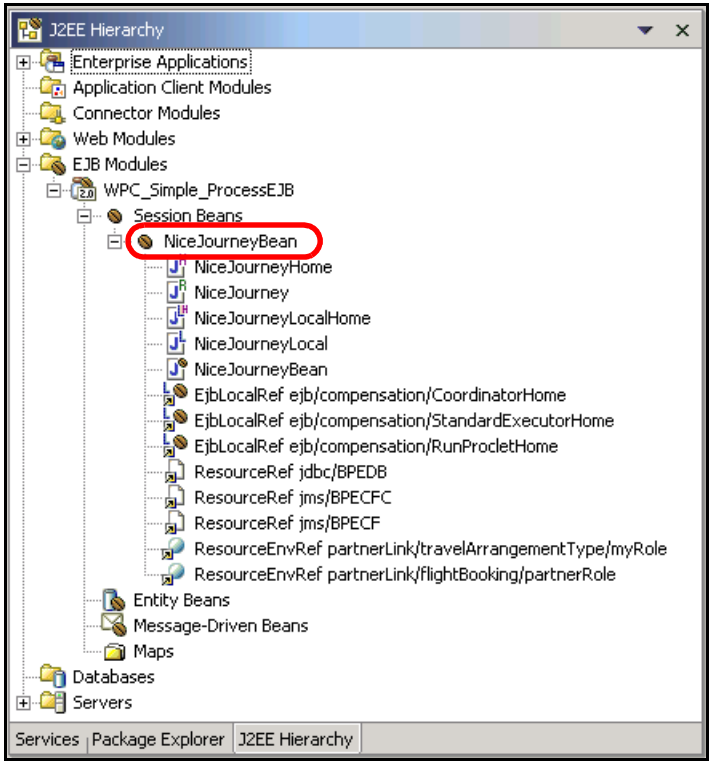


Figure 9-2 J2EE Hierarchy view

Note: If the refresh function (menu) is not working properly in the J2EE Hierarchy view, close the view and reopen it again to see the session bean.

For each receive or Pick activity in the business process, a wrapper method is added to the façade Session Enterprise Java Bean, wrapping all calls to the

business process. It can be accessed directly from other J2EE clients such as other Session Enterprise Java Beans, servlets or J2EE application clients in a common J2EE manner, JNDI lookup and so on.

To test the invocation of the business process over the Session Enterprise Java Bean, the Universal Test Client can be used. It allows the use of the Session Enterprise Java Bean which then interacts with the business process. To start the Universal Test Client, follow the steps below.

1. Add the generated deploy code to an Integration Test Server Environment; for details, refer to “Deploying a process to the WebSphere Test Environment” on page 179.
2. Make sure that on the Configuration tab of the Integration Test Server, the checkbox **Enable universal test client** is selected; this is the default for a new generated Integration Test Server Environment.
3. Start the Integration Test Server Environment.
4. Start the Universal Test Client; right-click the started **Integration Test Server** from the pop-up menu and select the item **Run universal test client**. For more information about how to use the Universal Test Client, refer to the WebSphere Studio Application Developer Integration Edition help by clicking **WebSphere Studio** → **Testing** → **Testing the enterprise service and its deploy code** → **Testing the session bean with the IBM Universal Test Client**.

SOAP binding for deployed business processes

The SOAP binding type is only available for short running business processes.

As for the EJB binding type, for this binding type the façade is implemented as a Session Enterprise Java Bean, available in a J2EE EJB 2.0 project in the workspace of WebSphere Studio Application Developer Integration Edition, as shown in Figure 9-2 on page 290. In addition, a WSDL file is generated containing binding and service information to provide access to the business process by any SOAP compatible client.

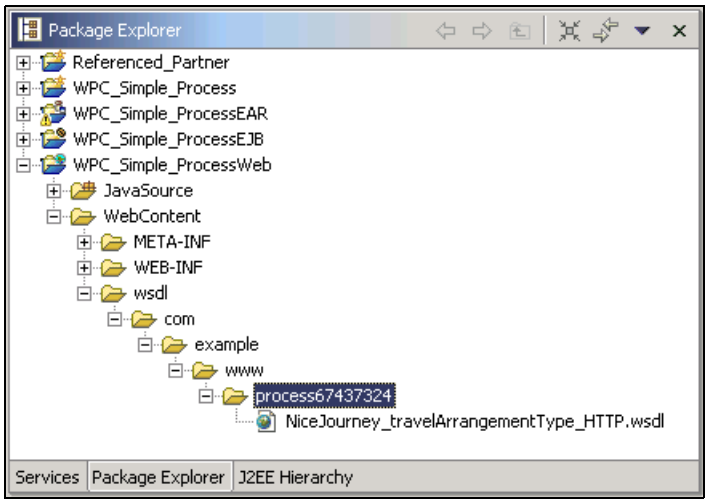


Figure 9-3 Package Explorer view

In fact, a client using the above-mentioned WSDL file will access the façade Session Enterprise Java Bean (through a generic Web Services Router Servlet) which, finally, interacts with the business process. This is the best deploy option to allow non-Java clients to interact with the business process.

To test the invocation of the business process using the SOAP binding, the Web Services Explorer can be used, as follows:

1. Add the generated deploy code to an Integration Test Server Environment, as discussed in “Deploying a process to the WebSphere Test Environment” on page 179.
2. Start the Integration Test Server Environment.
3. Right-click the WSDL file that contains the SOAP binding and service information; see also Figure 9-3 on page 292.
4. In the pop-up menu, select **Web Services** → **Test with Web Services Explorer**.

The Web Services Explorer can now be used to send SOAP messages to the business process, as shown in Figure 9-4.

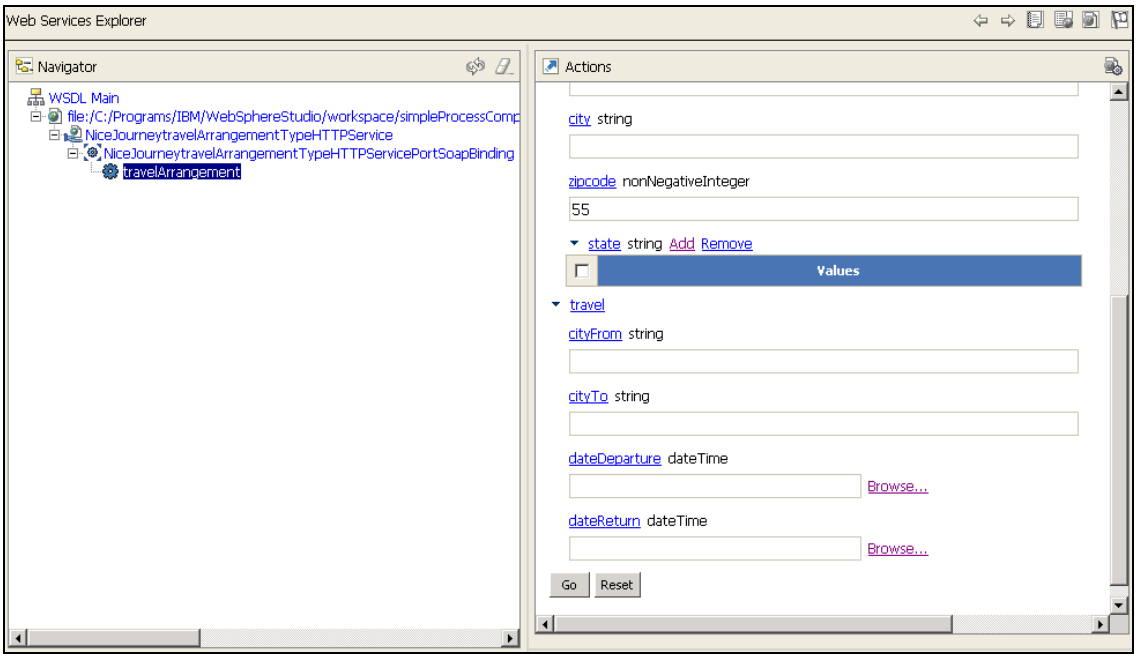


Figure 9-4 Web Services Explorer

JMS binding for deployed business processes

The JMS binding is the default binding type for long running business processes. If the invocation of the long running business processes is a request/response operation, the binding must be of JMS type. A JMS binding can also be generated for a short running process.

For this binding type, the façade is implemented as a Session Enterprise Java Bean, and in case of a long running business process, an additional Entity Enterprise Java Bean is generated. This will be used to store states of the long running business process in a database.

In version 5.0.x of WebSphere Studio Application Developer Integration Edition, there is also a Message-Driven Bean generated to invoke the process triggered by a message on a queue. In V5.1, this Message-Driven Bean is no longer generated; instead, a generic WebSphere Process Choreographer Message-Driven Bean can be used to invoke the business process with a message on a queue. For more details, refer to the InfoCenter at:

`http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp`

From there, navigate to **WebSphere Business Integration Server Foundation** → **Developing** → **Applications** → **Process choreographer** →

Developing applications for BPEL-based process → Accessing the process choreographer JMS interface.

The default values for the JMS destination is jms/BPEIntQueue; for the connection factory, it is jms/BPECF.

9.1.3 Invoking a business process as a Web service using the generated proxy

When the deployed code is generated for the business process, as discussed in “Generating deploy code” on page 177, a WSDL file will be generated according to the chosen binding type. This WSDL file can be used to invoke the business process as a Web service. To do this, a service proxy can be generated from it.

If there is a good deal of external access to the process instance, this is the recommended way to call a business process as a Web service, because the invoker and the business process are loose coupled.

Generating a service proxy from a WSDL file which contains binding and service information is very similar for all available bindings of a business process. The differences will be pointed out in the next three sections.

EJB binding proxy

If the deployed code for the BPEL process was generated for an EJB binding (refer to “EJB binding for deployed business processes” on page 290), a WSDL file for this binding will be generated in the same package as the BPEL process. This WSDL file can be used to generate a service proxy to access the business process, as shown in Figure 9-5.

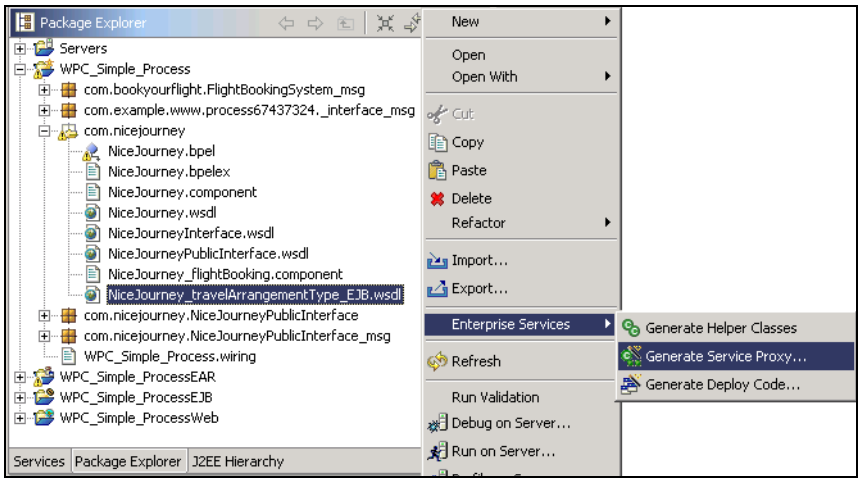


Figure 9-5 Generate Service Proxy

If the **Generate Service Proxy** item is selected, you will be guided through three dialogs to define:

1. The type of proxy to generate: Web Services Invocation Framework (WSIF) or Java API for XML-based RPC (JAX-RPC).

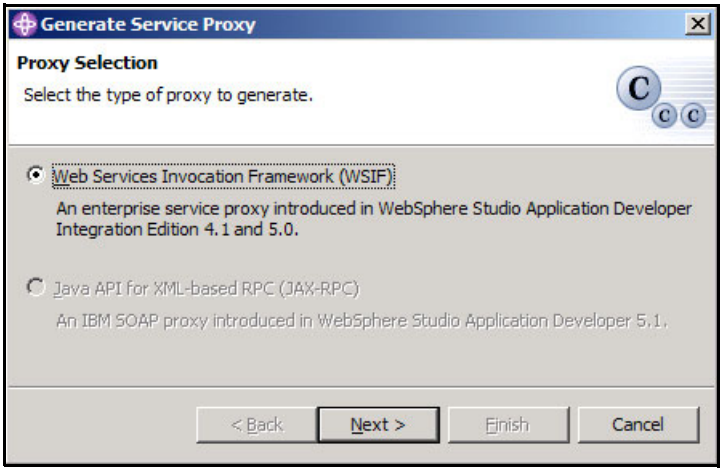


Figure 9-6 Proxy selection

Important: The JAX-RPC type proxy is not available; it is disabled on the panel because the current version of JAX-RPC does not support RMI/IIOP transport.

2. The service for which to generate the proxy and coordinates of the proxy class.

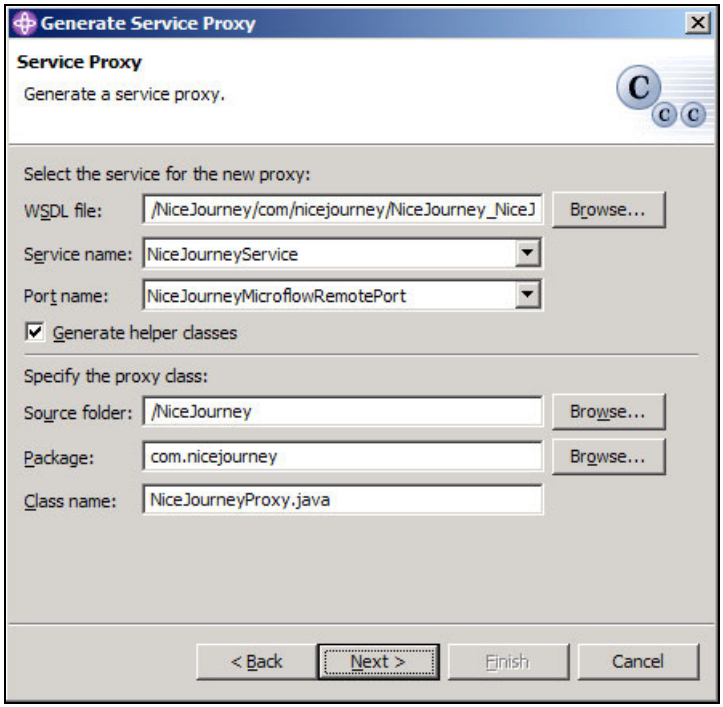


Figure 9-7 Service proxy

3. The proxy style: Client stub or Command bean.

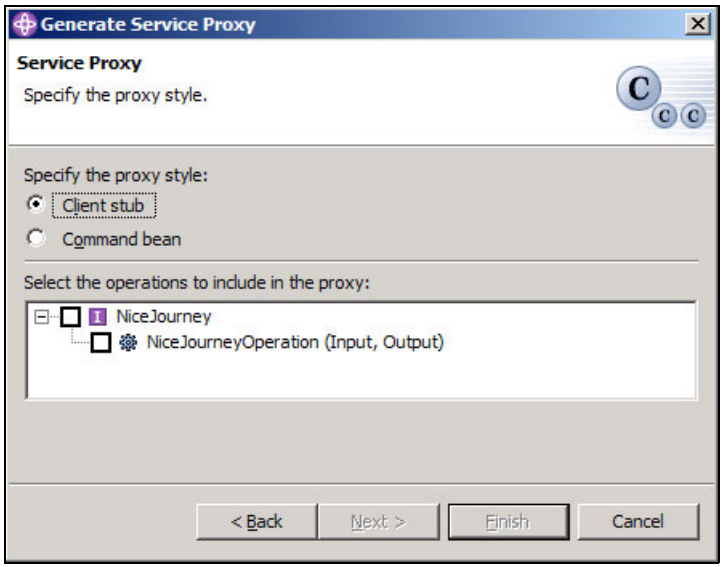


Figure 9-8 Service proxy

SOAP binding proxy

If the deployed code for the BPEL process was generated for a SOAP binding (refer to section “SOAP binding for deployed business processes” on page 291), there are several options available on the type of SOAP binding.

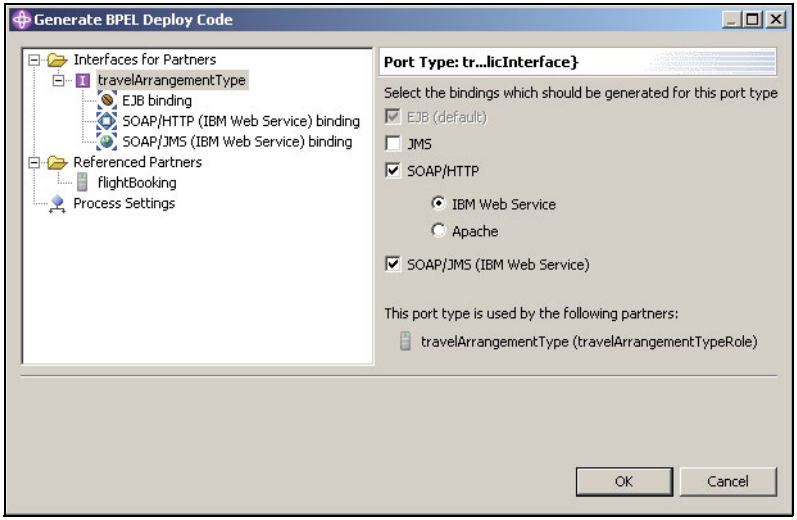


Figure 9-9 Generate BPEL Deploy Code

For each option, a WSDL file for the binding will be generated. This WSDL file can be used to generate a service proxy to access the business process; simply right-click it.

For the SOAP bindings, the generated proxies can be of type Web Services Invocation Framework (WSIF) or Java API for XML-based RPC (JAX-RPC).

JMS binding proxy

If the deployed code for the BPEL process was generated for a JMS binding (refer to section “JMS binding for deployed business processes” on page 293), a WSDL file for this binding will be generated in the same package as the BPEL process. This WSDL file can be used to generate a service proxy to access the business process; simply right-click it.

For a JMS binding, only a proxy of type Web Services Invocation Framework (WSIF) can be generated.

9.2 Web client

This section provides details about the Web client interface for WebSphere Process Choreographer. The Web client is a Web application, installed by default together with the Business Process Container.

The BPEL Editor provides interfaces to customize certain pages for the Web client. The following is a list of activities with customizable pages:

- Staff
- Receive
- Reply
- Pick

We will show only the Staff activity customization; the rest of the activities are very similar in terms of their pages and customization.

The process itself has customizable pages; these are discussed in the following section.

9.2.1 Customizing process pages

When you create customized or additional pages for your Web client, you need to place them in a Web project. There are two options available:

- Import the Web client Enterprise Application in WebSphere Studio Application Developer Integration Edition, then customize the pages and add new ones.

This option provides the most flexibility but requires the most knowledge about the default Web client.

You can import the Web client from the <WebSphere_Studio_root>/runtimes/ee_v51/installableApps directory; the file is called processportal.ear.

This option is beyond the scope of this redbook.

- Create a new Web project and use it together with the default Web client Enterprise Application.

  This option is useful if you only want to make minor changes to the Web client or if you only want to create customized pages for the activities.

  We will discuss this option in the following sections.

Perform the following steps to create your own customized pages for the Web client.

1. Create a new Web project and provide a name, for example: `NJWebClient`. Add the new Web project to the Enterprise Application where your process runs, if it is not added automatically.
2. Fix the context root for your Web application to reflect the one you want to set up to serve the custom pages for the Web client.
3. Copy the following JAR files under the WEB-INF/lib folder:
   - jaxen-full.jar
   - jstl.jar
   - processportal.jar
   - saxpath.jar
   - standard.jar

   You can get the JAR files from any installed Web client, for example: <WebSphere_Studio_root>/runtimes/ee_v51/installedApps/localhost/BPEWebClient_localhost_server1.ear/processportal.war.
4. Create a *tld* folder under WEB-INF and copy the webclient.tld file into the directory.

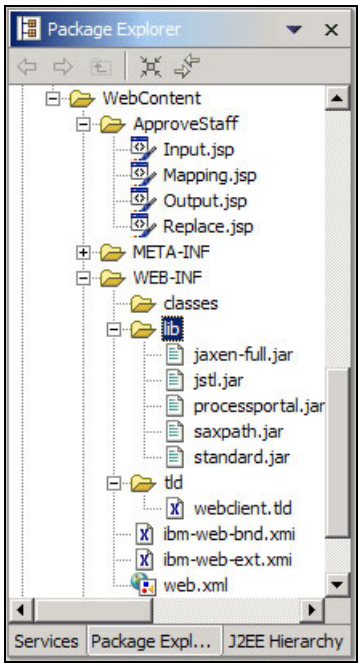


Figure 9-10 WEB-INF folder

5. Open the web.xml file and click **Add** to add the WebContent/WEB-INF/tld/webclient.tld JSP tag library under the **References** → **JSP tag libraries** tab.

   You can get the TLD file from any installed Web client, for example: <WebSphere_Studio_root>/runtimes/ee_v51/installedApps/localhost/BPEWebClient_localhost_server1.ear/processportal.war.

6. Create your custom pages. It is recommended that you create subfolders for each set of custom page, for example ApproveStaff for the Approve Staff activity.

7. We will create three JSPs in the following section: Input.jsp, Output.jsp, Mapping.jsp. Create all three of them under the ApproveStaff folder.

9.2.2 Staff activity

In WebSphere Studio Application Developer Integration Edition V5.1, it is possible to define specific JSPs for staff activities. These JSPs can provide additional functionality to the WebSphere Web Client by including upfront data validation (performed in the Mapping JSP) and the ability to provide a more descriptive user interface.

To implement user-defined JSPs for the Staff Activity, you will create these JSPs:

- **InputMessageJSP** - to display an activity Input Message.
- **OutputMessageJSP** - to contain the user data and wrap it for the business process engine.
- **MessageMappingJSP** - this message-mapping JSP will receive the user data, wrap it in an appropriate message object and then forward the message to the business process container.

In this sample, we have extended the NiceJourney Simple Process as described in Chapter 7, “Process choreographer: developing a simple process” on page 135 by adding a Staff activity. In this activity, the staff member will decide whether or not to approve the customer’s request to book a flight. This scenario aims to show how customizable JSPs can be implemented.

The input message for the approval activity has only one part:

- question (xsd:string)

The output message for the approval activity has two parts:

- approve (xsd:boolean): represents the approval, `true` (approved) or `false` (rejected)
- notes (xsd:string): represents the additional notes provided by the approver

Note: The code presented can be used for any other approval scenario in other processes with the necessary customizations.

The following figure represents a sample process to show how the approval Staff activity is set up in our sample.

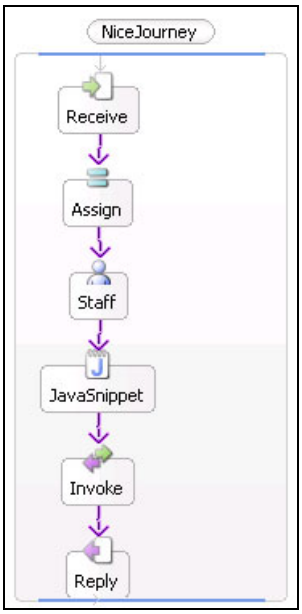


Figure 9-11 The NiceJourney process with an additional Staff activity

Note: The NiceJourney BPEL process is now a long-running/interruptible process due to the inclusion of the Staff activity.

InputMessageJSP

The Web client will invoke the Staff activity input JSP each time it needs to display the data this activity received when it started. This provides the owner of the activity with enough information to progress and complete this activity.

In our sample, we will replace the approve text input field with a checkbox; instead of typing `true` or `false`, selecting the checkbox will implement the approval.

Example 9-1 Input.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
   language="java"
   import="com.ibm.bpe.api.*,
   com.ibm.bpe.client.MessageUtilities,
   org.apache.wsif.WSIFMessage"
%>
<%@ taglib uri="http://portal.bpe.ibm.com/Taglib" prefix="bpe"%>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
```

```
<bpe:InitContext/>
<c:set var="activity" value="${context[constants.JSP_ACTIVITY]}" />
<%
   //initializing the message object
   BusinessProcessService bean =
MessageUtilities.getBusinessProcessService(request);
   String aiid = request.getParameter("WF_AIID");
   ClientObjectWrapper messageWrapper = bean.getInputMessage(aiid);
   WSIFMessage message = (WSIFMessage) messageWrapper.getObject();
%>

<c:choose>
   <c:when test="${activity.executionState ne
constants.activity.STATE_FINISHED}">
      <p>The question is: <%= message.getObjectPart("question") %></p>
   </c:when>
   <c:otherwise>
      <p>The question was: <%= message.getObjectPart("question") %></p>
   </c:otherwise>
</c:choose>
```

As you can see in the code, two cases are described. The first is when the process is not finished; in that case, the client should show the current input variables. The second is when the process has finished and the client shows the client variables from the time when the activity finished.

OutputMessageJSP

In the sample Staff activity, the staff member must approve/reject the request and can provide additional notes if required.

Depending on the state of the current activity, the JSP displays the activity output message in different ways.

- When the activity has been claimed, it displays an input form so approve/reject data can be entered.
- After the activity has finished, the option that has been chosen is presented as text, for example: `The claim has been approved`.
- Otherwise, the activity just waits for an authorized user to claim it.

Example 9-2 Output.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
   language="java"
   import="com.ibm.bpe.api.*,
   com.ibm.bpe.client.MessageUtilities,
   org.apache.wsif.WSIFMessage"
```

```
%>
<%@ taglib uri="http://portal.bpe.ibm.com/Taglib" prefix="bpe"%>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c"%>

<bpe:InitContext/>
<c:set var="activity" value="${context[constants.JSP_ACTIVITY]}" />

<c:choose>
   <c:when test="${activity.executionState eq
constants.activity.STATE_CLAIMED}">
<%
       String approve = "";
       String notes = "";
       if (MessageUtilities.getValidationError(request) != null) {
          approve = request.getParameter("approve");
          notes = request.getParameter("notes");
       }
%>
       <p>I approve this claim: <input type="checkbox" name="approve"
value="on" <%= (approve.equals("on") ? "checked" : "") %> > (check the box if
approved). </p>
       <p>Notes: <input type="text" name="notes" value="<%= notes%>"></p>
   </c:when>
   <c:when test="${activity.executionState eq
constants.activity.STATE_FINISHED}">
<%
       BusinessProcessService bean =
MessageUtilities.getBusinessProcessService(request);
       String aiid = request.getParameter("WF_AIID");
       ClientObjectWrapper messageWrapper = bean.getOutputMessage(aiid);
       WSIFMessage message = (WSIFMessage) messageWrapper.getObject();
       boolean approve=false;
       if(message.getBooleanPart("approve")) approve=true;
       String notes=(String)message.getObjectPart("notes");
%>
       <p>The claim was <strong><%= ( approve ? "approved" : "declined")
%></strong>.</p>
       <p>Notes: <%=notes%></p>
   </c:when>
   <c:otherwise>
       <p>You have to claim the activity.</p>
   </c:otherwise>
</c:choose>
```

MessageMappingJSP

The MessageMapping JSP ensures that the required input data is supplied and checks the data type before it is forwarded to the business process container.

The sample MessageMapping JSP gets the client responses and defines the message that is sent to the business process container.

Example 9-3 Mapping.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
   language="java"
   import="com.ibm.bpe.api.*,
   com.ibm.bpe.client.MessageUtilities,
   org.apache.wsif.WSIFMessage,
   org.apache.wsif.base.WSIFDefaultMessage"
%>
<%
try {
   WSIFMessage message = new WSIFDefaultMessage();
   // set approved part of message...
   if(request.getParameter("approve")!=null &&
request.getParameter("approve").equals("on")) message.setBooleanPart("approve",
true);
       else message.setBooleanPart("approve", false);
   if (!request.getParameter("notes").trim().equals("")) {
      //this is how we assign String object to the message
      String notesMsg=(String)request.getParameter("notes");
      message.setObjectPart("notes", notesMsg);
   }
   // forward message to controller...
   ClientObjectWrapper messageWrapper = new ClientObjectWrapper(message);
   MessageUtilities.forwardMessageToController(request, response,
messageWrapper, null, null);
} catch (Exception e) {
   // problem creating message...
   MessageUtilities.validationFailed(request, "You must enter valid values!");
   MessageUtilities.forwardMessageToController(request, response, null, null,
null);
}
%>
```

Adding the user-defined JSPs to the Web Client

To use these JSPs instead of the standard WebSphere Web Client, follow these steps:

1. Open the NiceJourney.bpel and select the **Staff** activity.
2. Select the **Client** tab and in the Web Client settings, in the InputMessageJSP row, click the **...** button in the Value column. Navigate to /ApproveStaff/Input.jsp and click **OK**.
3. Add Mapping.jsp and Output.jsp as per the previous step and save the file. See Figure 9-12 on page 306 for a screenshot of the Client area.

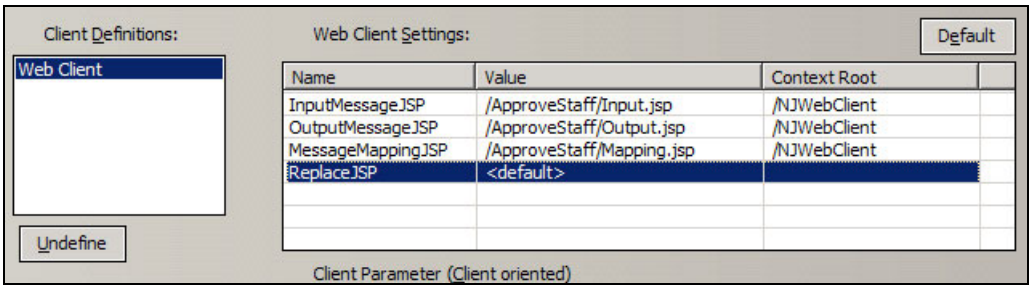


Figure 9-12 Adding user-defined JSPs to the Staff activity

Results

After creating a test server and running the process, the user-defined JSPs will appear as follows.

1. After creating a new instance and opening the activity:

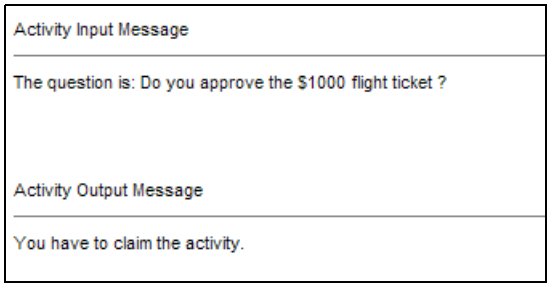


Figure 9-13 New instance

2. After claiming the activity:

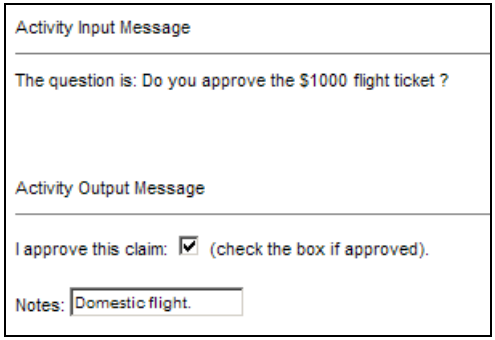


Figure 9-14 Submitting the activity

3. After completing the activity, the finished process details for the Staff activity.

Figure 9-15 Checking the finished process activity

9.2.3 More information about Web Client customization

You can find more information about Web Client customization in the following ways:

- WebSphere Studio Application Developer Integration Edition help can be found by clicking **WebSphere Studio** → **Developing** → **Processes** → **The Process editor (BPEL)** → **Claims handling**. This tutorial comes with sample code which will help you to understand the client code in more detail.
- You can import the default Web client application, processportal.ear, from the <WebSphere_Studio_root>/runtimes/ee_v51/installableApps location, then inspect the code to learn further details.
- The IBM developerWorks article at:

  `http://www-106.ibm.com/developerworks/websphere/library/techarticles/wasid/WPC_Client1/WPC_Client1.html`

  is a three-part article about Web Client customization for WebSphere Enterprise V5.

10

Common Event Infrastructure

The Common Event Infrastructure is an IBM technology which allows events to be created, stored, routed and retrieved by applications. There are many applications within the CEI, ranging from consistent logging across multiple platforms to complex autonomic infrastructures.

WebSphere Business Integration Server Foundation is one of the first implementations of this technology, providing a method of emitting, storing and handling events which can be used within applications.

Important: This chapter relates to the Technology Preview of the CEI which ships with WebSphere Business Integration Server Foundation V5.1.

10.1 Introduction

The Common Event Infrastructure is a set of modular components which provides simple event management. CEI has been designed as a core technology that will be integrated into many other IBM products in the medium term. WebSphere Business Integration Server Foundation is one of the first application platforms to provide an implementation of this core technology.

The fundamental concept behind the CEI is that applications or middleware components create events whenever they perform some processing that could be of relevance to an external application. The event contains information relating to event identification, timing and other details.

The component creating the event object is called the *event source*. The event object is passed to the event infrastructure, whose role is to forward the event on to *event consumers*. These are other applications which have expressed an interest in the event. The event infrastructure may also store the event object in a database for later retrieval.

Figure 10-1 shows the activities that are supported by the CEI.

Note: The Technology Preview version of CEI does not support the event distribution. This will only be available in the full version of this technology in a future release of WebSphere Business Integration Server Foundation.

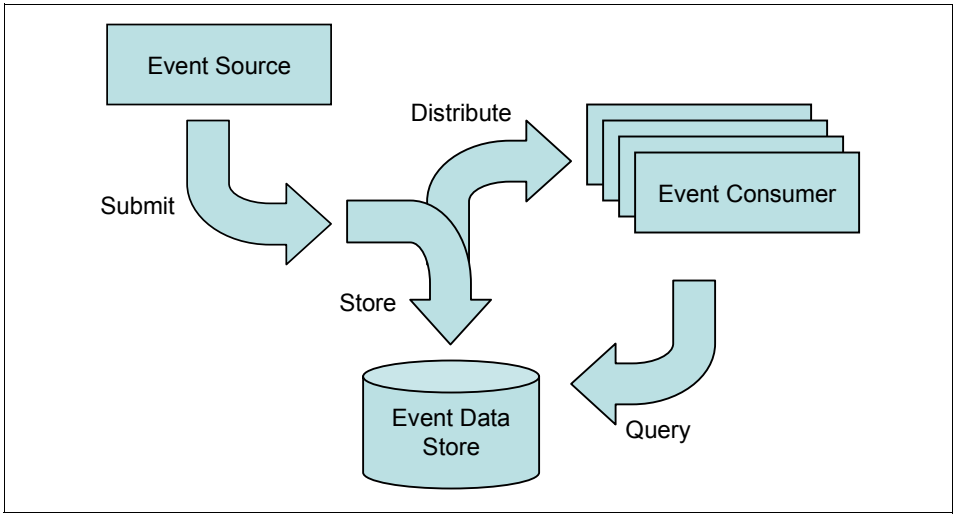


Figure 10-1 Common Event Infrastructure activities

Event structure

The standard structure used for event objects is the Common Base Event (CBE), which is part of the IBM Autonomic Computing Toolkit. The CBE standard defines a set of common fields, the values they can take and the meaning of these values. Further details of the CBE and the IBM Autonomic Toolkit can be found in:

- The IBM Redbook *A Practical Guide to the IBM Autonomic Computing Toolkit*, SG24-6635
- The IBM developerWorks article *Standardize messages with the Common Base Event model*, found at:

  `http://www.ibm.com/developerworks/autonomic/`
- *Specification: Common Base Event*, from IBM developerWorks, found at:

  `http://www.ibm.com/developerworks/webservices/library/ws-cbe/`

CEI implementation

The InfoCenter has detailed information about the Common Event Infrastructure. Review the information by clicking **WebSphere Business Integration Server Foundation** → **All topics by feature** → **Application** → **Application Services** → **Working with the Event Programming Model in WebSphere**.

WebSphere Business Integration Server Foundation provides support for both the CEI client and server components.

- Client support is provided by three mechanisms. CBEs can be created:
  - Explicitly, with a Java API
  - By configuring deployment descriptors

    Note: The deployment descriptor method is not available in the technology preview, and so is not covered in detail in this redbook.

  - By configuring activities within a BPEL4WS process

    These are passed to the State Observer Plugin (SOP) which uses the CEI exploitation layer to populate the CBE.
- Server components include:
  - A database to store the CBEs

    Note: The only supported database for the Technology Preview is Cloudscape.

    With V5.1.1, DB2 and Oracle will also be supported.

– An application to handle the incoming events

  The application can store the events in a database, forward the events to another CEI server or do both.

  Note: The Technology Preview only supports a single-server instance of the CEI application.

– A sample Web application to examine CBEs in the database

10.2 Sample scenario

We will use the CEI to add an audit capability to the simple process described in Chapter 7, “Process choreographer: developing a simple process” on page 135. This audit capability will be implemented as a custom event containing some relevant data from the business context. The event we wish to audit is the receipt of a new travel request. The audit will be triggered based on content of the input message.

Note: While the current Technology Preview of the CEI capability is not suitable for use as a business audit mechanism, future enhancements should make this technology suitable for such an application.

The audit function introduced in the sample application does not implement or use any audit framework or environment; it is solely an example of using CEI.

This will demonstrate the capabilities of the CEI Technology Preview by:

- Reviewing the events generated by a running process
- Using the Java API to create a custom event within a process

Note: The third method of generating events, by using deployment descriptor metadata, is not provided as part of the technology preview.

The programming model for Java applications of the technology preview will not be continued in future versions of WebSphere Business Integration Server Foundation. In this redbook, we have added notes to point out wherever examples will not be applicable for future versions.

Development

This section will demonstrate the different ways of creating CBEs within the CEI. It will show how to:

- Configure a process to report events
- Create new custom events with the Java API

The section starts with the necessary steps you have to perform in order to set up the development environment for CEI development. It continues by enabling the built-in capabilities of the product to report relevant events from within a BPEL4WS process. It then shows how to extend the process to create an explicit event.

Configuration

CEI requires additional configuration in the runtime environment of the WebSphere Business Integration Server Foundation. The configuration section will outline the steps required to perform this task.

Unit test

The CEI is not supported within the Universal Test Environment, so any test activity has to take place in a runtime environment.

Assembly

We are going to use the simple process to introduce CEI. There are no specific assembly tasks to be performed for CEI. For the simple process assembly, follow the directions given in Chapter 7, “Process choreographer: developing a simple process” on page 135.

Deployment

We will use the simple process to introduce CEI. There are no specific deployment tasks to be performed for CEI. For the simple process deployment, follow the indications in Chapter 7, “Process choreographer: developing a simple process” on page 135.

Testing

We will run the enhanced process and review the results.

10.3 Development

This section outlines the two methods of generating events currently provided in the WebSphere Business Integration Server Foundation implementation of CEI.

10.3.1 Setting up the development environment

Before development can start, the development environment must be customized. The steps to do so are outlined in the following section.

Importing the simple process project

We will be basing the rest of the chapter on the simple process developed in Chapter 7, “Process choreographer: developing a simple process” on page 135. Before continuing, you should set up the project workspace.

1. Launch WebSphere Studio Application Developer Integration Edition with a new workspace and make sure you enable server targeting support.
2. Follow the instructions from “Project Interchange archive import/export” on page 562to import the simple process project, simpleProcess.pi.zip, from the additional material.
3. Switch to the Business Integration perspective and rename the project to distinguish this solution from other projects. Right-click **WPC_Simple_Process** and select **Refactor** → **Rename**. Enter `WPC_CEI_demo` and click **OK**.

Preparing the project environment

Note: These steps are required because CEI is a technical preview. It is not currently officially supported in WebSphere Studio Application Developer Integration Edition.

The classpath must be altered to allow the project to be built successfully.

1. Create a directory on your development workstation called C:\cei_libs.
2. Locate the files events-client.jar and events-consumer.jar on the WebSphere Business Integration Server Foundation server which you set up earlier. You will find these in the *<WebSphere_root>*`\lib` directory. Copy these files into C:\cei_libs.
3. Right-click the **WPC_CEI_demo** project folder which contains the NiceJourney.bpel file and select the **Properties** menu item.
4. Within the properties window that is presented, select the **Java Build Path** menu item.
5. Select the **Libraries** tab.
6. Click the **Add External JARs...** button on the Libraries tab.
7. Navigate to C:\cei_libs.

8. Select both **events-consumer.jar** and **events-client.jar**. Click **Open**. YOu will be taken back to the **Libraries** tab.
9. Click the **Add variable...** button on the Libraries tab.
10. Select the **WAS_EE_V51** variable in the list and click **Extend**.
11. Select both **lib\common.jar** and **lib\ecore.jar** in the window. Click **OK**.
12. You should be presented with a screen similar to Figure 10-2.

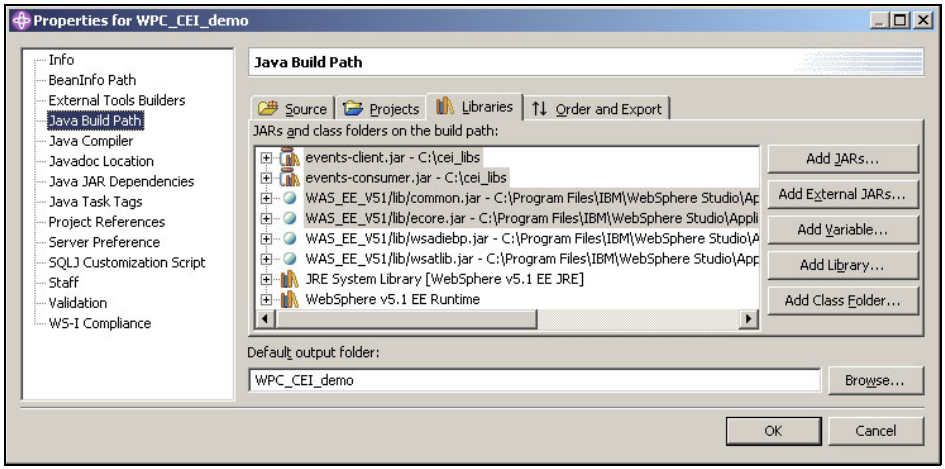


Figure 10-2 Service project Java build classpath

10.3.2 Configuring a process to report events

Each activity within a process, including the process itself, can be set to report relevant events to the Common Event Infrastructure. This is performed using the Business Relevant checkbox on the Server tab of the details area. Figure 10-3 on page 316 shows this flag.

Review the process to see which events are marked as business relevant.

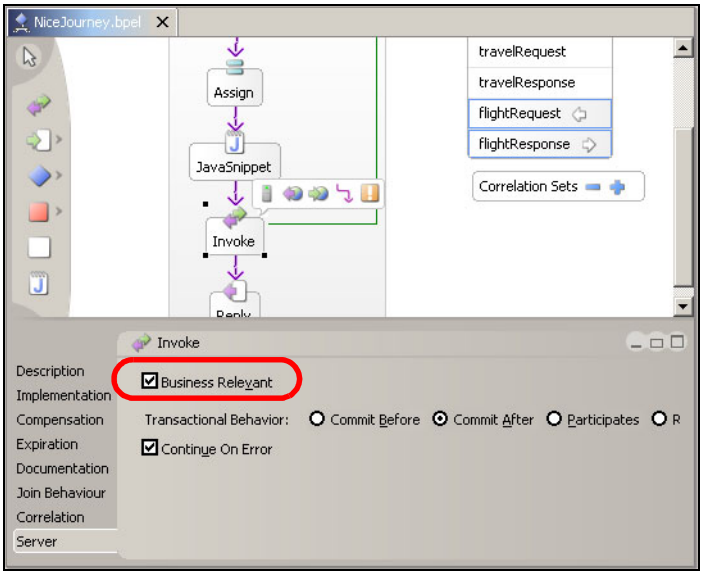


Figure 10-3 The Business Relevant flag

This changes the wpc:businessRelevant attribute of the activity tag in the BPEL4WS XML file. This is shown in Example 10-1.

Example 10-1 BPEL4WS representation of the business relevant flag

```
<invoke inputVariable="flightRequest" name="Invoke"
   operation="getReservation" outputVariable="flightResponse"
   partnerLink="flightBooking" portType="wsdl2:FlightBookingSystem"
   wpc:businessRelevant="yes" wpc:displayName="Invoke" wpc:id="4">
   <target linkName="Link4"/>
   <source linkName="Link1" transitionCondition="DefinedByJavaCode">
      <wpc:transitionCondition>
...................
      </wpc:transitionCondition>
</invoke>
```

Note: `wpc:businessRelevant` is implicitly set to `yes` for certain activities, such as Invoke. To make the `wpc:businessRelevant` setting appear as Example 10-1, you must uncheck and check the **Business Relevant** checkbox, then save the process.

10.3.3 Creating custom events using the Java API

We now have to address creation of a custom audit event, which could be picked up by another application.

Creating new Java classes

We will create two new Java classes: an audit utility which generates a custom event type, and an audit exception handler.

1. Create a new package by right-clicking the **WPC_CEI_demo** project and selecting **New** → **Package**. Name the package `com.nicejourney.utility`.
2. Create a new class by right-clicking the **com.nicejourney.utility** package and selecting **New** → **Class**. Name the class `AuditException`, and select all the defaults.
3. Replace the generated Java code with the following code:

Example 10-2 AuditException.java

```
package com.nicejourney.utility;
public class AuditException extends Exception {
   public AuditException() {
      super();
   }
   public AuditException(String message) {
      super(message);
   }
   public AuditException(String message, Throwable cause) {
      super(message, cause);
   }
   public AuditException(Throwable cause) {
      super(cause);
   }
}
```

4. Create another new class under the com.nicejourney.utility package, this time named `AuditUtil`.
5. Replace the generated code with the following code.

Example 10-3 AuditUtil.java

```
package com.nicejourney.utility;
import javax.naming.*;
import com.ibm.events.*;
import com.ibm.events.cbe.*;
import com.ibm.events.emitter.*;
public class AuditUtil {
   private static EventFactory eventFactory = null;
```

```
    private static EmitterFactory emitterFactory = null;
    public AuditUtil() throws AuditException {
        try {
            initLookup();
        } catch (NamingException ne) {
            throw new AuditException(ne);
        }
    }
    private void initLookup() throws NamingException {
        if (eventFactory == null || emitterFactory == null) {
            Context context = new InitialContext();
            // Locate the event factory and emitter factory
            eventFactory = (EventFactory) context.lookup(
                  "com/ibm/websphere/events/factory");
            emitterFactory = (EmitterFactory) context.lookup(
                  "com/ibm/events/configuration/emitter/Default");
        }
    }
    public void AuditLog(String auditText) throws AuditException {
        try {
            initLookup();
            // Create the event
            CommonBaseEvent commonBaseEvent = eventFactory.createCommonBaseEvent(
                  "NewTravelArrangementReceived");
            commonBaseEvent.addExtendedDataElement("eventDomain", "Business");
            commonBaseEvent.addExtendedDataElement("eventPurpose","Information");
            commonBaseEvent.addExtendedDataElement("customerName", auditText);
            // Write the event
            Emitter emitter = emitterFactory.getEmitter();
            emitter.sendEvent(commonBaseEvent);
        } catch (Exception e) {
            throw new AuditException(e);
        }
    }
}
```

6. Close the open editor windows.

Note: With version 5.1.1 a convenience class ECSEmitter will be provided that will also support a new correlation model. This convenience class will encapsulate all JNDI lookup calls.

Changing the process interface

We wish to pass a flag to the process to define whether or not an audit event is performed. This is achieved by adding a header section into the interface.

1. Edit the NiceJourneyPublicInterface.wsdl file.

2. In the Source editor, add the following code below the second </xsd:element> tag.

```
<xsd:element name="header">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="auditThis" type="xsd:boolean"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

3. Add the following line below the <message name="travelAgencyIn"> tag:

```
<part element="tns:header" name="headerData"/>
```

4. Close and save the file.
5. Select **Project** → **Rebuild All** to rebuild the projects. Check for and resolve any errors.

Extending the BPEL4WS process for Audit

This following section is going to extend the simple process with our custom audit activity.

1. Prepare the NiceJourney process to use the audit utilities.
   a. Open the NiceJourney.bpel process in the BPEL Editor.
   b. Select the **NiceJourney** process lozenge at the top of the screen and change to the Imports tab on the details area.
   c. Add the following line to the bottom of the list of imports in the editor:

```
import com.nicejourney.utility.*;
```

2. Create the Audit activity
   a. Drag a new Java snippet onto the process canvas.
   b. Rename the Java snippet to Audit.
   c. Link this below the Receive activity.
      i. Right-click **Receive** and select **Set Link between Flow Activities**.
      ii. Drag the line to the top of the Audit activity.
3. Change the condition on the link
   a. Select the link between **Receive** and **Audit.**
   b. Select the **Condition** tab in the details area.
   c. Select **Expression** in the drop-down list.
   d. Enter `return getTravelRequest().getHeaderData().getAuditThis();` in the expression box.

4. Add the invocation of the audit code

   a. Select the **Audit** activity and change to the **Implementation** tab

   b. Enter the following Java code:

```
String auditText=getTravelRequest().getCustomerData().getLastName();
try {
   AuditUtil auditUtil=new AuditUtil();
   auditUtil.AuditLog(auditText);
} catch(AuditException ae) {
   System.out.println("Audit error:"+ae);
}
```

   This invokes the Java audit utility that we created earlier. You should end up with a process that resembles Figure 10-4.

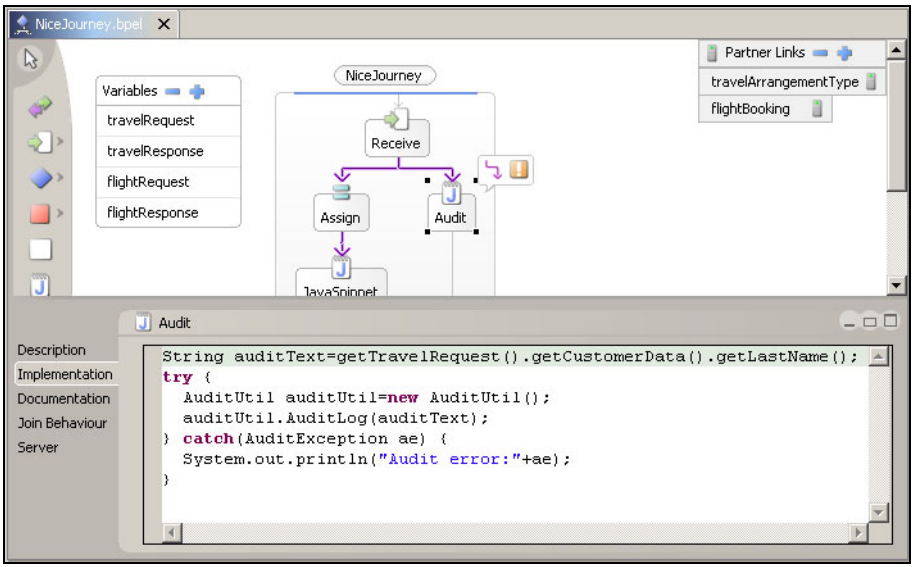


Figure 10-4 The new audit activity

5. Close all the open editors and regenerate the deploy code for the process.

   a. Right-click **NiceJourney.bpel** and select **Enterprise Services** → **Generate Deploy Code**.

   b. Ensure that there are no errors in the generation of the deploy code.

To test the code, you need to deploy the generated EAR file to the server which you set up with the CEI server by following the instructions in 10.4.1, “Configuring CEI in WebSphere Business Integration Server Foundation” on page 321.

10.4 Configuration

CEI insfrastructure needs to be configured on a full WebSphere Business Integration Server Foundation server, as it is not currently supported on the Universal Test Environment.

Note: This following instructions relate to the technology preview CEI, and are likely to change significantly for the final release. Installation is likely to be similar to the WebSphere Process Choreographer component, that is, largely silent during the installation of WebSphere Business Integration Server Foundation, with some configuration required afterwards.

10.4.1 Configuring CEI in WebSphere Business Integration Server Foundation

Assuming that WebSphere Business Integration Server Foundation has been set-up, configuring and validating the CEI infrastructure is a five-step process:

1. Configure the CEI database.
2. Install the CEI application.
3. Start the CEI application.
4. View the contents of the CEI database (optional).
5. Install and run the CBE viewer.

We used the single-server instance of WebSphere Business Integration Server Foundation which was configured in 4.2, “Basic configuration” on page 36. All of the steps are performed as the owner of the WebSphere Business Integration Server Foundation instance, which in this case is the Administrator user.

Note: Our test environment used the following variables:

- <WebSphere_root> = `C:\WebSphere\AppServer`
- node = `m23vnx61`
- server = `server1`

Configuring the CEI database

Note: This will change for the final version, when databases other than Cloudscape are supported.

Perform the following steps to configure the Cloudscape event database:

1. Start the WebSphere Business Integration Server Foundation server (server1).
2. Open a Windows command prompt, change to the *<WebSphere_root>*\event\dbconfig directory

   ```
   cd <WebSphere_root>\event\dbconfig
   ```

   Configure the event database using the supplied batch script and Cloudscape response file, as follows:

   ```
   config_event_database.bat .\CloudscapeResponseFile.txt
   ```

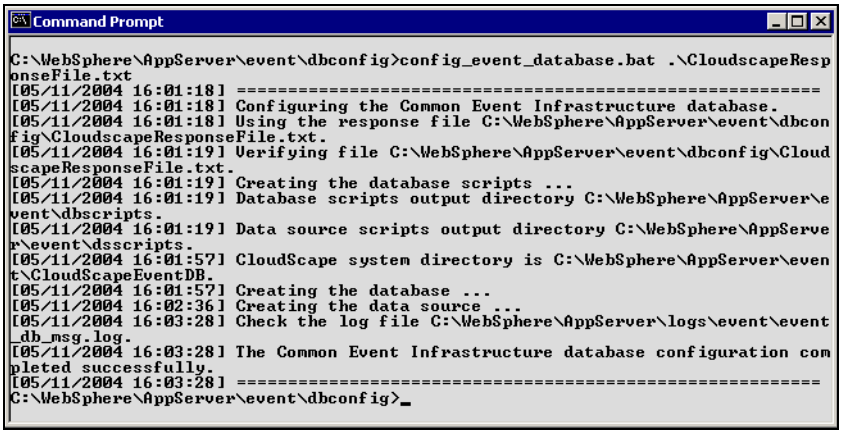


Figure 10-5 Configure the event database

Installing the CEI application

The CEI server is a J2EE application. This needs to be installed.

1. In the command prompt, change to the *<WebSphere_root>*\bin directory.

   ```
   cd <WebSphere_root>\bin
   ```

2. Execute the event-application.jacl script to install the CEI application.

   ```
   wsadmin -f "<WebSphere_root>\event\application\event-application.jacl"
   -action install -earfile
   "<WebSphere_root>/event/application/event-application.ear" -node node
   -server server
   ```

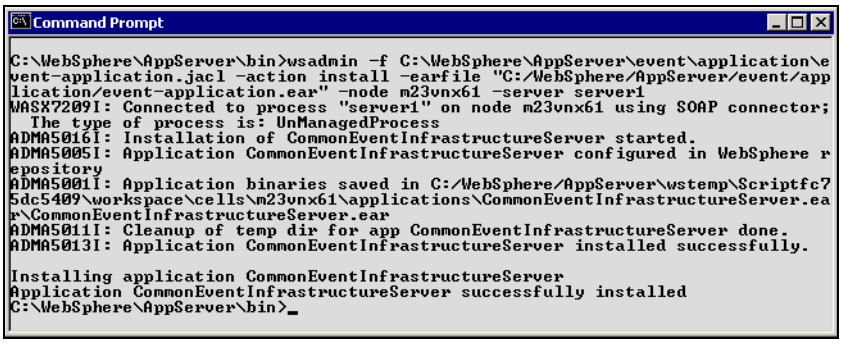


Figure 10-6 Result of application install script

The application is now installed.

Starting the CEI application

To start the application:

1. Open a new browser window and navigate to http://localhost:9090/admin.
2. Go to **Applications** → **Enterprise Applications**.
3. Find the application named CommonEventInfrastructure, select the checkbox next to it and click the **Start** button at the top of the list.
4. If the application starts successfully, you will get the following message at the top of the screen.

Figure 10-7 CEI Server application startup message

Viewing the contents of the CEI database (optional)

To validate that the installation has been successfully completed, we will review the contents of the CEI database.

1. Log out of the Administrative Console in the browser window, stop the application server (server1).

> **Note:** You have to stop the application server in order to connect to the CEI Cloudscape database. Cloudscape does not support multiple connections to a database.

2. You can use the Cloudscape viewer to look at the CEI database. Open Windows Explorer and navigate to the *<WebSphere_root>*\cloudscape\bin\embedded directory. Run cview.bat by double-clicking it**.** Cloudview will open.
3. Select **File** → **Open**. Browse to *<WebSphere_root>*\event\CloudScapeEventDB. Select the **event** Cloudscape database and click **Open**.
4. You should now be able to see the event tables. At the moment, they should contain no CBE objects.

Installing and running the CBE Event browser

Note: The CBE Event browser will be changed significantly with version 5.1.1 supporting a new correlation model.

We will now install the CBE Event browser application. We do this to prepare for the testing phase of the chapter and as a final confirmation that installation of the CEI infrastructure has been successful.

1. Make sure your application server is running.
2. Launch the Administrative Console then login.
3. Open **Applications** → **Install New Application**.
4. Using the local path box, browse to the **CBEViewer.ear** file which is located in **`<WebSphere_root>\eventbrowser\`**, click **Next**.
5. Make sure that default bindings will be generated by selecting the apropriate checkbox. Click the **Next** button.
6. Accept the default values by clicking the **Next** button for the upcoming pages, then press the **Finish** button on the last page.
7. Once the application is installed, save the configuration for WebSphere.
8. Start the application from the **Applications** → **Enterprise Applications** page. Locate **CBEViewer** on the list and click **Start**.
9. Open a new browser at `http://localhost:9080/cbeviewer`.

   You will be presented with the CBE Event browser window, as shown in Figure 10-8 on page 325. Close the viewer: we will come back to this later.

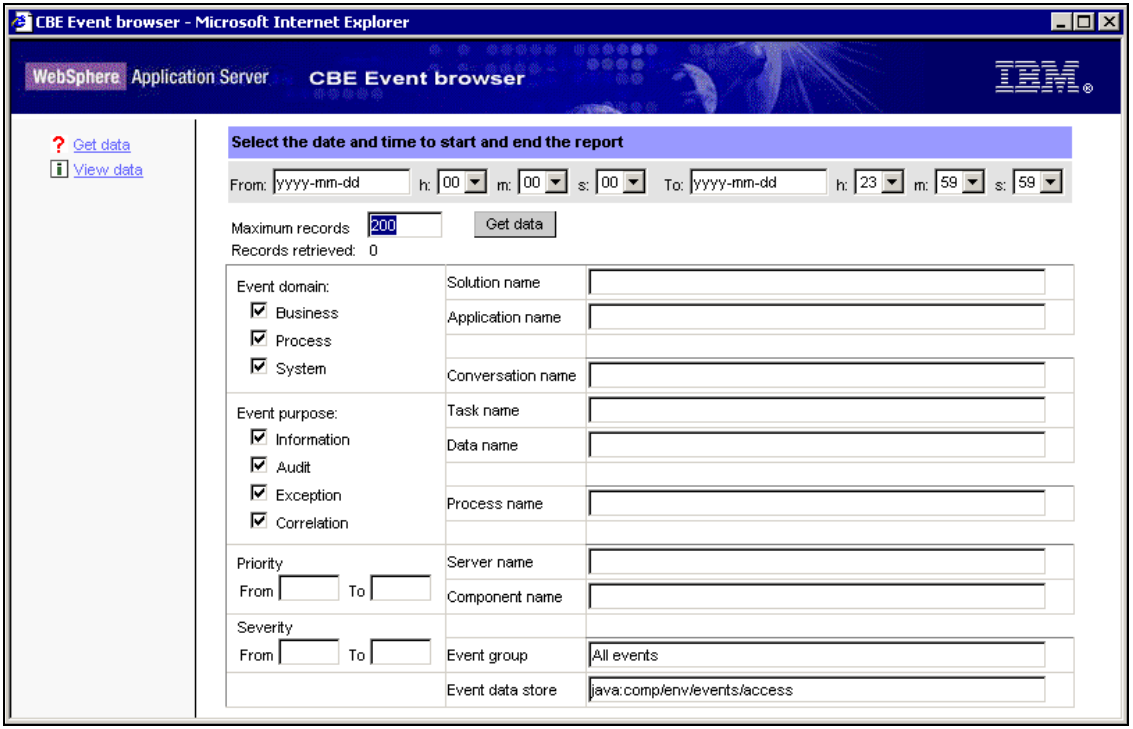


Figure 10-8 The CBE Event browser application window

10.5 Testing

This section describes running the updated business process and reviewing the events logged in the CBE viewer.

First, we need to execute the updated NiceJourney process.

1. Start the **BPE WebClient** by opening a new browser window and navigating to **http://server:9080/bpe/webclient**.
2. Browse to My Templates and select the checkbox next to the **NiceJourney** template. Click **Start Instance**.
3. First, we will start the process with the audit flag set to false. This will highlight the data that is recorded. The important values used in the test are listed in Table 10-1 on page 326.

Table 10-1 Input values for the first test

Input variable	Value
headerData.auditThis	false

The rest of the variables can be set as you wish. This will have logged some CBE records in the data store. We will now review these using the CBE Event browser.

4. Start the CBE Event browser by opening a new browser and navigating to **http://server:9080/cbeviewer**.
5. Clear the Business, Process, System, Information, Audit, Exception and Correlation checkboxes.
6. Click the **Get Data** button. The count of records retrieved should change.

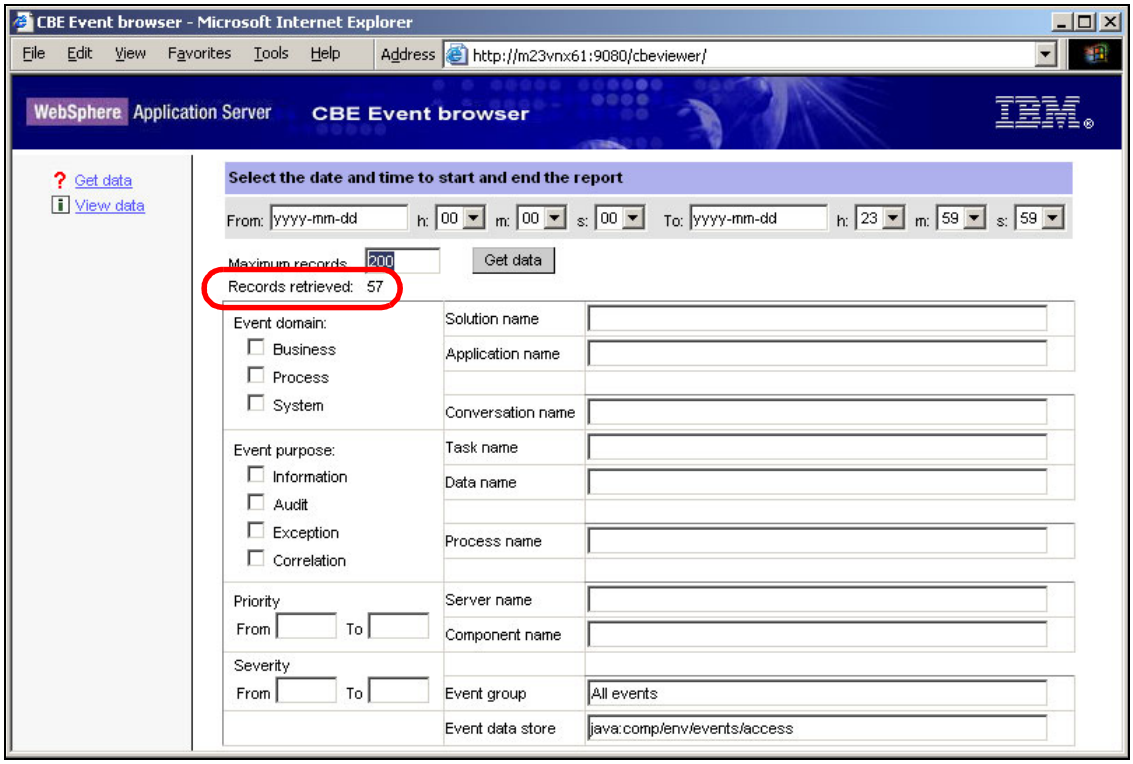


Figure 10-9 The CBE Event browser

7. Select the **View data** menu item at the left of the screen. You will be presented with the data viewer. Select the **Processes** tab at the top of the screen.

8. The viewer is now split into three sections, covering process template, process instances and the events triggered by the instance.
9. Select the **NiceJourney** process, and the instance second from the bottom. Select one of the events in the right-hand pane. You should be presented with a window similar to Figure 10-10.

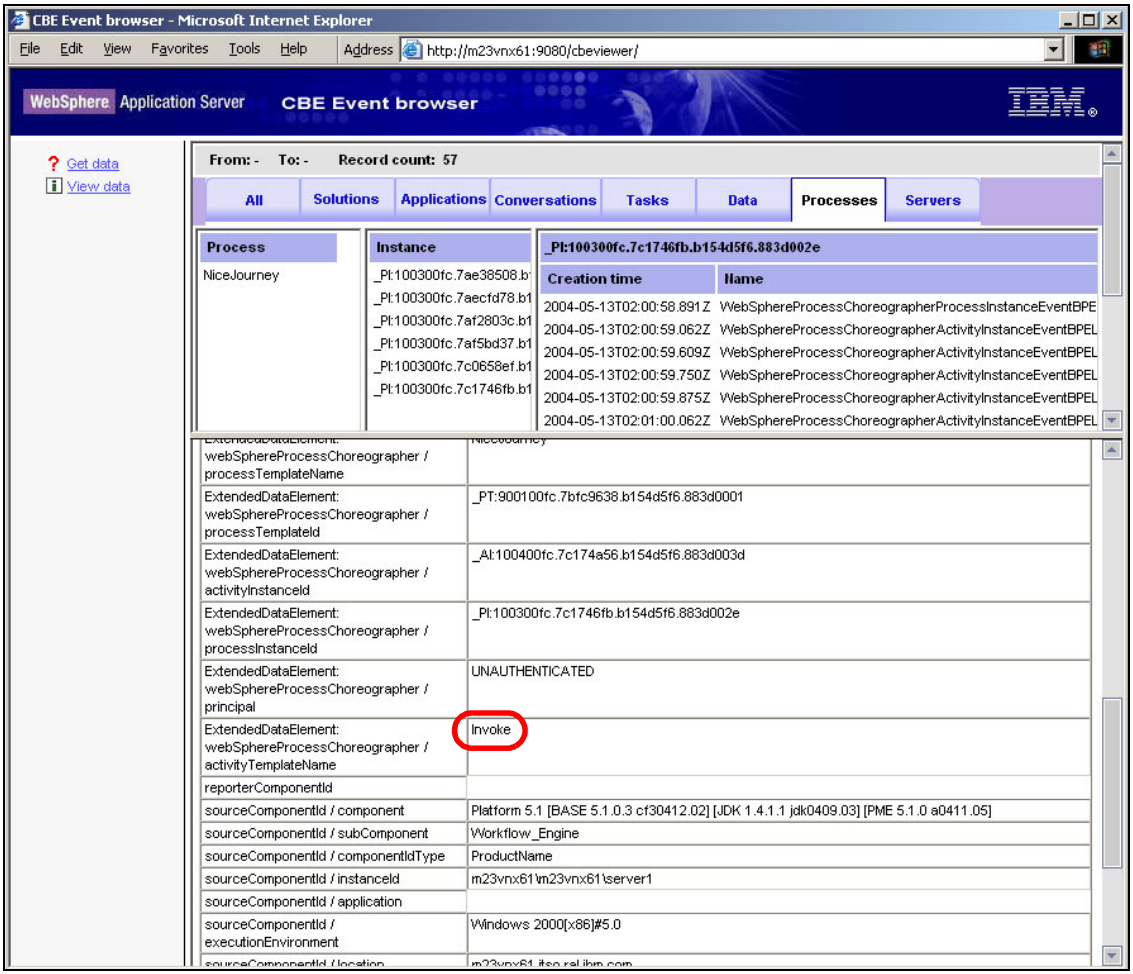


Figure 10-10 Viewing an event in the CBE Event browser

The event that has been selected here is related to the Invoke activity in the NiceJourney BPEL4WS process.

10. Browse the other events created by the process, noting the types of data that are automatically logged.

11. Invoke the process for a second time by following the first two steps. This time we will make the process audit itself by setting the audit flag to true. The input message used is shown in Table 10-2.

Table 10-2 Input values for the second test

Input variable	Value
customerData.lastName	Smith
headerData.auditThis	true

Other values can be set as you wish.

12. Starting this instance will place additional audit records in the event log. Repeat steps 4-8 to start the CBE Event browser.

Important: You must rerun steps 4-8 to refresh the data in the CBE Event browser. If you do not, the latest events will not be shown.

13. Select the NiceJourney process and select the last instance in the list. This should contain the custom event created as a result of the new Audit activity. Select this event in the list, and review the data in the event record. This is shown in Figure 10-11 on page 329.

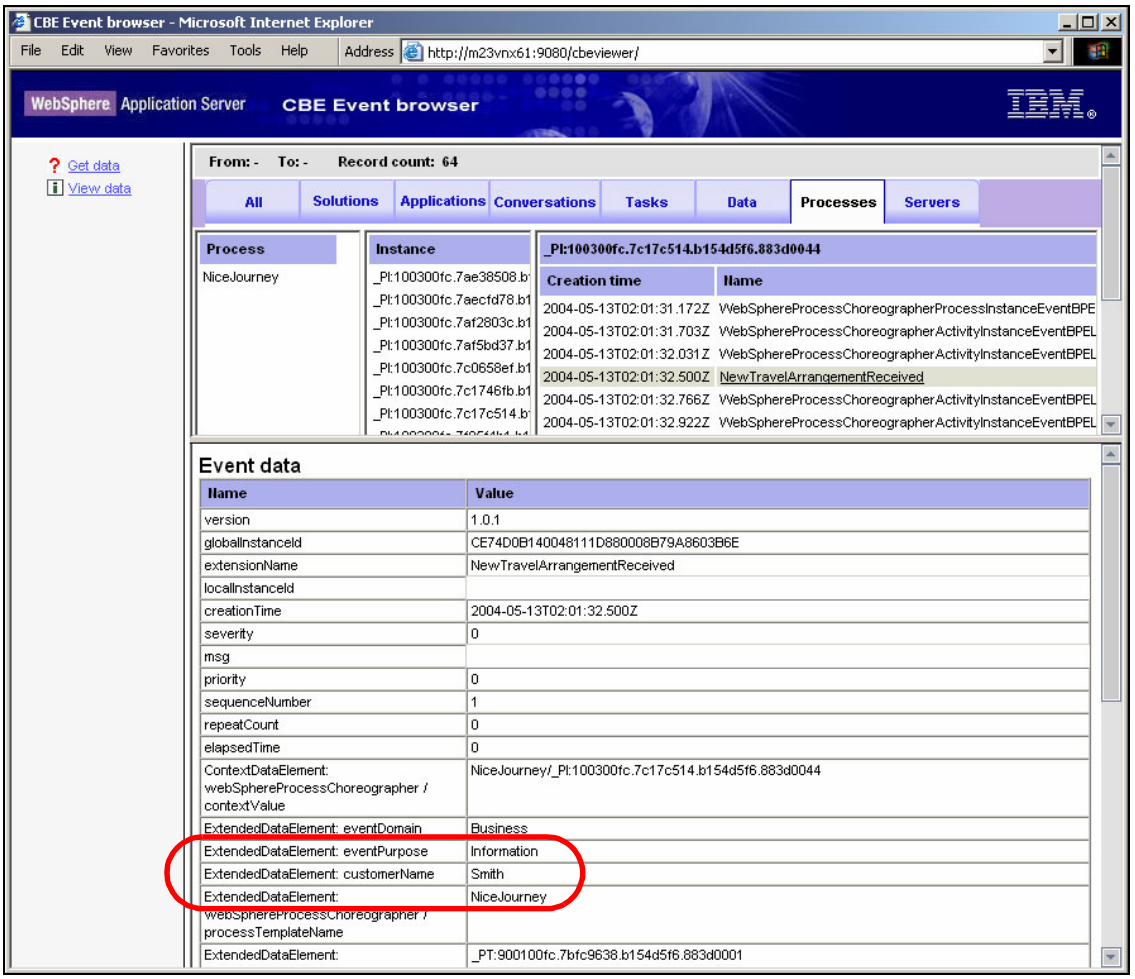


Figure 10-11 The NewTravelArrangementReceived custom audit event

Again, notice the amount of additional data that is recorded by the infrastructure. The three data elements that were explicitly placed in the event are highlighted.

10.6 More information

Further information can be found in the InfoCenter at;

`http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp`

Navigate to **WebSphere Business Integration Server Foundation -> All topics by feature -> Applications -> Application services -> Working with the Event Programming Model in WebSphere**.

An overview of the aims of the CBE can be found at:

http://www.ibm.com/developerworks/rational/library/2084.html

11

Business Rule Beans

Business Rule Beans are statements that define or constrain some aspect of a business by asserting control over some behavior of that business. A business rule officiates over frequently changing business practices, and can come from government regulations, company practices, customer status, or other external factors such as adapting business processes to react to competitive pressure. These types of rules can change periodically, and do not require a rebuild of all applications that support the business process. By externalizing rule processing, the rule can change without affecting the business process. At its simplest level, a business rule is little more than a well-placed if/then statement that compares a variable against a determined value, and then issues a command when they match.

11.1 Prerequisites

Familiarize yourself with the following WebSphere Business Integration Server Foundation V5.1 topics in the InfoCenter at http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp; select **WebSphere Business Integration Server Foundation** on the right-hand side, then click **Highlights and Features** → **Business Rule Beans**.

11.2 Sample scenario

We are going to create a Business Rule Bean to be invoked in the sample process which was created in Chapter 7, “Process choreographer: developing a simple process” on page 135.

Our sample application of Business Rule Beans is for validating a data from a user request, for example, validating a price limit if the price which a user requested is higher than the minimum price.

Development

Once you understand how Business Rule Beans work, you can start developing your own. WebSphere Studio Application Developer Integration Edition does not provide any specific tooling for Business Rule Beans development, but you can easily develop your components in the Java perspective, following a few steps:

1. Import the Business Rule Beans EAR application supplied in WebSphere Business Integration Server Foundation.
2. Develop the Rule Implementor for your rule.
3. Create and configure the rule using the Rule Management Application.
4. Create a rule client.
5. Integrate the Business Rule Beans into your application, for example, into a business process.

Unit test

If you want to test your Business Rule Beans before deployment, you can do so using the IBM Universal Test Client in the WebSphere Test Environment inside WebSphere Studio Application Developer Integration Edition.

Assembly

Packaging your Business Rule Beans with an application does not have any special requirement beyond configuring resource references (JNDI names) for your components.

Deployment

Deployment requires a few additional steps before you can use Business Rule Beans. The Business Rule Beans application (EAR) has to be installed on the application server and configured with a proper datasource. This is a one-time configuration for the application server to use Business Rule Beans from every other enterprise application.

Installing your application using Business Rule Beans does not require any specific tasks during deployment in the runtime environment.

11.3 Development

There is one BRB in the sample scenario; the following steps will guide you through the process of developing this component for your application.

11.3.1 Development environment setup

This section guides you throug the process of setting up the development environment for developing applications using Business Rule Beans. In the test environment, we will use a Cloudscape database to store the rules for Business Rule Beans.

1. Open WebSphere Studio Application Developer Integration Edition with a new workspace; make sure you enable server targeting support.
2. We need to add a JAR file for database mapping. We choose the Cloudscape JAR file for the development environment.
   a. Switch to the J2EE perspective.
   b. Select **File** → **Import**. Then select **EJB JAR file** in the list box and click **Next**.
   c. Click **Browse...** beside the EJB JAR file field and select **<WebSphere_Studio_root>\runtimes\ee_v51\BRBeans\ BEBeansCloudscape.jar**.
   d. Select **Integration Server V5.1** for the target server.

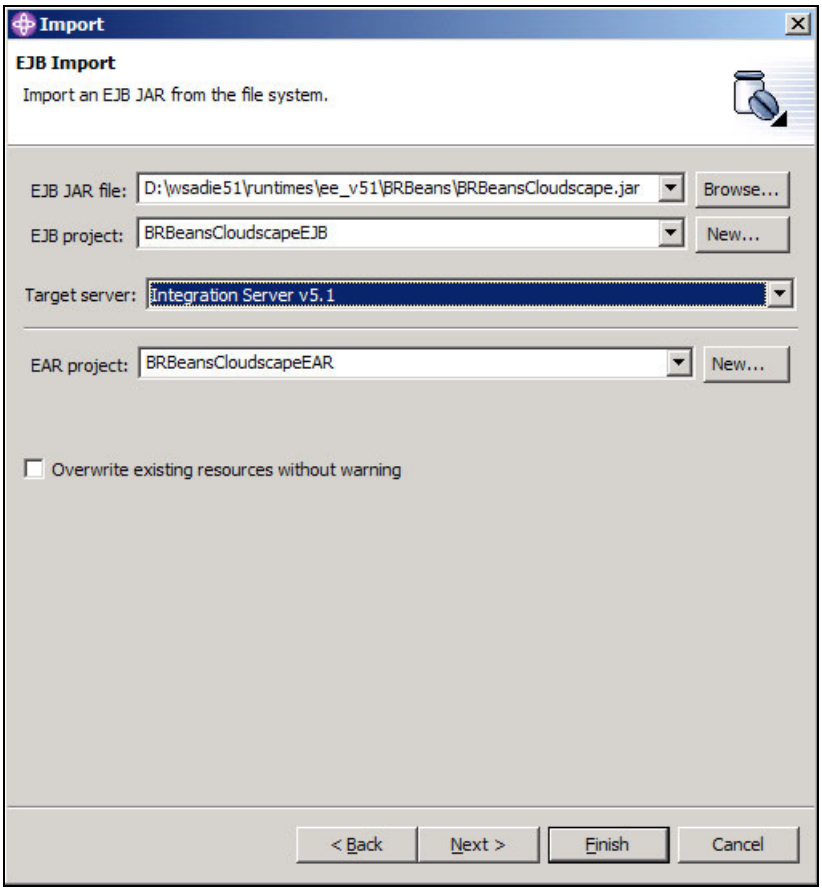


Figure 11-1 Importing EJB JAR

e. Click **Finish**.

3. Next, we can generate Business Rule Beans deployment code.

   a. Double-click the **BRBeansCloudscapeEJB** item under the EJB Modules to open the deployment descriptor.

   b. Go to the bottom of the Overview tab and enter `jdbc/BRBeansDataSource` for the JNDI - CMP Connection Factory Binding JNDI Name.

   c. Switch to the Beans tab, select the **Rule** bean, and enter the `jdbc/BRBeansDataSource` for CMP Connection Factory JNDI Name.

   d. Select the **RuleFolder** bean and again, enter the `jdbc/BRBeansDataSource` for CMP Connection Factory JNDI Name.

   e. Save and close the file.

f. Right-click the **BRBeansCloudscapeEJB** item under the EJB Modules, then select **Generate** → **Deployment and RMIC code...**.

g. Click **Select All** on the new panel, then click **Finish**.

4. Open the Server perspective and create a new Integration Server V5.1 test server, with the name: `BRBeansUnitTestServer`.

   a. Add the BRBeansCloudscapeEAR project to the server.

   b. Open the server configuration, select the **Environment** tab, and add the following item under the System properties:

      - Name: `brbPropertiesFile`
      - Value: `<WebSphere_Studio_root>\runtimes\ee_v51\bin\brbeansDefaultProperties`

   c. Switch to the Data sources tab and make sure that the Cloudscape JDBC Driver already exists in the JDBC provider list.

   d. Save and close the server configuration.

   e. Right-click the **BRBeansUnitTestServer** and select **Create tables and data sources**.

   f. You will see a pop-up window if tables and data sources were created successfully.

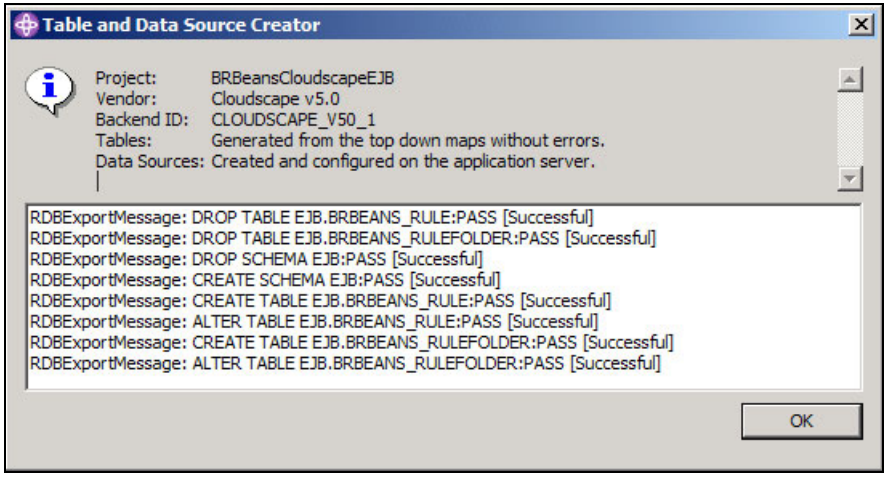


Figure 11-2 Table and Data Source Creator

g. Click **OK** to close this window.

11.3.2 Developing the rule implementor

We need to develop the rule implementor, which means creating Java classes including business logic. We need to implement the RuleImplementor interface to our business rule code. The interface has three methods.

- init()

  This method is called only once when Business Rule Beans are fired for the first time. The method is used to initialize parameters or objects.

- fire()

  This method implements business logic of the rules.

- getDescription()

  This method gives you the description of this rule.

Follow these steps to create the Rule Implementor:

1. Switch to the J2EE perspective and create a new Enterprise application with the name `BRBeans`, then create a new EJB module: `BRBeansEJB`. Make sure you select **Integration Server V5.1** for the target server.
2. Create a new package under the BRBeansEJB: `com.nicejourney.brb.rules`.
3. Create a new class in the new package with the name `PriceLimit`, then select the interface **RuleImplementor** for the class. The class is not available for selection; you have to type the full name or just part of it, then select from the list.

   PriceLimit.java should be generated on the com.nicejourney.brb.rules package.
4. Implement the methods in the code, open the PriceLimit.java and use the code shown in Example 11-1 for implementation.

Example 11-1 PriceLimit.java

```
package com.nicejourney.brb.rules;

import com.ibm.websphere.brb.*;
import com.ibm.websphere.brb.mgmt.*;

public class PriceLimit implements RuleImplementor {
   private int approvalLimit = 0;

   public Object fire(
      TriggerPoint arg0,
      Object arg1,
      IRuleCopy arg2,
      Object[] arg3)
```

```
      throws BusinessRuleBeansException {
         Boolean result = null;
         int userPriceLimit = 0;
         userPriceLimit = ((Integer) arg3[0]).intValue();
         if (userPriceLimit >= approvalLimit) {
            result = new Boolean(true);
         } else {
            result = new Boolean(false);
         }
         System.out.println("Is approval required for:" + userPriceLimit + "?
" + result.toString());

         return result;
   }

   public String getDescription() {
      return "This rule checks if price limit which user entered is over our
least price limit";
   }

   public void init(Object[] arg0, String[] arg1, String arg2, IRuleCopy arg3)
      throws BusinessRuleBeansException {
         //Store the initialization parameters.
         approvalLimit = ((Integer)arg0[0]).intValue();
   }
}
```

11.3.3 Creating and configuring the rule using the Rule Management Application

The Rule Management Application is a J2EE Client Application GUI tool to create and configure business. The following steps will show you how to start the client in WebSphere Studio Application Developer Integration Edition and how to create a new rule with it.

1. Switch to the J2EE perspective.
2. Create a new Enterprise Application Project with the name `RuleManagement`, and add only a new Application Client module: `RuleManagementApplicationClient`.
3. Right-click **RuleManagementApplicationClient** under the Application Client Modules item, select **Properties**, then navigate to Java Build Path. Under the Libraries tab, add a new variable: `WAS_EE_V51`, and extend it with `lib/brbRuleMgmtApp.jar`. Close the window by clicking **OK**.
4. Right-click **RuleManagementApplicationClient** and select **Open With** → **JAR Dependency Editor**.

5. Type the following value as the Main Class:

   `com.ibm.ws.brb.rm.ui.RuleManagement`

6. Save and close the editor.
7. Select **Run** → **Run...** from the menu.
8. Navigate to WebSphere v5.1 Application Client and click **New**.
   a. Set the name of the configuration to `Rule Management`.
   b. Set the server type to `WebSphere v5.1 EE`.
   c. Set Enterprise Application to `Rule Management`.
   d. Set Application Client to `RuleManagementApplicationClient`.
   e. Under the Arguments tab, in the VM arguments text area, replace all the base_v51 directory names with `ee_v51`; you have to replace eight occurrences of the wrong directory name.
   f. Add the following directive to the VM arguments:

   ```
   -DbrbPropertiesFile="<WebSphere_Studio_root>\runtimes\ee_v51\bin\brbeans
   DefaultProperties"
   ```

   > **Tip:** In Windows, the default installation root for WebSphere Studio Application Developer Integration Edition is C:\Program Files\IBM\WebSphere Studio\Application Developer IE\v5.1.

   g. Click **Apply** to save the settings, then click **Run** to start the application.

   The Rule Browser will be launched. You will see a folder tree.

   > **Note:** The Test server has to be started to be able to run the Rule Browser application.

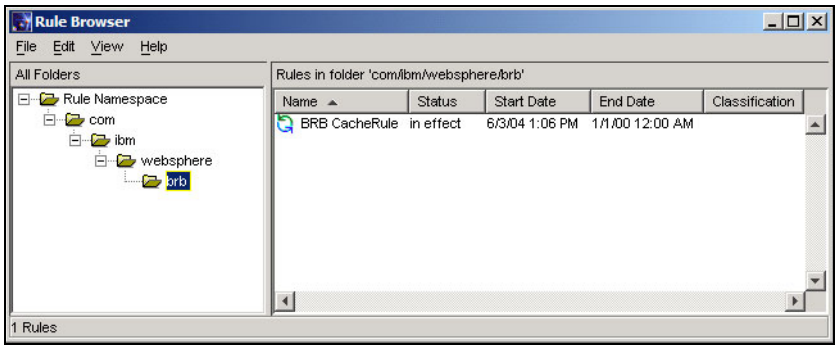


Figure 11-3 Rule Browser

9. Add a new rule using the Rule Management application.
   a. It is recommended that you follow the Business Rule Beans package naming rule. Create three folders to follow the naming rule. Right-click the **com** folder and select **New** → **Folder**, then name the folder to `nicejourney`. Create the subfolders to achieve the following hierarchy: `com/nicejourney/brb/rules`.
   b. Right-click the **rules** folder and select **New** → **Rule**. The New Rule Properties window will appear.
   c. Make sure `com/nicejourney/brb/rules` is specified for the Folder name. Enter `Price Limit` in the name field.
   d. Enter the Start date; this indicates when this Business Rule Bean will start applying. For example, this could be `01/01/04 0:00 AM`; the date is in the *mm/dd/yy h:mm a* format.
   e. Make sure you select the **Rule performs a classification** option.

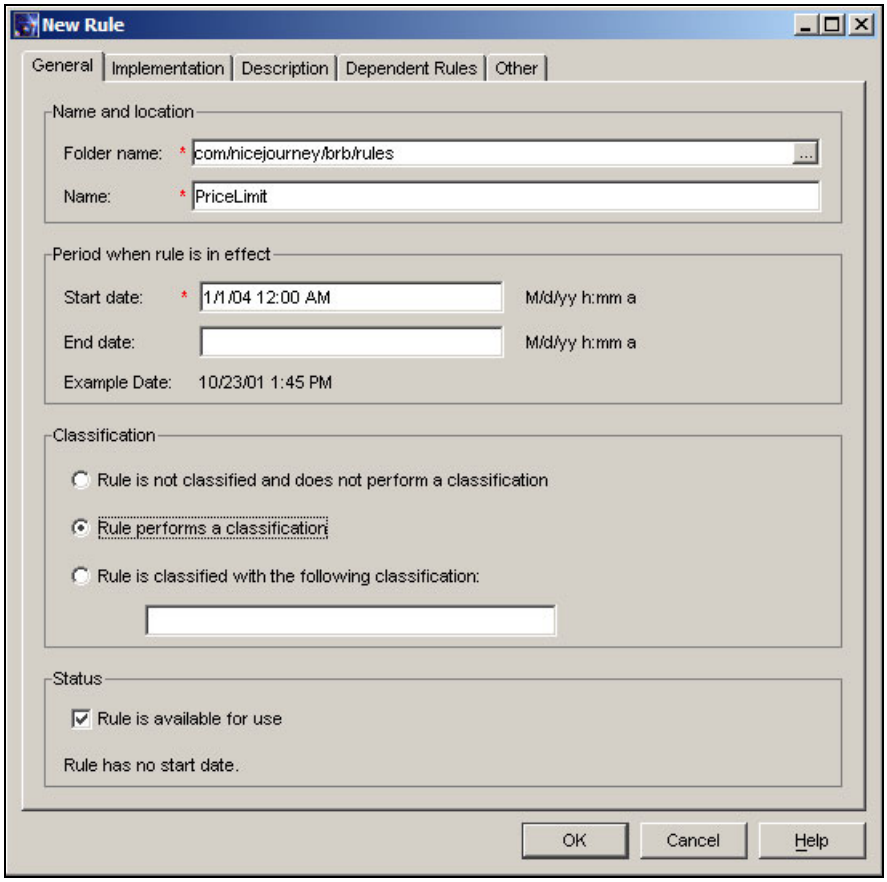


Figure 11-4 New Rule window

f. Select the Implementation tab, then enter `com.nicejourney.brb.rules.PriceLimit` in the Java Rule Implementor.

g. Click **Add** to add Initialization parameters; the Add Initialization Parameter window will appear.

h. Enter the Description which is a parameter name, for example `Approval limit for customers`. Select **Integer** for the type, which is a type of the parameter.

i. Enter the Value, which is the initial value, for example `100`.

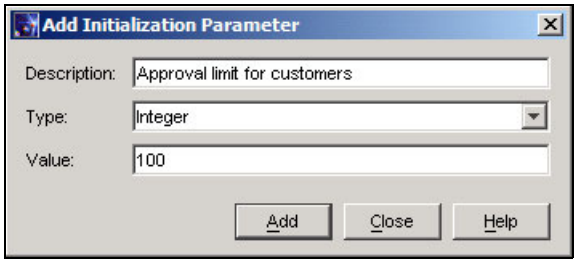


Figure 11-5 Rule Browser -Add Initialization Parameter window

j. Click **Add**, then click **Close**. Then new line will be added to the Initialization Parameters table.

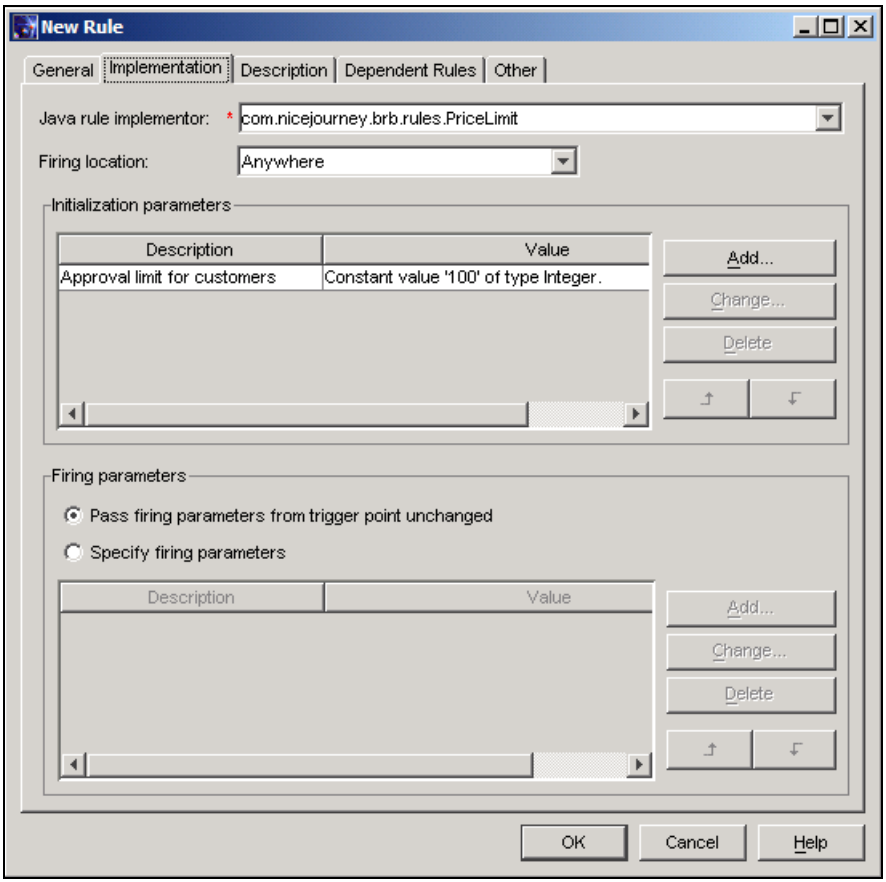


Figure 11-6 New rule implementation

k. Click **OK**; you should find your new rule on the Rule browse.

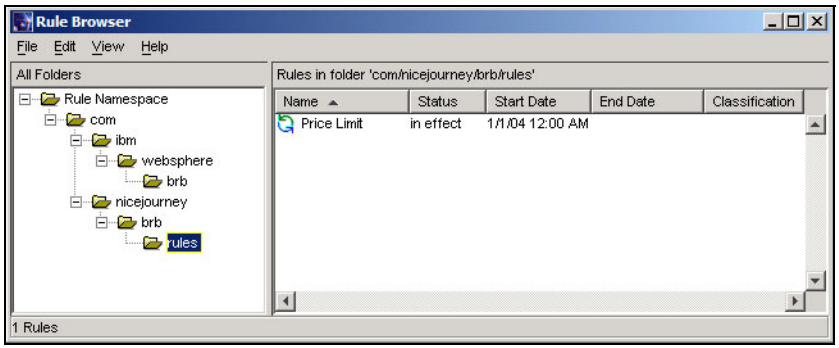


Figure 11-7 Rule Browser after defining a rule

l. Close the Rule Browser.

11.3.4 Creating the rule client

In our sample application, the rule client is a session bean that is used by another application or process.

To create the rule client session bean and to place a trigger point, do the following:

1. Switch to the J2EE perspective.
2. Create a new Enterprise Application Project with the name `BRBeans`, then add a new EJB Module: `BRBeansEJB`. Make sure the **Integration Server V5.1** is selected as the server target.
3. Create a new EJB in the BRBeansEJB project with the following details:
   - Type: `Session bean`
   - Bean name: `PriceLimitClient`
   - Default package: `com.nicejourney.brb.ejbs`
   - Session type: `Stateless`
   - Transaction type: `Container`

   Check both **Local** and **Remote client view**.
4. Open the PriceLimitClientBean.java file and add a new import statement to the top of the file:

   ```
   import com.ibm.websphere.brb.*;
   ```
5. Add the isPriceLimit() method to the class; insert the code into the source shown in Example 11-2 on page 343.

Example 11-2 isPriceLimit() method

```
public boolean isPriceLimit (int total) {
   boolean result = false;
   try{
      //Create new trigger point
      TriggerPoint tp = new TriggerPoint();

      //Disable caching - for demonstration purpose
      tp.disableCaching();

      //The rule expects one parameter: the total purchase order amount
      Object[] firingParams = { new Integer(total) };
      // Specify the business rule path.
      String ruleName = "com/nicejourney/brb/rules/PriceLimit";
      //Call the rule
      Object resultObject = tp.triggerClassifier(
         null, //Target Object-not required by rule implementation
         firingParams, //Rule Implementor firing parameters
         ruleName); //The name of the rule to fire
      if (resultObject!=null){
         //Rule successfully called
         result = ((Boolean) resultObject).booleanValue();
   }
   }catch (BusinessRuleBeansException brb){
      brb.printStackTrace();
      // normally here you would do proper exception handling
   }catch (Exception e) {
      e.printStackTrace();
      // normally here you would do proper exception handling
   }
   return result;
}
```

6. Select the **Outline** view for the lower left view, right-click the **isPriceLimit(int)** method, then select **Enterprise Bean** → **Promote to Remote Interface**.
7. Perform the previous step again, clicking **Enterprise Bean** → **Promote to Local Interface**.
8. Save and close the file.
9. Generate the deployed code for the EJBs in the BRBeansEJB project.
10. Switch to the Server view and add the BRBeans project to the BRBeansUnitTestServer.

11.3.5 Using Business Rule Beans in Process Choreographer

This section will use the Business Rule Bean developed in the previous section; we will use it in the process developed in Chapter 7, “Process choreographer: developing a simple process” on page 135. We have a slightly modified version of the simple process in order to support the Business Rule Bean in the process.

The customer can set a maximum price for the travel booking. The Business Rule Bean validates this maximum price, and does not let the customer set a very low value. If the maximum price is lower than the PriceLimit set for the process then the process terminates.

You can call Business Rule Beans as a Partner Link using the Invoke activity in the process. The sequence of steps with which the Business Rule Beans is called is as follows:

1. The Business Rule Beans Client, which is the session EJB, is called by the Invoke activity.
2. The Business Rule Beans Client calls Business Rule Beans using the TriggerPoint.
3. The Business Rule Beans gets the rules which you set and adapt to the parameters.
4. An object is returned as a result to the Business Rule Beans Client from the Business Rule Beans.
5. The result is returned to the process.

You need to add or modify the following activities and variables to use BRBeans in the process.

- Partner Link
  Define the Business Rule Beans that can be accessed by an Invoke activity.
- Variables
  Define request and response which are sent and received between an Invoke activity and Business Rule Beans.
- Assign
  Copy a request parameter from TravelData to PriceLimitRequest.
- Invoke
  Create this activity to call the Business Rule Beans and get a result in the process.
- Java snippet
  Modify this activity to show you the value of response from the Business Rule Beans.

Prerequisites

Import the simpleProcess_for_BRB.pi.zip archive from the additional material into the workspace using the Project Interchange plug-in. You can find details about how to import the simple process using the Project Interchange plug-in in “Project Interchange archive import/export” on page 562. Select the simpleProcess.pi.zip for import from the additional material.

Creating a Partner Link

First, you need to create a Partner Link for the Business Rule Beans service.

1. Select the **Business Integration** perspective and open nicejourney.bpel.
2. Switch to the Package Explorer view, then select the EJB Remote Interface, **LimitPriceClient.java**, from the com.nicejourney.brb.ejbs package in the BRBeansEJB project. Drag and drop the file to the BPEL Editor. A new Partner Link will be created.
3. Rename the partner link to `PriceLimitRule`.
4. Select **PriceLimitRule** and switch to the Implementation tab. Make sure that the PriceLimitClient is specified in the EJB Remote Interface field.

   Important: If you try to create from the Partner Link in the BPEL Editor and change the type to EJB, be aware that there is a known bug; although WSDL files will be generated, the WSDL files will not define the binding as Java instead of the EJB type.

5. Save the file.

Creating variables

The next step is to create variables for the rule invocation.

1. Click the **+** button next to *Variables* in the BPEL Editor. Name the new variable to `priceLimitRequest`. Switch to the Message tab, then click **Browse...** and click **WPC_Simple_Process** → **com** → **nicejourney**, then select **comnicejourneybrbejbsPriceLimitClientNiceJourneybpel.wsdl**, making sure **isPriceLimitRequest** is selected.
2. Create another variable: `priceLimitResponse`, and set the message to `isPriceLimitResponse` from the comnicejourneybrbejbsPriceLimitClientNiceJourneybpel.wsdl file.
3. Save the file.

Adding the Assign element

The Assign element will prepare the variables for the validatePrice activity.

1. Select the **prepValidatePriceLimit** Assign activity and set up the following assignments; click **travelRequest** → **travelAgencyIn** → **customerData** → **priceLimit to priceLimitRequest** → **isPriceLimitRequest** → **total**.

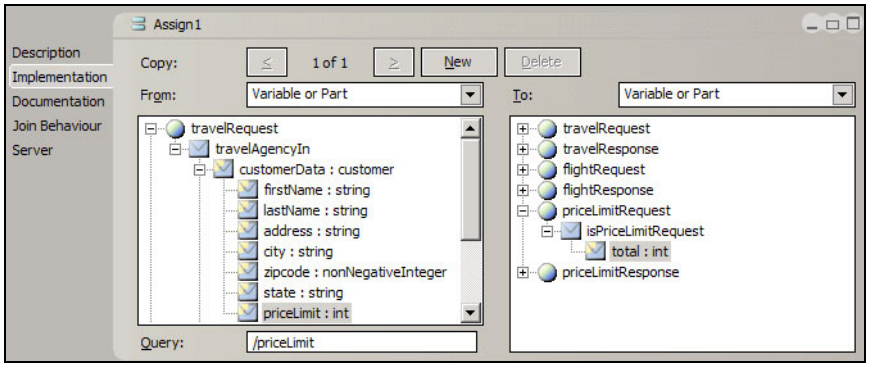


Figure 11-8 Assign

2. Save the file.

Creating an Invoke activity for the rule client

We are going to call the rule bean via the rule client EJB using the Invoke activity.

1. Right-click the **validatePriceLimit** empty activity and change the type to `Invoke`.
2. Right-click the **validatePriceLimit** Invoke activity and set the partner link to the `PriceLimitRule` partner link.
3. Switch to the Implementation tab and select **isPriceLimit** for the Operation.

   Select **PriceLimitRequest** for the Request variable, then select **PriceLimitResponse** for the Response variable.
4. Save the file. The final version of your process should look similar to Figure 11-9 on page 347.

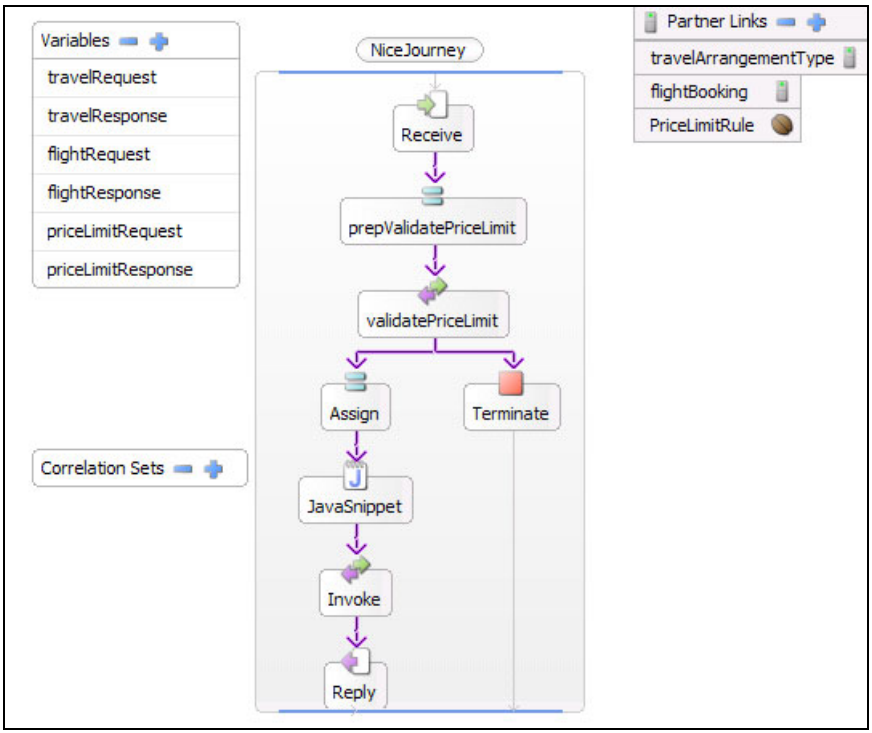


Figure 11-9 Final process for the Business Rule Beans sample

Generating the deployed code and running the process

At this point, the process development is complete. You can generate the deployed code for the process then add it to the test server to run.

11.4 Unit test

In this section, we test our business rule and business rule client using the Universal Test Client.

In order to use the Universal Test to test the rule itself without the sample application, do the following:

1. Start the BRBeansUnitTestServer test server.
2. Switch to the J2EE perspective, navigate to **EJB Modules** → **BRBeansEJB** → **Session Beans**, then right-click the **PriceLimitClient** and select **Run on Server...**.
3. When you get a confirmation window, click **Finish**.

4. Select **Test EJB remote interface** in the Select a server client window; click **Finish**.
5. The IBM Universal Test Client should start with the PriceLimitClient EJB already listed under EJB References.
6. Create an instance of the bean then open the new instance and select the **boolean isPriceLimit(Integer)** method.
7. Provide a numerical value in the input field, for example: `120`. Then click **Invoke**. You should get the result `true`, as shown in Figure 11-10.

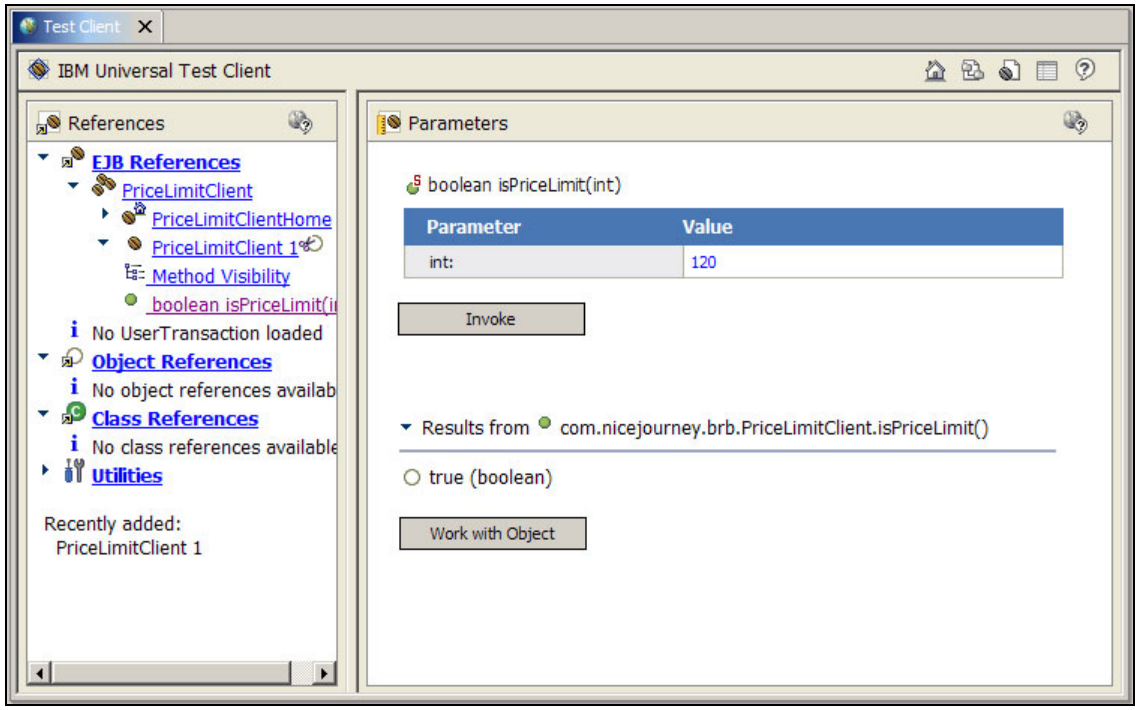


Figure 11-10 Testing the Business Rule Beans

8. Enter another number in the input field (lower than 100), for example `90`, and click **Invoke** again.
9. The client should return the result `false`.

11.5 Deployment

This section contains detailed information about the setup and configuration of an application using Business Rule Beans. There are a few additional steps to be performed before you can use your enterprise application.

In the runtime environment, described below, we are going to use the IBM DB2 database. If you need to use other database servers, the steps below can be easily applied to other situations.

1. Start the application server, launch and log on to the Administrative Console.
2. We have to define the *brbPropertiesFile* directive for the application server JVM. Select **Servers** → **Application Servers** then select **server1**. In the additional properties section, select **Process Definition**, which is used to define the command line information necessary to start/initialize a process.
   a. In the Process Definition additional properties, select **Java Virtual Machine** → **Custom Properties**.
   b. In the Custom properties window, click the **New** button to create a new property. Enter the following values:
      - Name: `brbPropertiesFile`
      - Value: `<WebSphere_root>\bin\brbeansDefaultProperties`
   c. Click **OK** and save the configuration for WebSphere.
3. Restart the application server and launch the Administrative Console again.
4. Navigate to **Environment** → **Manage WebSphere Variables**; make sure that the DB2UNIVERSAL_JDBC_DRIVER_PATH and the DB2_JDBC_DRIVER_PATH variables are pointing to the <DB2_root>/java directory.
5. Navigate to **Resources** → **JDBC Providers** and create a new non-XA JDBC Provider (ConnectionPool) for DB2.
6. Create a JAAS Authentication Entry for the database access with the following details:
   – Alias: `BRBDBAlias`
   – User ID: <DB2 user ID for the BRB database>
   – Password: <DB2 password for the BRB database>
7. Create a new datasource (version 4) under the DB2 (non-XA) JDBC provider with the following details.
   – Name: `BRBeansDatasource`
   – JNDI name: `jdbc/BRBeansDatasource`
   – Component-managed authentication alias: `BRBDBAlias`
   – Container-managed authentication alias: `BRBDBAlias`

- Database Name: `BRBDB`
- serverName (under the Custom properties): `localhost`

Note: You have to create a V4 data source, because CMP 1.x EJBs do not work with relational resource adapters in newer versions (V5).

8. Install the enterprise application BRBeansDB2.jar from the <WebSphere_root>\BRBeans directory.
   a. At step 1, make sure the **Deploy EJBs** box is checked, so the deployment code for the EJBs will be generated during installation.
   b. At step 4, provide default datasource mapping for modules containing 1.x entity beans. Also provide the JNDI name for the datasource: `jdbc/BRBeansDatasource`.
   c. At step 5, map datasources for all 1.x CMP beans and provide the JNDI name `jdbc/BRBeansDatasource` for both EJBs (Rule, RuleFolder).
   d. At step 8, map security roles to users/groups and map your users according to your security policy. In our case, for testing purposes, we set `Everyone` for both roles (RuleManager, RuleUser).
9. The next step is to set up the database for the Business Rule Beans. We left this step for the end, so we can use the DDL file to create the database tables, generated during the application deployment in WebSphere.
   a. Navigate to **Applications** → **Enterprise Applications**.
   b. Check the box next to **BRBeansDB2_jar**, then click **Export DDL**.
   c. On the following page, click the **BRBeansDB2_jar.ear/_BRBeansDB2.jar_Table.ddl** link to save the DDL file on your machine. Save it to a temporary location, for example: c:\temp.
   d. Open a DB2 Command Window and issue the following commands:

```
db2 create db BRBDB
db2 connect to BRBDB
db2 -tvf c:\temp\_BRBeansDB2.jar_Table.ddl
db2 disconnect current
```

   Check that the script ran successfully, then close the command window.
10. Switch back to the WebSphere Administrative Console and navigate to **Applications** → **Enterprise Applications**. Check the box next to **BRBeansDB2_jar**, then click **Start**.

    The application should start.

Rule management application

The rule management application can be started from a command window from the <*WebSphere_root*>/bin directory. Use the following command:

```
rulemgmt <WebSphere_root>/bin/brbeansDefaultProperties
```

The brbeansDefaultProperties file defines the connection properties; if you connect to a server other than localhost or the port (RMI) is other than 2809, then you will need to modify the properties file.

12

Chapter 12.

Extended messaging

Asynchronous messaging patterns are an important part of the J2EE applications programming model. These patterns provide for "loose coupling" between applications and are useful for process flows, parallel processing, time-independent processing, and event-driven processing.

Extended messaging extends the base JMS support, support for EJB 2.0 Message-Driven beans, and the Enterprise Java Bean (EJB) component model, to use the existing container-managed persistence and transactional behavior.

In addition to providing such container-managed messaging, extended messaging provides new types of enterprise beans and administrative objects for messaging, and new functionality like data mapping and late response handling. (The abbreviation, CMM, for the term container-managed messaging is sometimes used to represent extended messaging.)

12.1 Prerequisites

Read the details about WebSphere Business Integration Server Foundation's Extended Messaging at the following locations:

- WebSphere Business Integration Server Foundation InfoCenter at:

  http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp

  Select **Extended messaging** under Highlights and Features.
- Open the WebSphere Studio Application Developer Integration Edition help and navigate to WebSphere Business Integration Server Foundation, then select **Extended messaging** under Highlights and Features.
- Read the article *IBM WebSphere Developer Technical Journal: Creating Extended Messaging Applications for WebSphere Application Server Enterprise, Version 5* at IBM developerWorks:

  http://www-106.ibm.com/developerworks/websphere/techjournal/0304_klinger/klinger.html
- Read the article *WebSphere V5 Extended Messaging Support* at IBM developerWorks:

  http://www-106.ibm.com/developerworks/websphere/library/techarticles/0302_wadley/wadley.html
- Read the article *IBM WebSphere Developer Technical Journal: Simplify Applications by Using WebSphere Extended Messaging* at IBM developerWorks:

  http://www-106.ibm.com/developerworks/websphere/techjournal/0303_green/green.html

12.2 Sample scenario

The sample application is using the travel agency (NiceJourney) scenario to show the usage of Extended Messaging. First, we will develop a self-contained messaging component as part of an enterprise application (the only component of the application). The sample is quite simple; it implements a fire-and-forget pattern; the resulting component is a Message Sender bean.

Later, we will use the Extended Messaging component in the simple business process introduced in Chapter 7, "Process choreographer: developing a simple process" on page 135. The Sender bean will send a log message using the same fire-and-forget pattern introduced in the self-contained application.

Migration

There are no specific tasks to perform to migrate WebSphere Enterprise V5 applications using Extended Messaging to WebSphere Business Integration Server Foundation.

Development

Developing messaging applications and application components using Extended Messaging is quite simple. Extended Messaging reduces the efforts spent on coding messaging for applications using messaging APIs. Extended Messaging is supported by a wizard-based rapid application development environment in WebSphere Studio Application Developer Integration Edition.

The development of a simple Sender bean is described in 12.3, “Development” on page 356. Later, we will extend the development using the Sender bean in a simple business process. Since we do not go into details about Extended Messaging patterns and development, the focus in this chapter is to show how to use an Extended Messaging component in a business process.

Unit test

Unti testing a Extended Messaging component or application is very similar to testing any other EJB component or application.

Assembly

There are a few settings for Extended Messaging in the EJB deployment descriptor. We will check these settings to see what can be done in assembly time.

Deployment

There are a few specific tasks to perform when deploying an application with Extended Messaging.

Applications with Extended Messaging components simply deploy like any other application with messaging components. You have to configure your messaging providers and components for WebSphere Business Integration Server Foundation through administration before you deploy your application.

Once you have the messaging provider and components taken care of, you can configure the Extended Messaging components.

Testing

Testing in the runtime environment can be performed using the same methods as in the development environment. You can either turn on tracing for the messaging component to see the messaging activities or you can use a

messaging queue browser when you have an external messaging provider, such as MQ Explorer in the case of WebSphere MQ.

12.3 Development

We will first create an Extended Messaging bean, and then invoke it from a process.

12.3.1 Creating an Extended Messaging bean

This section describes how to develop a simple Sender bean. Discussing all the messaging patterns and developing all the different bean types is outside of the scope of this book. If you want to know more about these details, refer to 12.1, “Prerequisites” on page 354 for help.

The Sender bean will be the message sender in our fire-and-forget scenario. We will not develop the receiver side for the pattern. To test the application, we can simply enable tracing to see the results, or we can use a message queue browser, for example: MQ Explorer for WebSphere MQ.

Creating the project

In this section, we will create a new project for this development.

Important: Before developing, make sure that Server Targeting is enabled. Open WebSphere Studio Application Developer Integration Edition and select **Window** → **Preferences** from the menu. Open the J2EE preferences page and select the **Enable server targeting support** radio button.

First, create the EJB project and its associated EAR project.

1. Open the WebSphere Studio Application Developer Integration Edition J2EE perspective and change to the J2EE view.
2. Select **File** → **New** → **EJB Project** from the menu and select **Create 2.0 EJB Project**. Click **Next**.
3. Name the project `EMS_loggingQueue` and select **Integration Server v5.1** as the target server.
4. Click the **New...** button to create a new EAR project called EMS_loggingQueueEAR; and target this at **Integration Server V5.1**. Click **Finish**.
5. Click **Finish**.

Creating the Sender bean

Once we have the project established, we will create the Sender bean for the fire-and-forget pattern.

1. Right-click the **EMS_loggingQueue** project and select **Extended Messaging** → **Create Sender Bean**.
2. Click **Create sender.**
   - Bean name: `LogSender`
   - Default package: `com.nicejourney.log`

   Click **Next**
3. On the Enterprise Beans details window, make sure that both **Local client view** and **Remote client view** checkboxes are selected.
4. Click **Finish**.
5. Fill in the port, response and data mapping information as per Figure 12-1, then click **Next**.

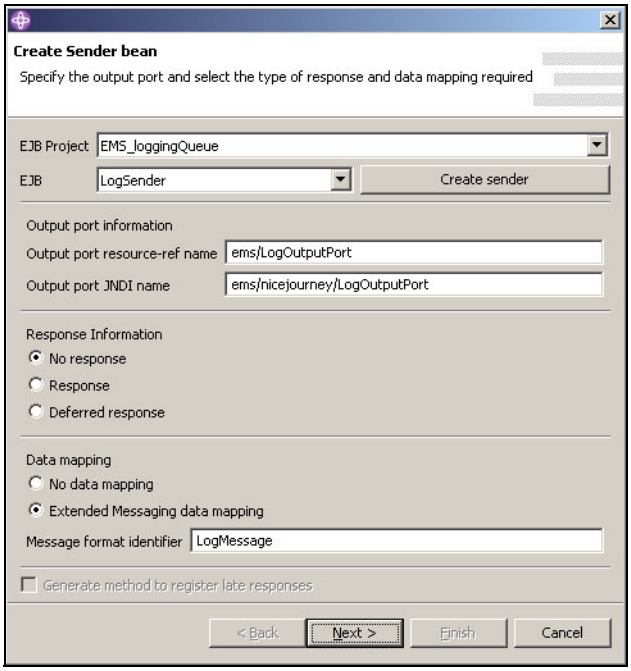


Figure 12-1 Create LogSender bean

The Output port resource-ref name is the hard-coded reference used internally in the extended messaging EJB. This is mapped to the Output port JNDI name in the EJB deployment descriptor. The setting entered here is

merely an initial value which can be changed at a later date. This JNDI name eventually maps to a physical JMS provider, which could be a WebSphere MQ queue.

The response information setting of `No response` means that this sender is not expecting a response (as we are implementing a fire-and-forget pattern).

The data mapping setting of `Extended Messaging data mapping` provides a signature to the LogSender send method which assists in creating a message. The format identifier is an arbitrary name which will be used when creating the JMS message.

Click **Next**.

6. On the Send with no response window, select both the **Add to remote interface** and **Add to local interface** checkboxes. This adds the send() method to the LogSender bean interfaces. Click **Next**.

   Note: Although we will use the local interface during the unit test of this code, we have to use a remote EJB interface when using the Extended Messaging bean as part of a business process.

7. Select the **Define and validate method signature** radio button. This will define the parameters for use with the send() method. These will then be used by the CMMFormatter class to create the JMS message. Click **Next**.
8. The send() method parameters are listed in Table 12-1. Add these parameters on the *Define and validate the sending method signature* by entering the relevant details in the Parameter name and Parameter type boxes and clicking the **Add** button.

Table 12-1 LogSender send() method parameters

Parameter name	Parameter type
customerData	java.lang.String
travelData	java.lang.String

When complete, , the window should look similar to Figure 12-2 on page 359.

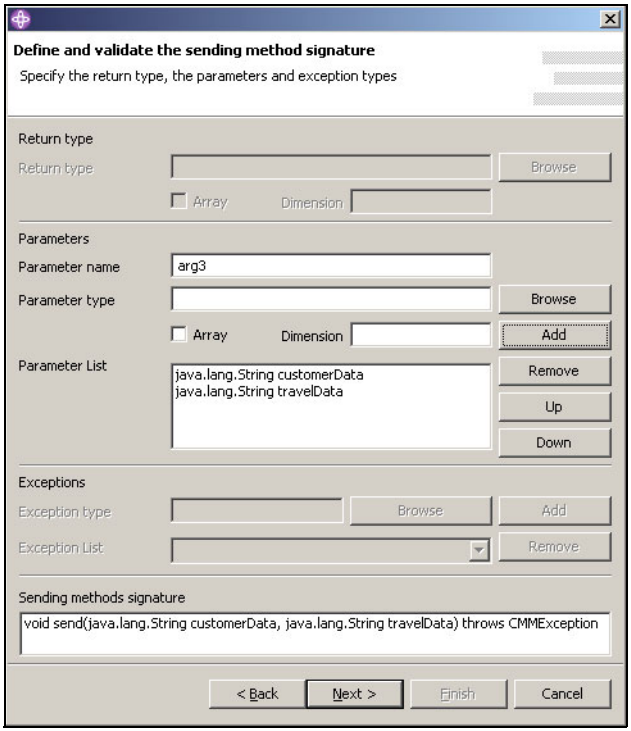


Figure 12-2 Define LogSender send() method signature

Click **Next**.

9. The summary window will be displayed. Check that the details are correct and click **Finish**.
10. Right-click the **LogSender** session bean and select **Generate Deployment Code** from the menu.

Reviewing the generated bean

The previous steps have created a stateless session bean with a send() method. Execution of the method will customerData and travelData strings provided as parameters, assemble a JMS message and place the message on the queue defined by the `ems/LogOutputPort` resource.

1. Expand **EJB Modules** → **EMS_loggingQueue** → **Session Beans** → **LogSender** in the J2EE Hierarchy view. Notice that both remote and local interfaces have been created because both were specified.
2. Double-click **LogSenderBean** to open the source code and scroll down to the `send()` method.

You will see:

– A Sender bean being created linked to the resource-ref name:

```
CMMSender sender = CMMFactory.createSender("ems/LogOutputPort");
```

– A formatter being created and then updated with the input parameters:

```
CMMFormatter formatter = CMMFactory.createCMMFormatter(factory);
formatter.addStringParameter(customerData);
formatter.addStringParameter(travelData);
```

– The assembled request message being received from the formatter:

```
Object request = formatter.getMessage();
```

– The message type set to LogMessage on the sender

```
sender.setRequestMessageType("LogMessage");
```

– The message being sent to the queue

```
sender.sendRequest(request);
```

3. Double-click **ResourceRef ems/LogOutputPort** under LogSender in the J2EE Hierarchy to open the References tab of the Deployment Descriptor. You should see a window similar to Figure 12-3.

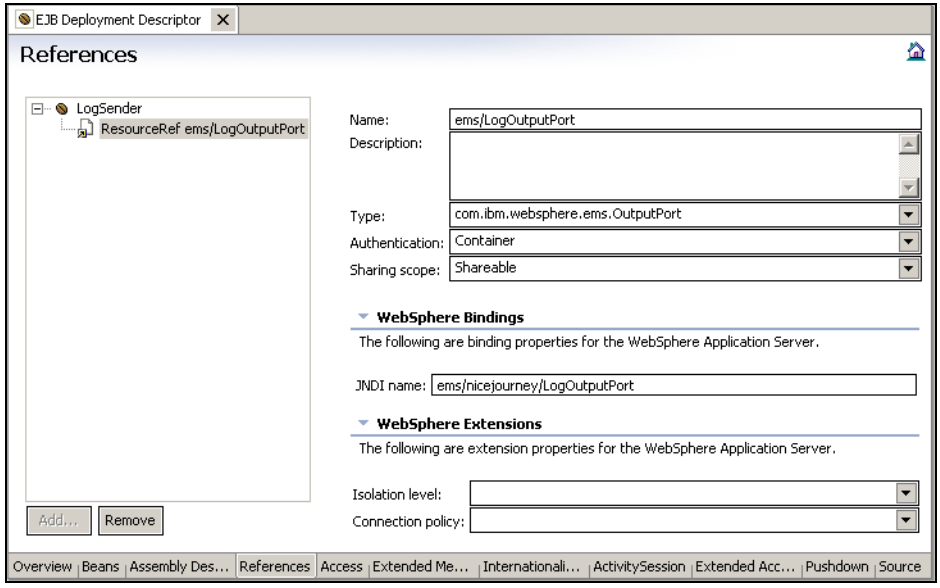


Figure 12-3 Deployment descriptor References tab for LogSender bean

Notice the local resource name `ems/LogOutputPort` which is bound to the WebSphere JNDI resource `ems/nicejourney/LogOutputPort`. This indirection

provides additional flexibility when deploying the code. We will configure the WebSphere JNDI resource later.

12.3.2 Using Extended Messaging with Process Choreographer

1. Import the simple process into the workspace. Refer to “Project Interchange archive import/export” on page 562 for detailed instructions.
2. Right-click the **WPC_simpleProcess** project, select **Refactor** → **Rename...** and rename the project `WPC_EMS_demo`.
3. Open NiceJourney.bpel using the BPEL Editor.
4. Switch to **Package Explorer** view, and open **EMS_loggingQueue** → **ejbModule** → **com.nicejourney.log**.
5. Drag and drop the file LogSender.java EJB remote interface source file to the BPEL Editor canvas. This will create a new EJB partner link called LogSender.
6. Add a JavaSnippet and an Invoke activity into the process, renaming and linking them so that they resemble Figure 12-4.

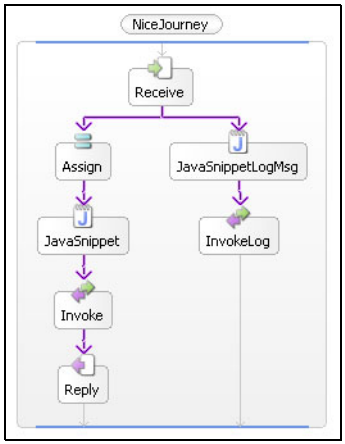


Figure 12-4 EMS demonstration process

7. Select the **InvokeLog** activity. Set the partner link to `LogSender`.
8. Open the Implementation tab in the details area.
9. Set the Operation to `send` using the drop-down list.
10. Click the **New** button next to the Request and create a new variable called `logMessage`.

11. Click the **New** button next to the Reply and create a new variable called `logResponse`.

> **Note:** Although there is no response from the LogSender bean, the BPEL process needs to have a response variable allocated. If you examine this variable, you will see that it has no parts.

12. Select the **JavaSnippetLogMsg** activity and open the Implementation tab.
13. Enter the following code into the Implementation text box.

Example 12-1 Contents of the JavaSnippetLogMsg

```
// Create and populate temporary strings
String customerDataLogMsg=getTravelRequest().getCustomerData().getLastName()+", "
    +getTravelRequest().getCustomerData().getFirstName();
String travelDataLogMsg = getTravelRequest().getTravelData().getCityFrom()+" to "
    +getTravelRequest().getTravelData().getCityTo();
// Set log message parts
getLogMessage(true).setCustomerData(customerDataLogMsg);
getLogMessage(true).setTravelData(travelDataLogMsg);
```

14. Save and close the NiceJourney BPEL4WS process. Check that there are no errors in the Tasks view.
15. Right-click the **NiceJourney** process and select **Enterprise Services** → **Generate Deploy Code**. This will open the Generate BPEL Deploy Code window.
16. Open **Referenced Partners** → **LogSender**. Check that the details resemble Figure 12-5, then click **OK**.

Figure 12-5 LogSender referenced partner

17. Check that the build activity did not generate any errors.

12.4 Unit test

In this section, we will test the Sender bean component independently from the process where we are going to use this component later.

12.4.1 Creating and configuring a server

Create a new Integration Test Environment with the name `EMS_unitTest`, then add the EMS_loggingQueueEAR project to the configured projects list. If you need detailed steps for this task, refer to “Integration Server V5.1 test environment setup” on page 563.

Before the code can be executed on the server, some configuration needs to be peformed.

1. Switch to the Server perspective, then double-click the **EMS_unitTest** server in the Server Configuration view to open the server configuration editor.
2. Select the **JMS** tab and open the Node Settings part.
3. Create a new JMS Connection Factory by clicking the **Add** button next to the WASQueueConnectionFactories entries, then specify the settings as shown in Figure 12-6 on page 364.

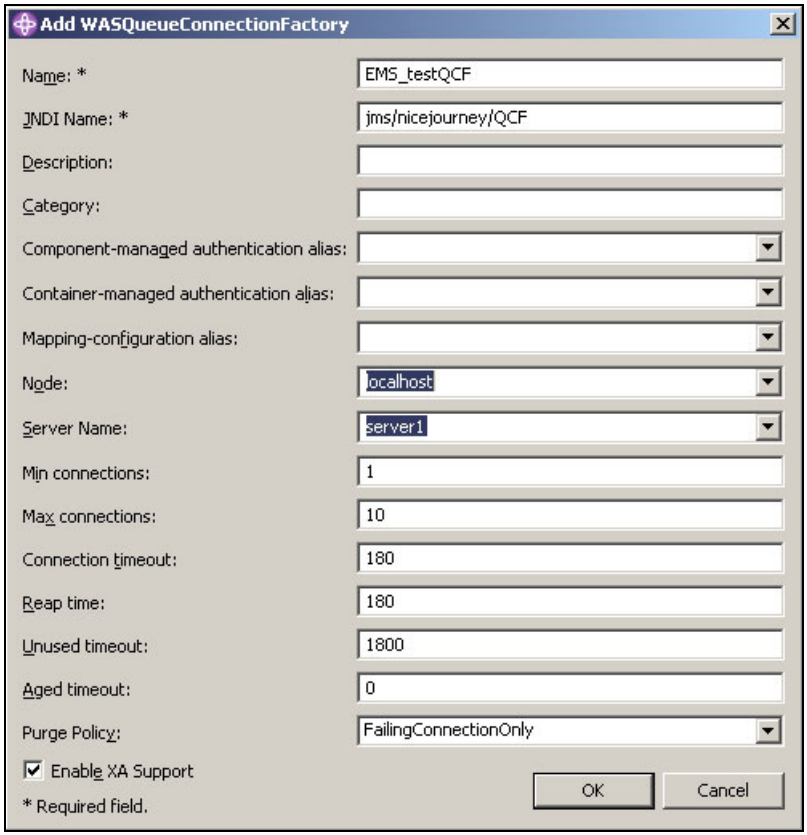


Figure 12-6 Queue Connection Factory settings

4. Create a queue for the server by clicking **Add** next to the WASQueue entries. Use the details shown in Figure 12-7 on page 365.

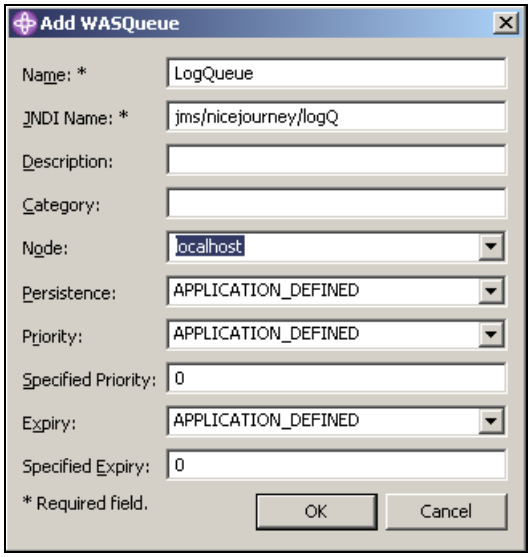


Figure 12-7 JMS Queue

5. Scroll down to the Server Settings section in the JMS window.
6. Add the new queue to the list by clicking the **Add** button and entering `LogQueue` in the text box.
7. Switch to the Extended Messaging tab.
8. Scroll down to the Node Settings section and click **Add** next to the list of Output Ports. Specify the settings shown in Figure 12-8 on page 366 and click **OK**.

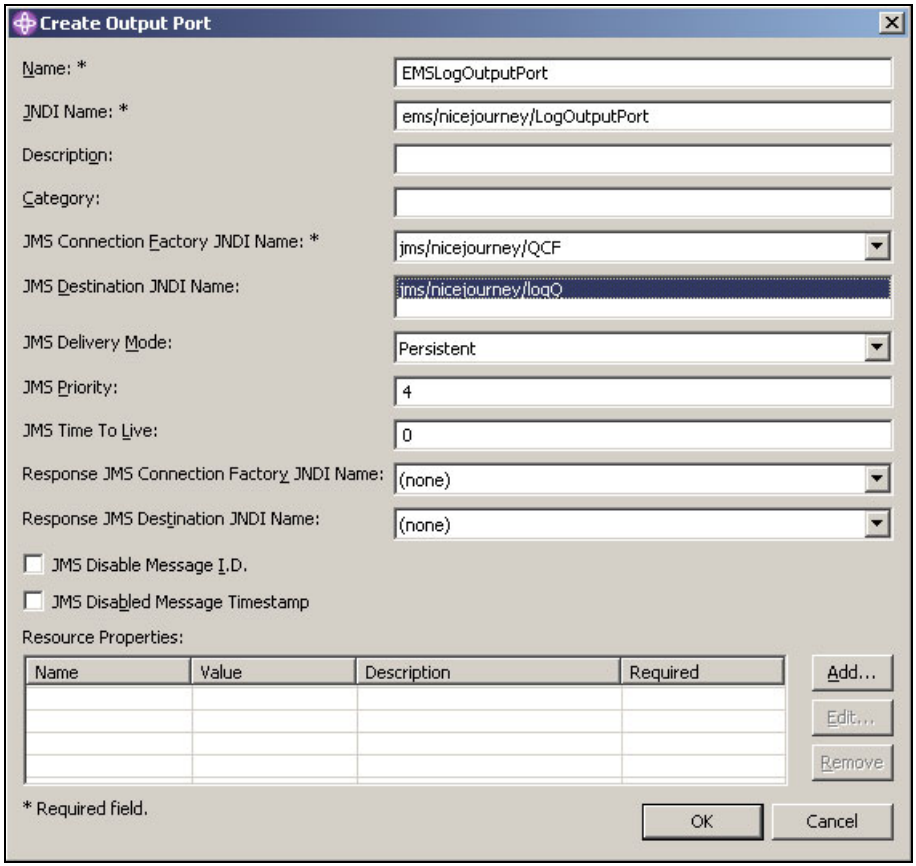


Figure 12-8 Creating the Extended Messaging Output Port

> **Note:** Make sure you select the JNDI name for the JMS destination, as shown in Figure 12-8, before you go any further.

9. Select the **Trace** tab and check that **Enable trace** is selected. Enter the following Trace string:

   ```
   com.ibm.ejs.jms.JMSQueueSenderHandle=all=enabled
   ```

10. Save and close the configuration.

11. Right-click **EMS_unitTest** in the workspace Servers view and select **Publish** from the menu. You should be presented with a message saying `Publishing was successful`.

12.4.2 Testing the LogSender in isolation

The following steps will show how to test the Sender bean in isolation, independently from other application components.

1. Start the EMS_unitTest server.
2. Check that the following messages appear in the console, and that no errors are reported.

```
Binding EMSLogOutputPort as ems/nicejourney/LogOutputPort
Binding EMS_testQCF as jms/nicejourney/QCF
Binding LogQueue as jms/nicejourney/logQ
The Extended Messaging service started successfully.
Starting application: EMS_loggingQueueEAR
Preparing to start EJB jar: EMS_loggingQueue.jar
Starting EJB jar: EMS_loggingQueue.jar
Application started: EMS_loggingQueueEAR
```

3. Switch to the J2EE perspective and select **LogSender** under the EJB Modules folder in the J2EE Hierarchy.
4. Right-click the **LogSender** entry and select **Run on Server...**. Select **Finish**.
5. Select **Test EJB local interface** from the window that is presented, then click **Finish**. The IBM Universal Test Client should start.
6. Expand the **LogSenderLocal** → **LogSenderLocalHome** tree in the References pane. Select **LogSenderLocal create()**.
7. Click the **Invoke** button in the Parameters pane, then click **Work with Object**. The test client should now look like Figure 12-9.

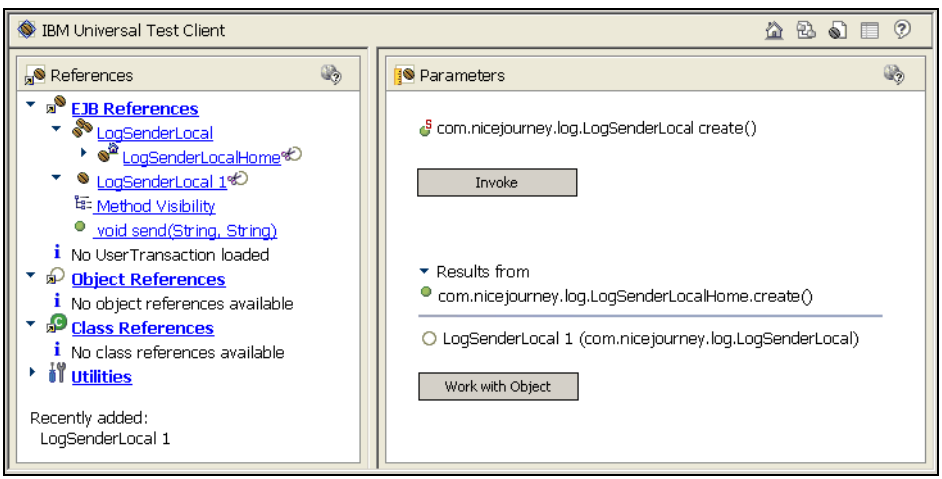


Figure 12-9 Test client ready to run the local interface

8. Select the **void send(String, String)** method from the References pane.
9. Enter the parameters as shown in Figure 12-10.

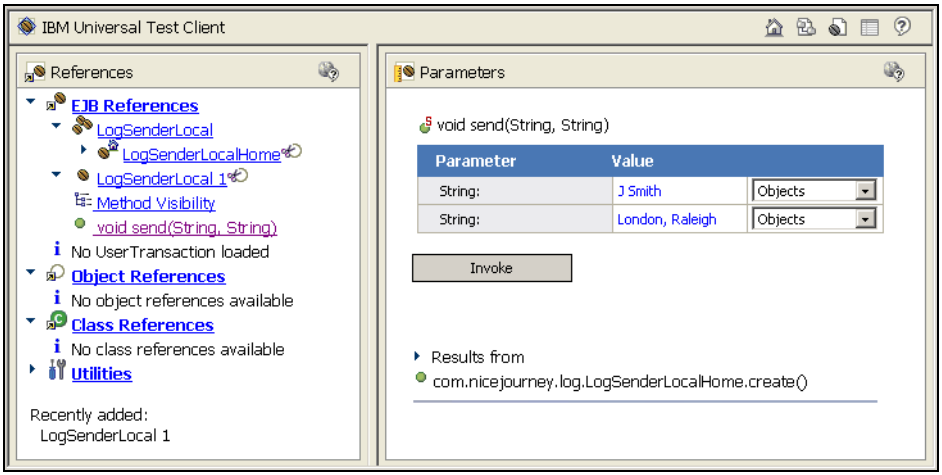


Figure 12-10 Prepare to run the send() method

10. Click the **Invoke** button.
11. You should see the message `The method completed successfully displayed` in the test client.
12. Review the trace.log file created by the server. This should be in the *<EMS_project_workspace>*\.metadata\.plugin\com.ibm.etools.server.core\tmp0\logs\server1 directory. You should see an entry similar to Example 12-2.

Example 12-2 trace.log output for unit test

```
[5/20/04 18:13:23:185 EDT] 606c51a1 JMSQueueSende > send
                                  queue:///WQ_LogQueue
JMS Message class: jms_stream
  JMSType:          LogMessage
  JMSDeliveryMode: 2
  JMSExpiration:    0
  JMSPriority:      4
  JMSMessageID:     null
  JMSTimestamp:     0
  JMSCorrelationID:null
  JMSDestination:   null
  JMSReplyTo:       null
  JMSRedelivered:   false
<stream><elt>J Smith</elt><elt>London, Raleigh</elt>
[5/20/04 18:13:23:195 EDT] 606c51a1 JMSQueueSende < send
```

12.4.3 Testing the Sender bean in the simple process

This section will test the Sender bean component as an integral part of the simple process.

1. Make sure the server is stopped by checking the Servers view.
2. Add the WPC_EMS_demoEAR project to the EMS_unitTest server.
3. Publish and start the server. Check that the console contains no errors.
4. Switch to the Servers view and right-click the **EMS_unitTest** server. Select **Launch Business Process Web Client** to start the BPE Web Client.
5. Select **My Templates** from the menu. Check the checkbox next to NiceJourney and click **Start Instance**.
6. Enter some message data in the form that is presented, then click **Start Instance**.

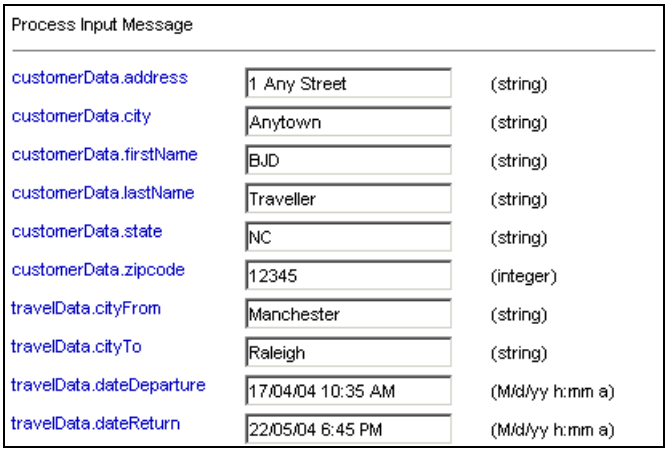


Figure 12-11 Input Message for NiceJourney process

7. You should see the output message stating the reservation ID.
8. Look into the trace.log file and check for the trace record as shown in Example 12-3.

Example 12-3 trace.log output for process invocation

```
[5/21/04 16:59:45:429 EDT] 34a971ce JMSQueueSende > send
                                    queue:///WQ_LogQueue
JMS Message class: jms_stream
  JMSType:         LogMessage
  JMSDeliveryMode: 2
  JMSExpiration:   0
  JMSPriority:     4
```

```
  JMSMessageID:    null
  JMSTimestamp:    0
  JMSCorrelationID:null
  JMSDestination:  null
  JMSReplyTo:      null
  JMSRedelivered:  false
<stream><elt>Traveller,BJD</elt><elt>Manchester to Raleigh</elt>
[5/21/04 16:59:45:449 EDT] 63c131cd JMSQueueSende < send
```

12.5 Assembly

There are a few settings that you can modify in your enterprise application (EAR) during assembly time. These settings are also available for the application at development time, of course.

When you have a Extended Messaging component in your EJB project, open the deployment descriptor for the EJBs in the Application Server Toolkit for WebSphere Business Integration Server Foundation or in WebSphere Studio Application Developer Integration Edition.

You should find an Extended Messaging tab in the EJB deployment descriptor editor as show in Figure 12-12 on page 371.

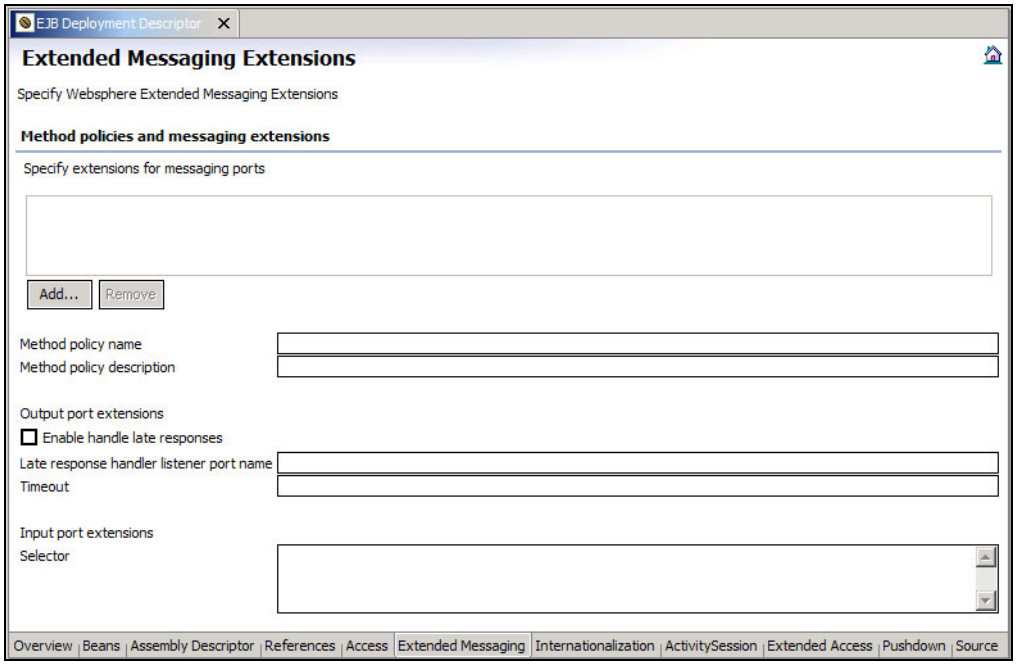


Figure 12-12 Extended Messaging tab in the EJB Deployment Descriptor editor

You can find more information about the Extended Messaging Extensions, method policies and handling late responses in the WebSphere Studio Application Developer Integration Edition help at WebSphere Business Integration Server Foundation, Extended messaging under the Highlights and Features section.

12.6 Deployment

In the WebSphere Business Integration Server Foundation runtime environment, the Extended Messaging configurations are available in the **Resources** → **Extended Messaging Provider** panel. You can configure input and output ports and custom properties for the provider.

The Extended Messaging service is enabled and starts up with the application server by default. You can change the behavior and disable the service by clicking **Servers** → **Application Server** → ***<your_server>*** → **Extended Messaging Service**. You can also configure listener port extensions for your Extended Messaging Service starting from this page.

Before you go further with the Extended Messaging configuation, you will need to configure your messaging provider for the application server under resources. You can use either embedded (WebSphere JMS), WebSphere MQ or other external messaging providers together with extended messaging.

Once you have the messaging provider and messaging components configured, you can configure your Extended Messaging components.

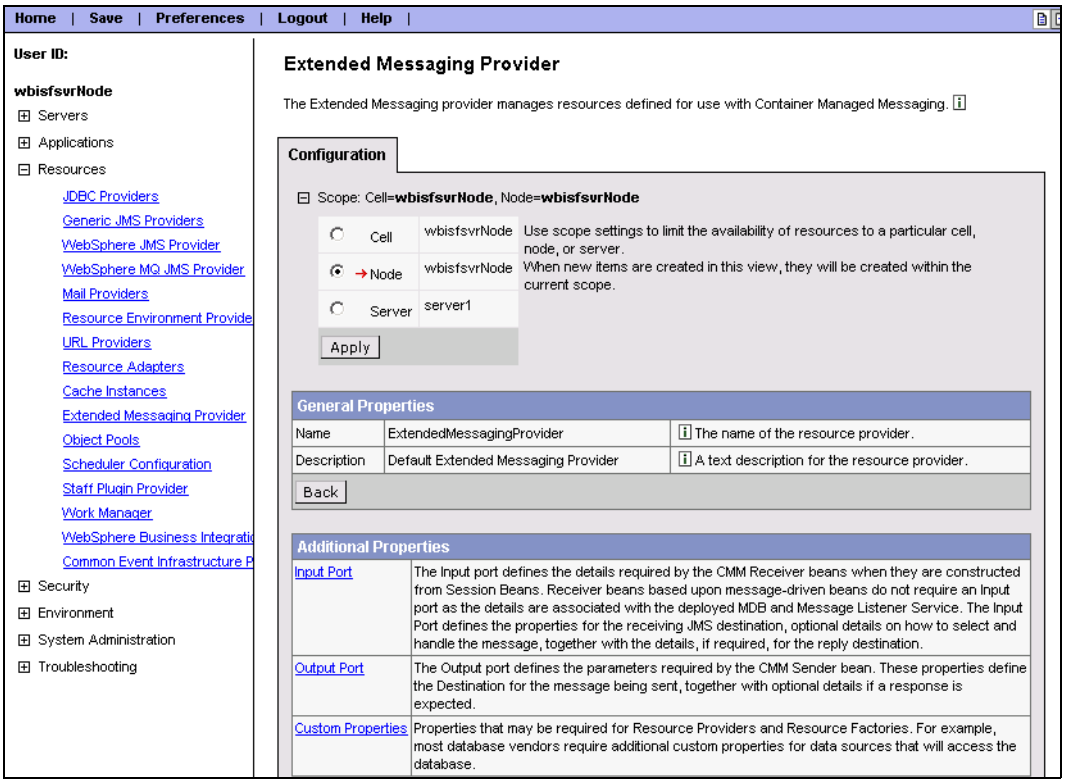


Figure 12-13 Extended Messaging provider configuration

To create a new Input port, select **Input Port**, then click **New**; you should get the page shown in Figure 12-14 on page 373.

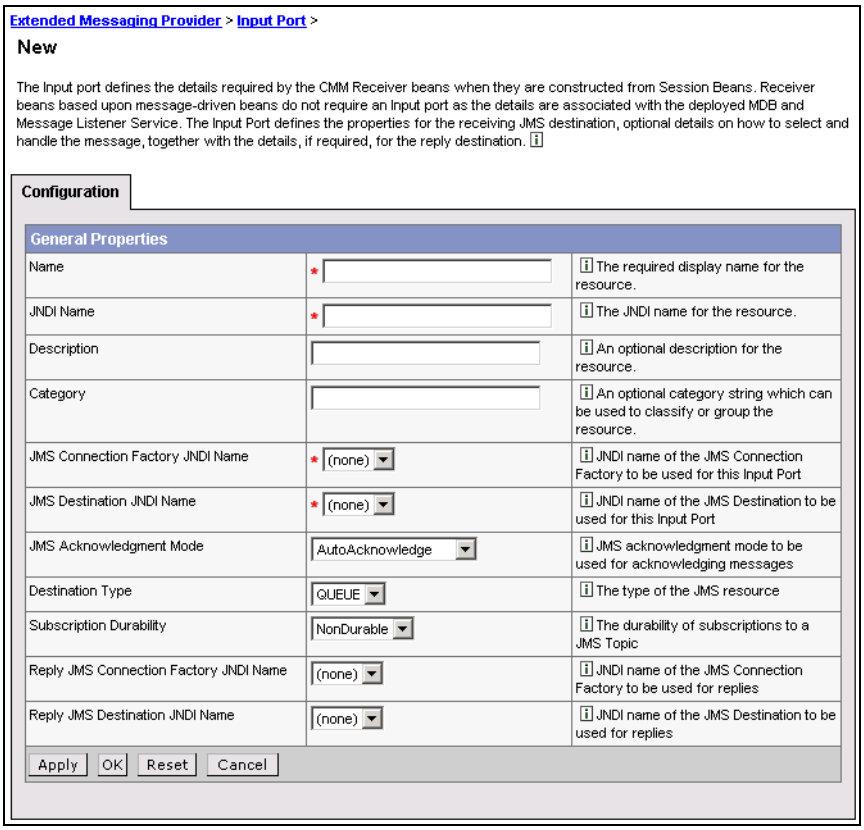


Figure 12-14 New Input port definition

To create a new Output port, select **Output Port** on the Extended Messaging Provider page, then click **New**; you should get the page shown in Figure 12-15 on page 374.

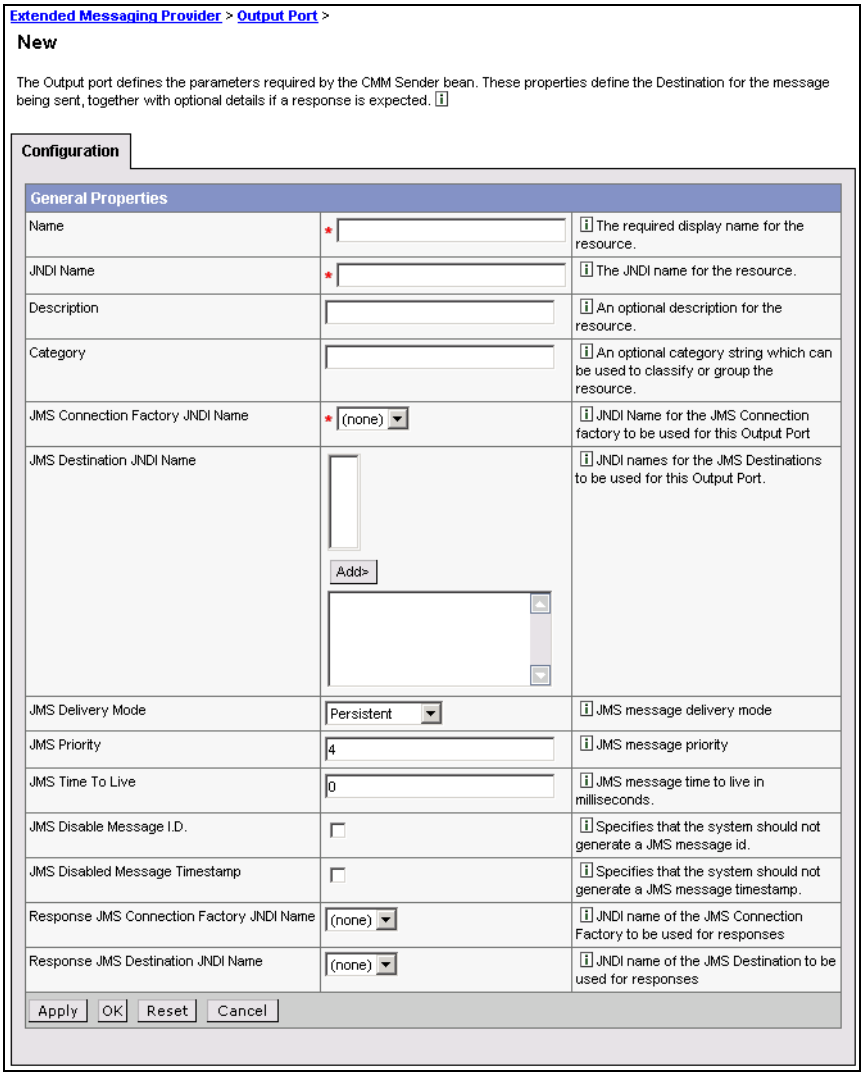


Figure 12-15 New Output port definition

Once you have finished the setup of Extended Messaging, save the configuration for WebSphere.

13

Chapter 13.

Startup beans

A Startup bean is a stateful session bean executed when an application starts. Startup beans enable J2EE applications to execute business logic automatically, whenever an application starts or stops normally.

Startup beans are used to perform initialization and cleanup tasks when a J2EE application is started or shut down. Startup beans have a number of uses. They can check application runtime dependencies such as the availability of a database or legacy system connection before application startup. They can also be used for data caching and to “warm up” entity beans, by executing a finder method and storing the EJB handles for future use.

Startup beans are particularly suited to asynchronous programming and they can be used for independent fast logging and for caching purposes. Where an application is using a large cache of objects but can operate without the cache, application processing does not need to be delayed until the full cache is built. By using a Startup bean, we can initiate an Asynchronous bean which populates the cache and starts the application.

Startup beans can also be used with the scheduler service. Before the application starts up, we can create a task and schedule it with the Scheduler service. A sample application illustrating this purpose can be found in Chapter 14, “Scheduler service” on page 391.

The benefits of using Startup beans are as follows:

- Initialization and cleanup tasks are contained within a single EJB
- Startup Beans run with full security context.
- Startup Beans run within WebSphere's name space. Therefore it uses the Java Naming and Directory Interface (JNDI) to find and use other resources.
- In comparison to using a servlet's init() method, the functionality of Startup beans also provides a stop() method that executes when the application is shut down.

13.1 Prerequisites

Here are some useful resources that will help you get started with Startup beans:

- WebSphere Business Integration Server Foundation V5.1 InfoCenter:

  http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp?topic=/com.ibm.wasee.doc/info/welcome_ee.html

- WebSphere Studio Application Developer Integration Edition V5.1 Help at the following locations:
  - **WebSphere Business Integration Server Foundation → Highlights and Features → Startup Beans**
  - **Cheat Sheets → Creating and Testing a Startup Bean**
- IBM Redbook: *WebSphere Application Server Enterprise V5 and Programming Model Extensions*, SG24-6932.

13.2 Sample scenario

This chapter demonstrates a simple example of a Startup bean being used to invoke a BPEL process on the startup of an application. For this scenario, we are invoking the Hello World process developed in “HelloWorld process application” on page 562 and printing the output to the Console.

Note: The final solution is also available as part of the additional material. You can import the startup.pi.zip file using the Project Interchange.

After importing the code you will get some error messages, they should disappear as soon as you generate the deployed code for the process.

Development

The development process consists of creating a Startup bean and implementing the start() and stop() methods as shown in 13.3, “Development” on page 378.

Unit testing

For Startup beans, no configuration in the WebSphere Test Environment is required since the Startup bean service is always enabled. When the development is finished, you simply publish the project that contains the Startup bean on the WebSphere Test Environment server.

Assembly

At Assembly time, you may need to specify properties such as Startup Bean priorities, security identity and transactional properties. For further information, refer to 13.4, “Unit test” on page 383.

Deployment

There are no specific deployment tasks for Startup beans.

Testing and runtime

Information regarding how the Startup Bean service operates in the runtime environment is in 13.6, “Runtime environment” on page 387.

13.3 Development

This section discusses how to create and use a Startup bean in an application. In this scenario, we will use a Startup bean to call an existing BPEL process to demonstrate how they can be used in conjunction with Process Choreographer.

Note: If you wish to use Startup beans for tasks not in conjunction with Process Choreographer, you can still use the code and techniques described in this chapter.

To add a Startup bean to the Hello World process, you need to:

- Import the Hello World process
- Create an EJB Project
- Fix the Java build path
- Create a new Startup session bean (EJB)
- Add Projects to the Java Build Path
- Add start() and stop() methods to the Startup bean

Follow these steps to develop the code for the Startup beans example:

1. Start WebSphere Studio Application Developer Integration Edition with an empty workspace, then make sure you enable server targeting.
2. Set up the Hello World BPEL process as specified in “HelloWorld process application” on page 562.
3. Generate the deployed code for the HelloWorld.bpel process.
4. Switch to the J2EE perspective, and create a new enterprise application with an EJB Project. Go to **File** → **New** → **Project** and select the **EJB** tab and **EJB Project**. Name the EJB project: `StartupEJB`, the enterprise application:

StartupEAR, and select **Integration Server v5.1** for Target server. Click the **Finish** button.

5. Create a new Startup session bean (EJB)

   A Startup Bean is a user-defined EJB 2.0 session bean. It can be either stateful or stateless. If it is stateful, the same instance is used for start and stop. Otherwise, two instances are created.

   The following home and remote interfaces are *mandatory*:

   – EJB home interface must be:

     `com.ibm.websphere.startupservice.AppStartUpHome`

   – EJB remote interface must be:

     `com.ibm.websphere.startupservice.AppStartUp`

   a. To create a new stateless Startup bean select **File** → **New** → **Other**, and select **EJB** on the left window and **Enterprise Bean** on the right, then click the **Next** button. Select **StartupEJB** from the drop-down list, then click **Next**.

   b. Type `HelloWorldStartup` in the Bean Name field, `com.ibm.itso.sg246318.startup` in the Package field and click the **Next** button.

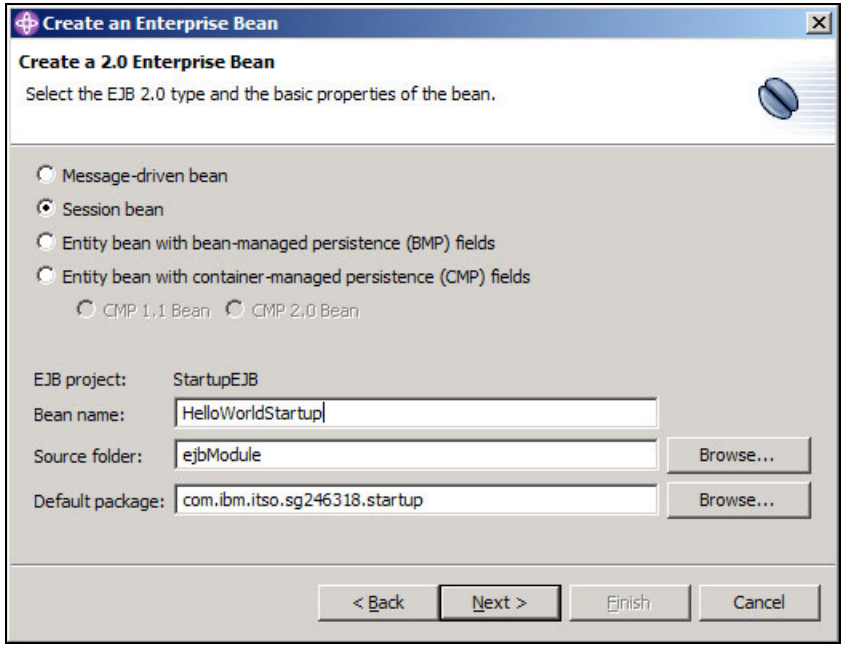


Figure 13-1 Creating a Startup Enterprise Bean

c. Type in `com.ibm.websphere.startupservice.AppStartUpHome` in the Remote home interface field and `com.ibm.websphere.startupservice.AppStartUp` in the Remote Interface field and click the **Finish** button.

6. Add Projects to the Java build path. Right-click **StartupEJB** project and select **Properties**. Select **Java Build Path** and the **Projects** tab. Select the **HelloWorldProcess** and **HelloWorldProcessEJB** and click the **OK** button.

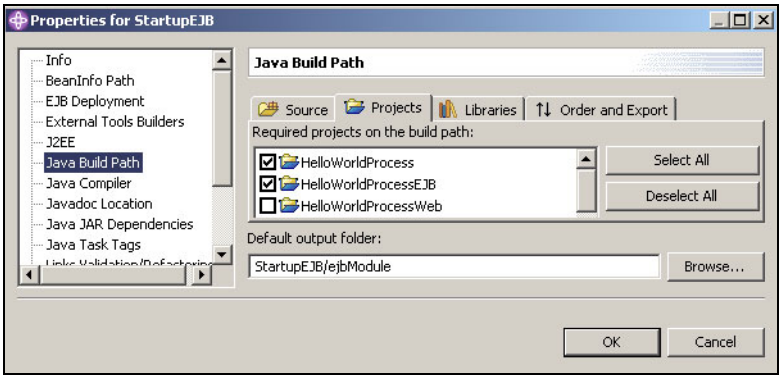


Figure 13-2 Adding projects to the Java build path

7. Add start() and stop() methods to the Startup bean.

   Now we will add two new methods called start and stop to MyStartupBean.java.

   The purpose of these methods are as follows:

   – **boolean start()**

     This method is called before an application starts. From this method we call all the user-defined initialization code we want to execute. The start() method returns true when the method runs successfully and does not throw any exceptions. Otherwise the start() method will return false and indicates that application start will be aborted.

   – **void stop()**

     This method is called before the application stops. Specifically it is called after you initiate the stop of the application and all the requests have been served. This method can call all the user-defined “cleanup” code we want to execute.

     Any Exceptions thrown by this method will be logged but ignored and the application will continue to stop.

What does the start() method do?

- In the start() method, we look up the reference of the existing Business Process bean (Hello World).
- Call the SayHello() method passing the input string “World”.
- The SayHello() method returns the string “Hello World” which is printed to the Console.

a. Open the HelloWorldStartupBean.java file and add the start() and stop() methods to the end of the source, as shown in Example 13-1.

Example 13-1 Implementation of the start() and stop() methods

```
//...
public boolean start() {
   String jndi_name_process = "java:comp/env/ejb/HelloWorldBean";
   System.out.println("************************************************");
   System.out.println("Startup bean: starting...");
   try {
      // Obtain the default initial JNDI context
      Context initialContext = new InitialContext();
      // Lookup the remote home interface of the BusinessProcess bean
      Object result = initialContext.lookup(jndi_name_process);
      // Convert the lookup result to the proper type
      HelloWorldHome processHome = (HelloWorldHome)
         javax.rmi.PortableRemoteObject.narrow(result,HelloWorldHome.class);
      // Get a reference to Hello World bean interface
      HelloWorld process = processHome.create();
      // Call the SayHello operation
      System.out.println("Startup bean: Executing SayHello() method");
      String response = process.SayHello("World");
      System.out.println("Startup bean: Finished executing SayHello()
method");
   } catch (Exception e) {
         System.out.println("Error in Startup bean: " + e.toString());
   }
   System.out.println("************************************************");
   return true;
}

public void stop() {
   System.out.println("************************************************");
   System.out.println("Startup bean: stopping...");
   System.out.println("************************************************");
}
//...
```

b. Also copy and paste these import statements at the top of HelloWorldStartupBean.java:

```
import javax.naming.*;
import process.helloworld.com.process91337437.*;
```

When you are developing a Startup bean for your own process, you will have to import the packages related to your process. The process EJB package names are unique to the process, you will get that after you have generated the deployed code for the process.

When you invoke a process, you will also need to instantiate message objects and build the message, those will require a couple more additional package imports as well.

8. At this point, you are done with creating the Startup bean and developing the code. You can generate the deployed code for the Startup bean.

   On the J2EE Hierarchy perspective, right-click the HelloWorldStartup EJB and select **Generate Deployment Code**.

13.3.1 Additional development considerations

Here are some additional considerations when developing Startup beans.

Transactional considerations

The startup session bean must use Container Managed Transactions. On the stop() and start() methods, any transactional attribute can be used except TX_MANDATORY because the methods are not started from a thread that has a defined transactional context. If you use the TX_MANDATORY attribute, an exception is thrown and the application start is aborted.

Refer to 13.6, “Runtime environment” on page 387 for more information.

Security considerations

There are no specific security settings for Startup Beans.

If you use security enabled for WebSphere, the default identity for Startup Bean is *system*. This means if you do not specify any deployment-related security settings on the Startup Bean, the identity of the Startup Bean will be the same as the one used for starting WebSphere.

If your Startup bean uses some secured object that requires a different caller identity, then the security identity must be enabled on the Startup Bean and Run-as on the Startup Beans start() and stop() methods must be configured to use the correct identity.

Required session bean time-out

Set the time-out to 0. If it is set to any other value it will be ignored since startup service automatically checks and sets it to 0.

13.4 Unit test

Upon application startup, WebSphere will recognize and run Startup beans.

To deploy the Startup Bean, you need to:

- Modify EJB Deployment Descriptor for Startup bean
- Modify EAR Deployment Descriptor for HelloWorldProcess
- Create, configure and start the server

Follow the steps below to deploy the Startup bean.

1. Generate Deploy Code for Startup bean

   In the Business Integration perspective, right-click StartupEJB and select **Generate** → **Deployment and RMIC Code**. Ensure **Startup** is selected and click the **Finish** button.

2. Add an EJB reference in the Startup bean descriptor.

   a. Switch to the J2EE perspective, right-click StartupEJB and select **Open With** → **Deployment Descriptor Editor**.

   b. Select the **References** tab and select **HelloWorldStartup**. Click the **Add** button and select the first option EJB reference and click the **Next** button.

   c. Click Enterprise bean in different EAR and navigate to **HelloWorldProcessEAR** → **HelloWorldProcessEJB** → **HelloWorldBean**. Click the **Finish** button.

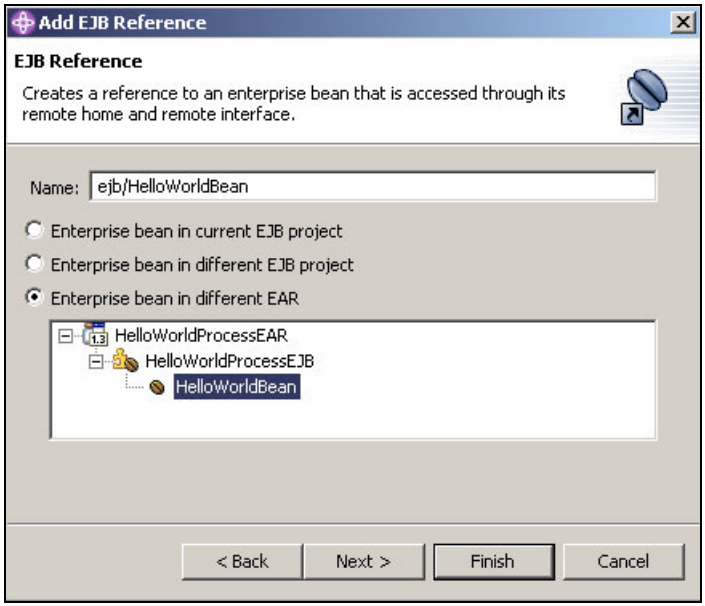


Figure 13-3 Adding an EJB reference

d. Save and close the file.

3. Create, configure and start the test server.

   a. Create a new server called `StartupServer`.

   b. Add the HelloWorldProcessEAR and the StartupEAR applications to the server.

   c. Open the Server configuration, switch to the **Configuration** tab and change the Application class loader policy to `SINGLE`.

   > **Note:** Changing the class loader policy ensures that the applications can "see" eachother. Technically speaking, in our case, the Startup bean can access the classes packaged with the process.
   >
   > Another solution would be, if you want to keep the classloader policy multiple, to create a new EJB module under the process enterprise application and code your startupbean there and add the process EJB jar as a dependency module to the startup EJB jar.

   d. Start the server.

In Example 13-2, you can see part of what is displayed in the test environment server output console. For more detailed traces of startup service execution for this sample, refer to 13.7, “Problem determination” on page 390.

Example 13-2 Output from Startup bean on the Console

```
...
SystemErr      ************************************************
SystemOut      Startup bean: starting...
SystemOut      Startup bean: Executing SayHello() method
SystemOut      Hello World
SystemOut      Startup bean: Finished executing SayHello() method
SystemErr      ************************************************
...
```

13.5 Assembly

At assembly time, you may need to specify the following settings for Startup Beans:

- Startup Bean priorities
- Security identity
- Transactional properties

Startup bean priorities

If there is more than one Startup Bean in your application, you can specify priorities among them. See 13.5.1, “Priorities when using multiple Startup beans” on page 386 for further details.

Security identity

You can set security for the bean using the Application Server Toolkit. You will need to install this as it is a separate executable from WebSphere Studio Application Developer Integration Edition V5.1.

In the J2EE perspective, double-click **StartupEJB** to open the EJB Deployment Descriptor. On the Assembly Descriptor tab, you can create roles, define method permissions (you can set security on your start and stop methods) and set transactional properties on these methods as seen in Figure 13-4 on page 386.

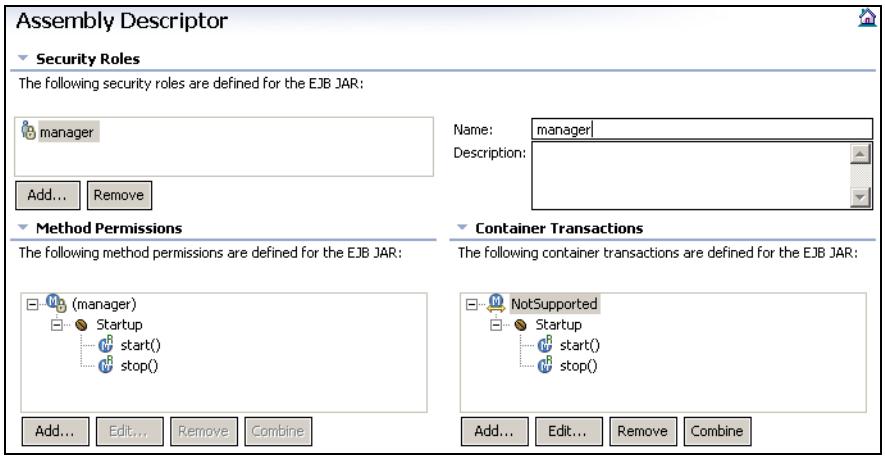


Figure 13-4 Setting security for the Startup EJB

For further information, consult the appropriate topics in the Help at **WebSphere Studio → Developing → EJB Applications → Editing the EJB Deployment Descriptor → Defining the assembly settings**.

JNDI considerations

During application assembly, we can specify a JNDI name for the Startup Bean. However, Startup beans are not exposed in JNDI this is not required since the only client for Startup Beans is the application server runtime.

13.5.1 Priorities when using multiple Startup beans

When an application has more than one Startup Bean, an environment property must be defined on each Startup Bean. The name of this priority must be wasStartupPriority and its type must be java.lang.Integer. In Figure 13-5 on page 387, you can see how to add an environment property in WebSphere Studio or in the Application Server Toolkit.

The default priority of a Startup Bean is 0. The property can be looked up using JNDI via java:comp/env/wasStartupPriority.

Within one application's scope, a Startup Bean with a higher priority will be executed first. Beans with the same priority will be executed in undefined order. Beans are stopped in the reverse order that they are started in.

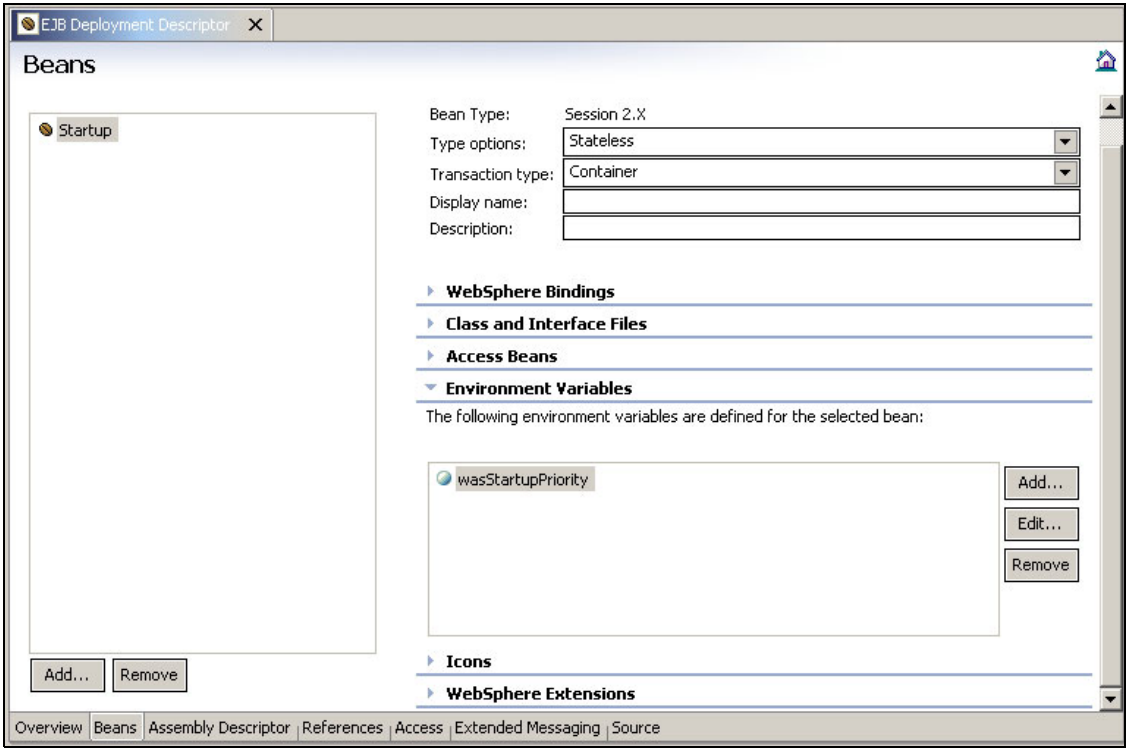


Figure 13-5 Adding the wasStartupPriority environment property to the Startup Bean

13.6 Runtime environment

This section describes how Startup Bean service behaves from the runtime point of view.

Startup service runtime flow

The following steps represent the different states in the life cycle of a Startup Bean. A diagram is shown in Figure 13-6 on page 389.

1. **Application startup**

   After the application's EAR is loaded, the service looks for all of the Startup Beans and finds them using the home interface. It then checks whether the property wasStartupPriority is defined for the startup EJB in the Deployment Descriptor. If there is no priority property defined or it is the wrong type, it would be set to the lowest possible value for the integer.

Startup Beans are ordered according to priority and the startup service takes the bean with the highest priority from the queue. It checks its transactional properties on the start() and stop() methods. If a TX_MANDATORY exception is thrown, the startup of the application is aborted. Then it finds the actual bean using JNDI and runs its start() method.

If there is no problem with the lookup and the start() method returned true, the startup service stores the bean handle and proceeds to next Startup Bean in the ordered queue.

When it finishes successfully with all the Startup Beans, WebSphere starts the application and its JMS, IIOP and HTTP listener.

Tip: The Startup service waits until the start() method finishes its execution. If there is a demanding task triggered by the start() method, it can take some time before the application starts. In these situations, it is recommended that you use Asynchronous Beans to perform the task so application startup is not unnecessarily delayed.

2. **Application shutdown**

   In the application shutdown process, after the containers stop serving requests Startup Beans are taken one by one in the reverse order of precedence they were run when the application started. For each bean, the stop() method is executed. Any exception thrown at this stage is ignored and the startup service continues to run the next bean in the queue.

3. **EJB Module re-start**

   Suppose we have an application with more than one EJB module and each contains Startup Bean(s). If one of the EJB modules is stopped and restarted, the Startup service will consider it an application start. Therefore, it will execute the Startup Beans that are part of the restarted EJB module.

4. **Application server crash**

   If the application server crashes, the Startup Bean stop() method may not be executed. When restarting the server, the start() method will be executed as normal.

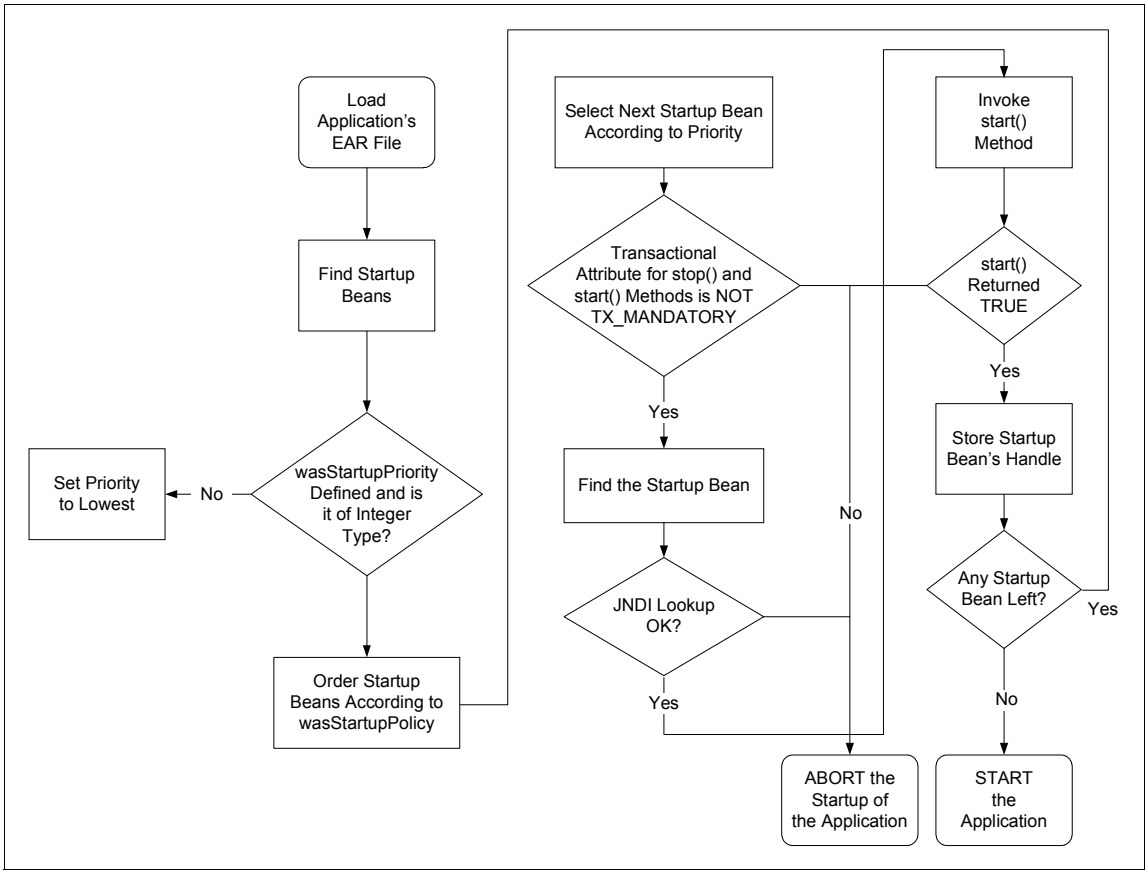


Figure 13-6 Startup Bean service runtime flow on startup

13.6.1 Scalability

If we use Startup Beans on clustered applications, multiple clones will run the applications. Therefore, every clone will have a Startup Bean instance of the application. The Startup Bean is invoked for each clone in a cluster.

Keep in mind that the Startup Bean service of a clone is not aware of other clones and does not synchronize any actions between instances of Startup Beans amongst clones. Therefore it is the programmer's responsibility to take the appropriate actions to avoid deadlocks or data inconsistencies.

13.7 Problem determination

If there is a problem with the configuration of the Startup Beans(s), the application will not start. The first thing to check is the transactional and security settings.

If you encounter problems using Startup Beans and you suspect that the problem source is within the startup service itself, you can use the following trace string to monitor what is happening during execution.

```
com.ibm.ws.startupservice.*=all=enabled
```

14

Chapter 14.

Scheduler service

The Scheduler service was previously introduced in WebSphere Application Server V5.0 Enterprise Edition. This service extends the normal J2EE services by introducing time-driven actions. These actions can be scheduled to run once, in the future, or on a recurring basis at regular intervals.

The advantages of using the Scheduler service are as follows:

- The service facilitates the development of date- or time-based functionality. This feature of WebSphere Business Integration Server Foundation can be applied to various scenarios which are listed in this chapter as well.
- Scheduler server administration is consistent with WebSphere resource management.
- Scheduler tasks can be persisted. This means that the Scheduler service stores the task definition in relational database (RDB) tables. These tables must be created ahead of time. RDB ensures that the task definition is retained as WebSphere Business Integration Server Foundation processes are recycled, in a single machine or cell configuration.
- The Scheduler service can be workload-managed for high availability configurations and for performance.
- Calendars are used to calculate the timing for a given scheduler task. Out-of-the-box calendars are shipped with WebSphere Business Integration Server. However, customized, business-oriented calendars can be developed to suit specific scheduler needs.

- The Scheduler task design is very flexible. Scheduled actions can be EJB-based or can be triggered using JMS messages.
- Scheduler can perform notification actions on various task activity events.

Typical usage scenarios for WebSphere Business Integration Server Foundation scheduler service are as follows.

- Any time certain actions must be performed on a regular basis.
- When scheduling regular back-ups, clean-ups, night-time or batch processing.
- When implementing automatic, non-user-driven activities. These could be primarily activities tied to the back-end system integration, for example, kick-starting a business process in Business Process Container of WebSphere Business Integration Server Foundation using the JMS triggering mechanism mentioned earlier.

14.1 Prerequisites

You can familiarize yourself with the following WebSphere Business Integration Server Foundation topics in InfoCenter at:

http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp

- For information about installing WebSphere Business Integration Server Foundation, see:

  http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/info/ee/ae/welc_installing.html

- For information about administering a Scheduler service in WebSphere Business Integration Server Foundation, see:

  http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/info/ee/scheduler/tasks/tsch_managescheduler.html

- It is important to gain familiarity with Scheduler tasks because the section 14.3, “Development” on page 395 assumes this knowledge. Using WebSphere Business Integration Server Foundation InfoCenter, navigate to **Developing** → **Applications** → **Application Services** → **Developing and scheduling tasks** in:

  http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/info/ee/scheduler/tasks/tsch_developtasks.html

Familiarize yourself with the managing of Programming Model Extensions by clicking **WebSphere Studio** → **Deploying Business Integration application** → **Deploying business integration application** → **Managing Programming model extensions** in the WebSphere Studio Application Developer Integration Edition Help Content.

14.2 Sample scenario

The scenario used in the following sections is meant to develop a background batch job using the WebSphere Business Integration Server Foundation Scheduler service. This background job will invoke a business process at specific interval of times. The example will illustrate the following key points:

- How to invoke a business process from a Java client.
- How to implement the Scheduler service in conjunction with the Startup bean service.
- How to receive notifications from the Scheduler service.
- How to programatically cancel and stop scheduled jobs.

Note: The sample application developed in this chapter is available as part of the additional material. You can import the scheduler.pi.zip into WebSphere Studio Application Developer Integration Edition using the Project Interchange plug-in.

After import, you will get some error messages. You have to generate the deployed code for the HelloWorld process.

You will also have to generate the deployed code for the EJBs before you can run the application.

Migration

The WebSphere Enterprise V5.0 Scheduler service has interoperability issues that affect the behavior of this service in a clustered environment. It is important to familiarize yourself with these issues. These are listed in the WebSphere Business Integration Server Foundation InfoCenter:

http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/info/ee/scheduler/tasks/tsch_interop.html

Development

The development of a Scheduler service consists of implementing a Task session bean and developing the code that will register this Task session bean in the Scheduler service (or the persistent store). The Scheduler daemon will then take care of invoking this task at specified intervals.

The role of the Scheduler service is to fire off tasks at specified times or intervals. Scheduler tasks can:

- Invoke a user-defined Stateless Session EJB
- Send a JMS message to a user-defined JMS queue or topic

Once the task is scheduled, the Scheduler service stores the task definition in the relational database table. The database and schema for the table must exist ahead of time. This ability to persists tasks in the database allows for the Scheduler to be configured in a clustered/cell configuration.

The Schedule service also supports the notion of *calendars*. Calendars are needed to schedule a task. The primary purpose of calendars is to calculate a point in time in the future when a scheduled task is to be executed. If no calendars are specified when a task is being scheduled, a default calendar is used. Two out-of-the-box calendars are provided; each of these is discussed later in this chapter.

- Arithmetic calendar
- CRON-based calendar

Unit test

As custom Scheduler tasks are developed, there can be tests within the WebSphere Test Environment in WebSphere Studio Application Developer Integration Edition V5.1. For details about WebSphere Test Environment, refer to 14.4, “Unit test” on page 407.

Assembly

Scheduled tasks, at the application code level, use local references to refer to the Scheduler service. Keep that in mind, before the application can use the service, the local JNDI name for the Scheduler must be specified. If the name of the Scheduler service is different in production environment compared to unit test environment, which would most likely be the case, Application Server Toolkit can be used to specify the JNDI name.

Deployment

As mentioned previously, Scheduler tasks and calendars are stateless session beans and should be treated as such when it comes to packaging, deployment and usage. There are no specific steps related to the deployment of Scheduler beans in the WebSphere Business Integration Server Foundation runtime.

Testing and runtime

Scheduler task and calendars can be tested in the unit test environment. However, for functional and integration tests, Scheduler components should be further tested in the runtime environment. Runtime environment is configured using the Administrative Console and primarily contain a scheduler daemon.

14.3 Development

The following section describes how to use the Scheduler service from an Java application.

In order to invoke a scheduled task, first a `BeanTaskInfo` object, set the properties on this object, and the appropriate `TaskHandler` that will be invoked when the scheduler kicks in. The final step is to schedule the task in the Scheduler service persistent storage.

In Chapter 13, “Startup beans” on page 375, the bean’s start() method implemented Java code to invoke a Business Process. However, in this section, the Startup bean code in start() will be changed to instead schedule a task.

The Task session bean, `BatchProcessBean`, developed in this section will execute exactly the same code as in the Startup Bean, `StartupBean`, in Chapter 13, “Startup beans” on page 375. That is, `BatchProcessBean` will invoke a Business Process.

In addition to changing the start() method code in StartupBean to schedule the task, stop() method will include code that illustrate how to Programatically cancel and purge a scheduled task that is either running, suspended, or any other state.

Additionally, a Notification bean is developed that demonstrates different notification events that are thrown by the scheduler demon and that be captured via in a session bean.

The Scheduler API is contained in the com.ibm.websphere.scheduler package. Check the complete description of the Scheduler API in the WebSphere Business Integration Server Foundation InfoCenter:

`http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/info/ee/javadoc/ee/com/ibm/websphere/scheduler/package-summary.html`

14.3.1 Steps for using the Scheduler API

There are no tools that provide help creating and scheduling tasks. All must be done programmatically in your J2EE application. However, administrative console can be used to configure the Scheduler service and resources. The following steps must be performed when developing the Scheduler service:

1. Developing a task

   Scheduler API supports two different ways to develop a Task, effectively a `TaskInfo` interface. Each of these ways can be used to schedule a particular type of work.

   a. Task that calls a session bean by implementing the process() method in the `com.ibm.websphere.scheduler.TaskHandler` remote interface. Place the business logic you want created in the process() method. The process() method is called when the task fires. The Home and Remote interfaces must be set as follows in the bean's deployment descriptor:

      - home: `com.ibm.websphere.scheduler.TaskHandlerHome`
      - remote: `com.ibm.websphere.scheduler.TaskHandler`

   b. Task that sends a Java Messaging Service (JMS) message. This task object can send a JMS message to either a queue or a topic. This activity consequently requires the creation of an instance of the `MessageTaskInfo interface` by using the `Scheduler.createMessageTaskInfo()` method.

2. Define the notifications.
3. Define the calendar.

4. Submit the task to a scheduler.
5. Manage the tasks with a scheduler.

Each of these steps is discussed in the following sections.

Configuring the Scheduler service

The first step is to have the Scheduler service and its prerequisites configured as the work manager and Scheduler database. This can be done using the WebSphere administration. Note that this step pertains more to the administrative tasks and is not necessarily tied to the development process of a scheduler. Scheduler task can be developed in parallel as long as the JNDI name for the Scheduler is known in advanced. Refer to 14.7.1, “Problem determination” on page 411

Important: Remember the JNDI name you used for the Scheduler service configuration.

Developing the Scheduler code

This step consists of developing a scheduled task, defining optional notification, defining optional calendar, submitting the task to a scheduler. Scheduled tasks can be managed programmatically. These steps will not be discussed in detail in this chapter since they are well documented in the WebSphere Business Integration Server Foundation InfoCenter:

http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/info/ee/scheduler/tasks/tsch_developtasks.html

Note: It is important to note that defining notification and custom calender is optional. Tasks can be scheduled without a notification mechanism or without providing a calender. WebSphere Business Integration Server Foundation implements two calendars that cover the majority of the needs. These calendars are used by default.

Note: Tasks are managed programmatically by using the scheduler API. There is no mechanism in place to delete the task except through the API.

Restriction: The Scheduler is available only to server-side components where the Scheduler daemon is running. If a Scheduler resource is defined at the node level, for example, all the application servers on that node have access to that Scheduler and will run its daemon. You can selectively disable the daemon on an application server.

14.3.2 Using Scheduler with Process Choreographer

As discussed in section 14.2, “Sample scenario” on page 393, the sample scenario uses Startup bean to register a scheduler task. The scheduler task then executes at specified intervals of time and invokes the same Business Process used in the Chapter 13, “Startup beans” on page 375.

Prerequisites

The following prerequisites must be met before developing for the Scheduler service:

- Set up the Hello World BPEL process as specified in “HelloWorld process application” on page 562.
- Have the Startup bean EJB project imported in the workspace prior to working with Scheduler chapter. Complete steps in Chapter 13, “Startup beans” on page 375 to create a StartupBean; refer specifically to 13.3, “Development” on page 378.
- For testing the sample with WebSphere Studio Application Developer Integration Edition, a test server needs to be created and configured. For more information, refer to 14.4, “Unit test” on page 407.
- The scheduler-client.jar and scheduler-server.jar files needs to be in the Java build path of the EJB module that will contain the Scheduler Bean.
- The Scheduler service will be used from the Startup Bean. For the startup EJB, you need to define a resource reference that points to the Scheduler service.

Creating the task session EJB

Perform the following steps to create the Task.

1. Launch WebSphere Studio Application Developer Integration Edition with an empty workspace. Make sure you enable server targeting.
2. Import the HelloWorld process into the workspace using the Project Interchange plug-in, for more information refer to, “HelloWorld process application” on page 562.
3. In the J2EE hierarchy window, right-click **EJB Modules**. From the context-menu, select **New** → **EJB Project.**
4. In the Select an EJB version window, select **Create 2.0 EJB** Project. Click **Next**.
5. In EJB Project window, type **Scheduler** as the name of the project. Select **SchedulerEAR** as the EAR Project. Select **Integration Server V5.1** for server target. Click **Next**.

6. Right-click the **Scheduler** EJB project and select **Properties**. Under Java Build path, switch to the Projects tab and check the box next to HelloWorldProcess and HelloWorldProcessEJB. Click OK.

7. Right-click the **Scheduler** EJB Project and from the context-menu select **New** → **Enterprise Bean**.

8. In Create an Enterprise Bean window, make sure that Scheduler EJB Project is selected. Click **Next**.

9. In the *Create a 2.0 Enterprise Bean* window, select the values as specified in Figure 14-1.

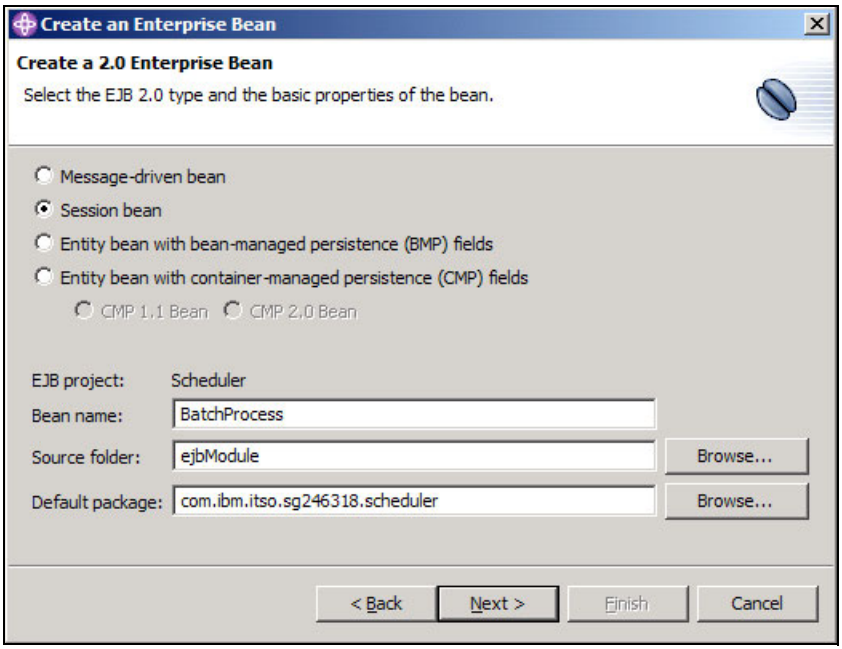


Figure 14-1 Create Enterprise Bean: Project and Bean name

10. In the Enterprise Bean Details window, accept the defaults but change the Remote home interface and Remote interface as follows.

    – Remote home interface: `com.ibm.websphere.scheduler.TaskHandlerHome`
    – Remote interface: `com.ibm.websphere.scheduler.TaskHandler`

    Click **Finish** when done.

11. Still in the J2EE Hierarchy view, expand the newly created **BatchProcess** Session bean and locate BatchProcessBean Java file. Open BatchProcessBean.java and add the following code to create process() method.

Example 14-1 BatchProcessBean.java

```
//...
import javax.naming.*;
import process.helloworld.com.process91337437.*;
import com.ibm.websphere.scheduler.TaskStatus;

//...

public void process(TaskStatus status) {
   String jndi_name_process = "java:comp/env/ejb/HelloWorldBean";
   System.out.println("************************************************");
   System.out.println("Scheduler bean: starting...");
   try {
      // Obtain the default initial JNDI context
      Context initialContext = new InitialContext();
      // Lookup the remote home interface of the BusinessProcess bean
      Object result = initialContext.lookup(jndi_name_process);
      // Convert the lookup result to the proper type
      HelloWorldHome processHome = (HelloWorldHome)
         javax.rmi.PortableRemoteObject.narrow(result,HelloWorldHome.class);
      // Get a reference to Hello World bean interface
      HelloWorld process = processHome.create();
      // Call the SayHello operation
      System.out.println("Scheduler bean: Executing SayHello() method");
      String response = process.SayHello("World");
      System.out.println("Scheduler bean: Finished executing SayHello()
method");
   } catch (Exception e) {
         System.out.println("Error in Scheduler bean: " + e.toString());
   }
   System.out.println("************************************************");
}
```

12. Save and close the file.

Adding an EJB reference

Next, we will add a EJB reference to the HelloWorld Business Process.

1. Double click the **Scheduler EJB Project** to bring up the EJB Deployment Descriptor for this project. Switch to the **References** tab.
2. Select **BatchProcess** session bean and click **Add** on the References tab.
3. Select **EJB Reference** then click **Next**.
4. Select **Enterprise Bean in a different EAR,** then select the **HelloWorldProcessEAR** → **HelloWorldProcessEJB** → **HelloworldBean**.
5. Click **Finish** to create the EJB reference.

Important: Take note of the JNDI name: `process/helloworld/com/process91337437/HelloWorld20030101T000000`. This is the JNDI name for the process that NiceJourney Session bean will invoke.This JNDI may be different in the Runtime environment hence it could be changed here prior to exporting the EAR. This process is discussed in 14.5, “Assembly” on page 410.

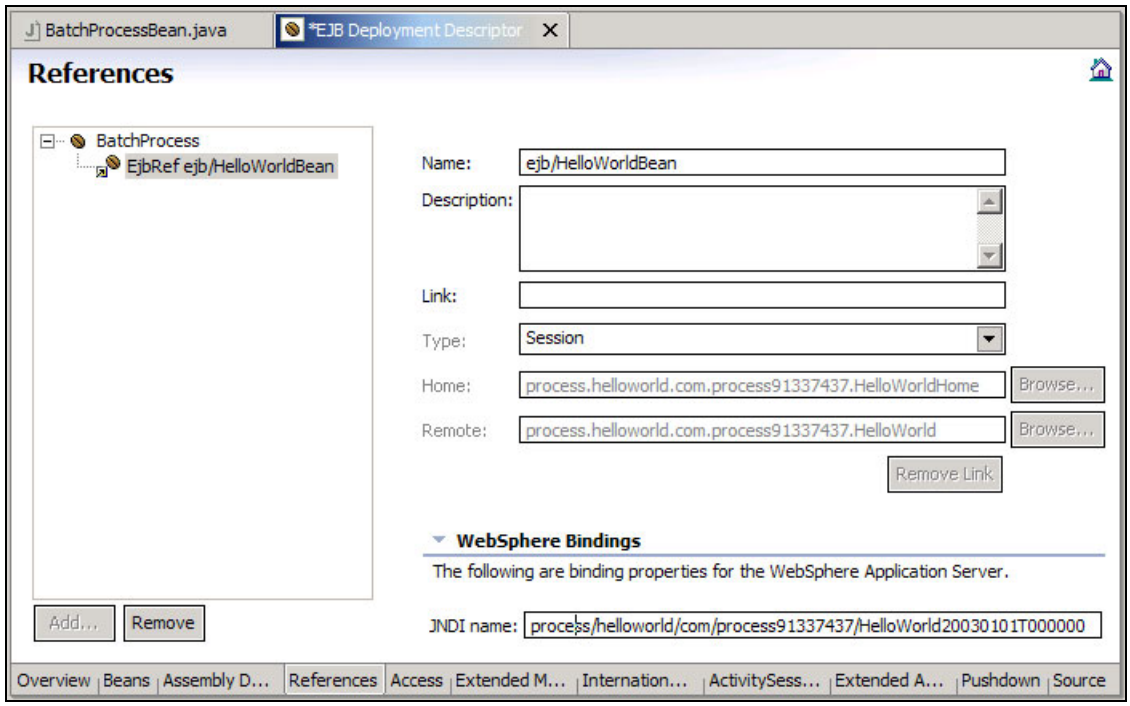


Figure 14-2 Scheduler EJB Project Deployment Descriptor

14.3.3 Notification bean

You can set a notification sink on a task in order to receive the notification events that are generated by a scheduler when it performs an operation on the task.

Follow the steps below to create ReceiveNotification bean. This bean will be made part of the Scheduler EJB project.

1. In the J2EE Hierarchy view, right-click the **Scheduler** EJB Project. From the context-menu, select **Enterprise Bean**.

2. In the Create an Enterprise Bean window, make sure the **Scheduler** EJB Project is selected and click **Next**.
3. In the Create a 2.0 Enterprise Bean window, accept the default values except the bean name and project. Enter `ReceiveNotification` as the bean name and `com.ibm.itso.sg246318.scheduler` as the project name.
4. In Enterprise Bean Details window, accept the defaults except change the Remote home interface and Remote interface as follows.

   Remote home interface:
   `com.ibm.websphere.scheduler.NotificationSinkHome`

   Remote interface: `com.ibm.websphere.scheduler.NotificationSink`

   Click **Finish** when done.
5. In the J2EE Hierarchy view, expand the newly created **ReceiveNotification** session bean and locate the ReceiveNotificationBean Java file. Open the ReceiveNotificationBean.java file and add the following code to create handleEvent() method.

Example 14-2 ReceiveNotificationBean.java

```
import javax.naming.*;
import com.ibm.websphere.scheduler.*;

//.....

public void handleEvent(TaskNotificationInfo task)  {
System.out.println("-------------------------------------------------------------------------");
    try {
        int eventType = task.getEventType();
        System.out.println("ReceiverNotificationBean.handleEvent(): Event received at " +
task.getTime());
        switch (eventType) {
            case TaskNotificationInfo.CANCELLED:
                System.out.println("ReceiverNotificationBean.handleEvent(): Task CANCELLED");
                break;
            case TaskNotificationInfo.COMPLETE :
                System.out.println("ReceiverNotificationBean.handleEvent(): Task COMPLETED");
                break;
            case TaskNotificationInfo.FIRE_DELAYED :
               System.out.println("ReceiverNotificationBean.handleEvent(): Task FIRE_DELAYED :
Task was unable to start and hence skipped");
                break;
            case TaskNotificationInfo.FIRE_FAILED :
              System.out.println("ReceiverNotificationBean.handleEvent(): Task FIRE_FAILED : Task
fired but TaskInfo threw an unexpected exception when executing");
                break;
            case TaskNotificationInfo.FIRED :
```

```
                System.out.println("ReceiverNotificationBean.handleEvent(): Task FIRED");
                break;

            case TaskNotificationInfo.FIRING :
                System.out.println("ReceiverNotificationBean.handleEvent(): Task is FIRING");
                break;
            case TaskNotificationInfo.PURGED :
                System.out.println("ReceiverNotificationBean.handleEvent(): Task PURGED from
persistant store");
                break;
            case TaskNotificationInfo.RESUMED :
                System.out.println("ReceiverNotificationBean.handleEvent(): Task was RESUMED");
                break;
            case TaskNotificationInfo.SCHEDULED :
                System.out.println("ReceiverNotificationBean.handleEvent(): Task was
SCHEDULED");
                break;
            case TaskNotificationInfo.SUSPENDED :
                System.out.println("ReceiverNotificationBean.handleEvent(): Task is
SUSPENDED");
                break;
        }
        checkTaskStatus(task.getTaskStatus());
    }
    catch (Exception e) {
        System.out.println("ReceiverNotificationBean.handleEvent(): " + e.getMessage());
    }
   System.out.println("----------------------------------------------------------------");
}

public void checkTaskStatus (TaskStatus status) {
    System.out.println("ReceiverNotificationBean.checkTaskStatus(): Task Name      : " +
status.getName());
    System.out.println("ReceiverNotificationBean.checkTaskStatus(): Task ID        : " +
status.getTaskId());
    System.out.println("ReceiverNotificationBean.checkTaskStatus(): Repeats left   : " +
status.getRepeatsLeft());
    System.out.println("ReceiverNotificationBean.checkTaskStatus(): Next fire time : " +
status.getNextFireTime());
    switch (status.getStatus()) {
        case TaskStatus.CANCELLED:
            System.out.println("ReceiverNotificationBean.checkTaskStatus(): Task Status    :
CANCELLED");
            break;
        case TaskStatus.COMPLETE :
            System.out.println("ReceiverNotificationBean.checkTaskStatus(): Task Status    :
COMPLETED");
            break;
        case TaskStatus.INVALID :
```

```
            System.out.println("ReceiverNotificationBean.checkTaskStatus(): Task Status    :
INVALID");
            break;
        case TaskStatus.RUNNING :
            System.out.println("ReceiverNotificationBean.checkTaskStatus(): Task Status    :
RUNNING");
            break;
        case TaskStatus.SCHEDULED :
            System.out.println("ReceiverNotificationBean.checkTaskStatus(): Task Status    :
SCHEDULED to execute");
            break;
        case TaskStatus.SUSPENDED :
            System.out.println("ReceiverNotificationBean.checkTaskStatus(): Task Status    :
SUSPENDED");
            break;
    }
}
```

6. Save and close ReceiveNotificationBean.java.

Adding a Startup bean

In this section, we will create a Startup bean to register the `BatchProcess` session bean (scheduler task) created earlier. You can follow the steps from the Chapter 13, “Startup beans” on page 375 to create the Startup bean.

1. Create a new EAR, `StartupSchedulerEAR`, and EJB, `StartupSchedulerEJB`, project with the Startup bean: `HelloWorldSchedulerStartup`.
2. Add the Scheduler project to the StartupSchedulerEJB Java build path.
3. Open the HelloWorldSchedulerStartupBean.java file Insert the start() method with the code given below.

Example 14-3 HelloWorldSchedulerStartupBean.java start() method

```
public boolean start() {
    boolean result = false;
    System.out.println("StartupBean.start()");
    try {
        // Check the TestProcessBean deployment descriptor under "Beans" tag to figure out the
JNDI name
        // to use below
        String jndi_name_sched = "java:comp/env/BPEScheduler";
        String jndi_name_task = "java:comp/env/ejb/BatchProcess";
        String jndi_name_notify = "java:comp/env/ejb/ReceiveNotification";
        Context initialContext = new InitialContext();
        // lookup the scheduler instance to be used
        System.out.println("StartupBean.start(): Getting the Scheduler");
        Scheduler schedulerHome = (Scheduler) initialContext.lookup(jndi_name_sched);
```

```
        // find the session bean to be called when the task executes
        System.out.println("StartupBean.start(): Getting the TaskHandler");
        Object taskHome = initialContext.lookup(jndi_name_task);
        TaskHandlerHome taskHandlerHome = (TaskHandlerHome)
javax.rmi.PortableRemoteObject.narrow( taskHome, TaskHandlerHome.class);
        // lookup session bean that should receive notification events from scehduler
        System.out.println("StartupBean.start(): Getting the ReceiverNotication");
        Object receiverNotificationHome = initialContext.lookup(jndi_name_notify);
        NotificationSinkHome notifyHome = (NotificationSinkHome
)javax.rmi.PortableRemoteObject.narrow(receiverNotificationHome,NotificationSinkHome.class);
        // Create our Schedule Task Info for our BatchProcessTask task handler
        System.out.println("StartUpBean.start(): Creating the task");
        BeanTaskInfo taskInfo = (BeanTaskInfo)
schedulerHome.createTaskInfo(BeanTaskInfo.class);
        // create a date object which represents 30 seconds from now
        // the task will start 30 seconds from now, it will run 5 times,
        // and it will repeat 1 minute.
        java.util.Date startDate = new java.util.Date(System.currentTimeMillis() + 30000);
        taskInfo.setTaskHandler(taskHandlerHome);
        taskInfo.setNotificationSink(notifyHome, TaskNotificationInfo.ALL_EVENTS);
        taskInfo.setName("BatchProcessTask");
        taskInfo.setStartTime(startDate);
        taskInfo.setRepeatInterval("1minutes");
        taskInfo.setNumberOfRepeats(5);
        // submit the Task to scheduler to be stored in persistant store
        TaskStatus status = schedulerHome.create(taskInfo);
        System.out.println("StartupBean.start(): Task submitted");
        System.out.println("StartupBean.start(): Task ID is: " + status.getTaskId());
            System.out.println( "StartupBean.start(): The name of the Task is: " +
status.getName());
        System.out.println( "StartupBean.start(): Task status is: " + status.getStatus());
        result = true;
    } catch (Exception e) {
        System.out.println("StartupBean.start(): " + e.toString());
    }
    return result;
}
```

4. Add the following import statements at the top of the file:

   ```
   java.util.*;
   javax.naming.*;
   com.ibm.websphere.scheduler.*;
   ```

5. We need to add three references to the Startup bean, as you see them in the source code. Open the EJB Deployment Descriptor and switch to the References tab.

   a. Select the **HelloWorldSchedulerStartup** and click **Add** at the bottom.

b. Add the following EJB reference:

   Enterprise bean in a different EAR: click **SchedulerEAR** → **Scheduler** → **BatchProcess**

c. Add the following EJB reference:

   Enterprise bean in a different EAR: click **SchedulerEAR** → **Scheduler** → **ReceiveNotification**

d. Add the following Resource reference:

- Name: `BPEScheduler`
- Type: `com.ibm.websphere.scheduler.Scheduler`
- Authentication: `Container`
- Sharing scope: `Shareable`
- JNDI: `BPEScheduler`

6. Save and close the file.

Modifying the Startup Bean for programmatic management of scheduled tasks

A scheduled task has an associated task ID. When a scheduled task is created, a TaskStatus object is returned to the caller. Using the task ID of the scheduled tasks, it is possible to perform various management tasks on the task. The scheduler API defines several additional methods that pertain to the management of tasks. In this section we will illustrate how to Cancel and Purge a task that has been scheduled.

1. Open the HelloWorldSchedulerStartupBean.java file. Insert the stop() method with the code given below.

Example 14-4 HelloWorldSchedulerStartupBean.java stop() method

```
public void stop() {
    System.out.println("StartupBean.stop()");
    try {
        // Check the TestProcessBean deployment descriptor under "Beans" tag to figure out the
JNDI name
        // to use below
        String jndi_name_sched = "java:comp/env/BPEScheduler";
        Context initialContext = new InitialContext();
        // lookup the scheduler instance to be used
        System.out.println("StartupBean.stop(): Getting the Scheduler");
        Scheduler schedulerHome = (Scheduler) initialContext.lookup(jndi_name_sched);
        // find all TaskStatus objects with a specified name that were
        // created in start() method above.  Keep in mind we set the
        // name of the task so that we can look it up later on.  This
        // is our only handle for finding a task.  If there are a lot
        // of tasks that you plan on creating, create a single place
```

```
        // where you can store all these tasks and then refer to them
        // instead of hard coding them in the code.
        System.out.println("StartupBean.stop(): Finding all Tasks : \"BatchProcessTask\"");
        Iterator list = schedulerHome.findTaskStatusByName("BatchProcessTask");
        while (list.hasNext()) {
            TaskStatus status = (TaskStatus)list.next();
            System.out.println("StartupBean.stop(): Task Found : ID is " + status.getTaskId());
            System.out.println("StartupBean.stop(): Task Found : Name is: " +
status.getName());
            System.out.println("StartupBean.stop(): Task Found : Status is: " +
status.getStatus());
            // Cancelling task programmetically
            System.out.println("StartupBean.stop(): Cancelling task id: " +
status.getTaskId());
            schedulerHome.cancel(status.getTaskId(), true);
        }
    } catch (Exception e) {
        System.out.println("StartupBean.stop(): "  + e.toString());
    }
}
```

2. Save and close the file.

14.4 Unit test

The Scheduler service must be configured in WebSphere Studio Application Developer Integration Edition test environment. In WebSphere Studio Application Developer Integration Edition, there is a limited set of administration tasks for the test server available through the GUI. In particular, there is no Scheduler administration using the server editor tool in WebSphere Studio Application Developer Integration Edition. You have to have the test environment server running and the Administrative Console enabled for the server. Once the server is running, use a Web browser to access the Administrative Console where you can configure the Scheduler service the same way as for the runtime application server.

1. Create a new Integration Test server and configuration. Add all the projects to the server: HelloWorldProcessEAR, SchedulerEAR, StartupSchedulerEAR.
2. Switch to the Configuration tab and change the Classloader policy to **SINGLE**.
3. Switch to the Applications tab and set the Start weight for each application to ensure the proper startup sequence
   - HelloWorldProcessEAR: 10
   - SchedulerEAR: 20

– StartupSchedulerEAR: 30

4. Scheduler can be configured in WebSphere Test Environment as illustrated in 14.6, “Configuration” on page 411. The figure below indicates that a scheduler must be defined in the WebSphere Test Environment for testing scheduler task beans.

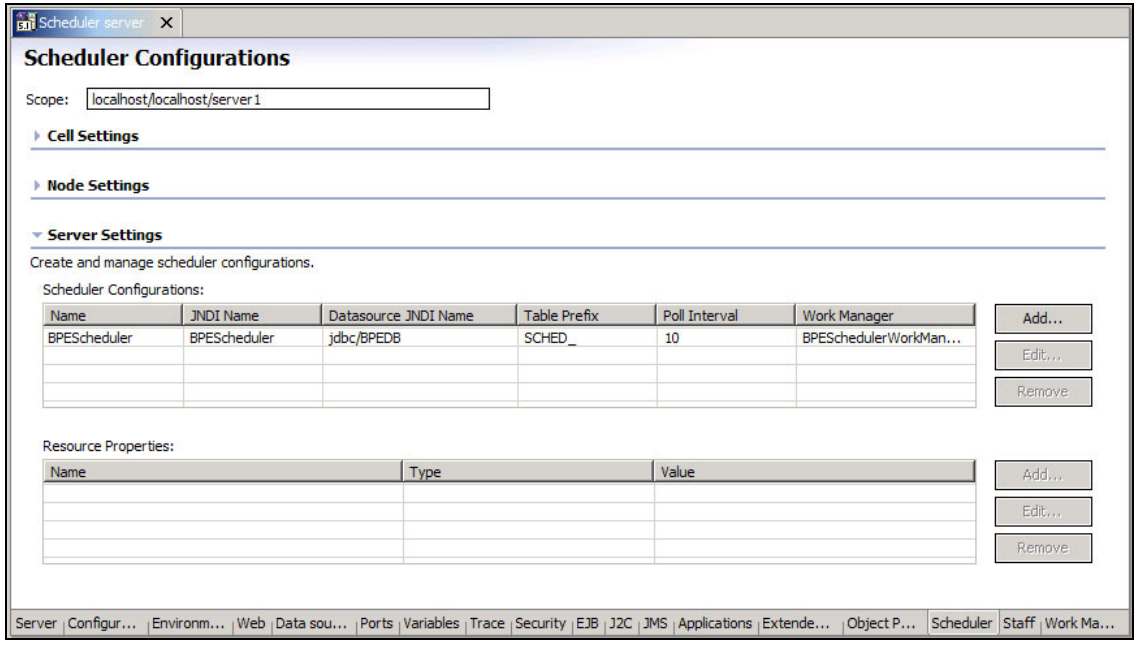


Figure 14-3 WebSphere Test Environment Scheduler configuration

5. Save and close the configuration.
6. You can start the server to test the application.

   During startup, you should see the following messages on the console.

Example 14-5 System out during startup

```
SystemOut     O StartupBean.start()
SystemOut     O StartupBean.start(): Getting the Scheduler
SystemOut     O StartupBean.start(): Getting the TaskHandler
SystemOut     O StartupBean.start(): Getting the ReceiverNotication
SystemOut     O StartUpBean.start(): Creating the task
SystemOut     O ----------------------------------------------------------------
SystemOut     O ReceiverNotificationBean.handleEvent(): Event received at Thu Jun 10 23:04:35
EDT 2004
SystemOut     O ReceiverNotificationBean.handleEvent(): Task was SCHEDULED
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Task Name      : BatchProcessTask
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Task ID        : 1
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Repeats left   : 5
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Next fire time : Thu Jun 10
23:05:04 EDT 2004
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Task Status    : SCHEDULED to
execute
SystemOut     O ----------------------------------------------------------------
SystemOut     O StartupBean.start(): Task submitted
SystemOut     O StartupBean.start(): Task ID is: 1
SystemOut     O StartupBean.start(): The name of the Task is: BatchProcessTask
SystemOut     O StartupBean.start(): Task status is: 1
```

A minute after the application server started, you shuld see the following messages on the console.

Example 14-6 System out after startup

```
WsServer      A WSVR0001I: Server server1 open for e-business
SystemOut     O ----------------------------------------------------------------
SystemOut     O ReceiverNotificationBean.handleEvent(): Event received at Thu Jun 10 23:05:04
EDT 2004
SystemOut     O ReceiverNotificationBean.handleEvent(): Task is FIRING
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Task Name      : BatchProcessTask
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Task ID        : 1
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Repeats left   : 5
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Next fire time : Thu Jun 10
23:05:04 EDT 2004
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Task Status    : SCHEDULED to
execute
SystemOut     O ----------------------------------------------------------------
SystemOut     O ************************************************
SystemOut     O Scheduler bean: starting...
SystemOut     O Scheduler bean: Executing SayHello() method
SystemOut     O Hello World
SystemOut     O Scheduler bean: Finished executing SayHello() method
SystemOut     O ************************************************
SystemOut     O ----------------------------------------------------------------
```

```
SystemOut     O ReceiverNotificationBean.handleEvent(): Event received at Thu Jun 10 23:05:07
EDT 2004
SystemOut     O ReceiverNotificationBean.handleEvent(): Task FIRED
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Task Name      : BatchProcessTask
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Task ID        : 1
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Repeats left   : 5
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Next fire time : Thu Jun 10
23:05:04 EDT 2004
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Task Status    : SCHEDULED to
execute
SystemOut     O ----------------------------------------------------------------
SystemOut     O ----------------------------------------------------------------
SystemOut     O ReceiverNotificationBean.handleEvent(): Event received at Thu Jun 10 23:05:07
EDT 2004
SystemOut     O ReceiverNotificationBean.handleEvent(): Task was SCHEDULED
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Task Name      : BatchProcessTask
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Task ID        : 1
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Repeats left   : 5
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Next fire time : Thu Jun 10
23:05:04 EDT 2004
SystemOut     O ReceiverNotificationBean.checkTaskStatus(): Task Status    : SCHEDULED to
execute
SystemOut     O ----------------------------------------------------------------
```

You can see five execution of the HelloWorld process, then the Task status becomes: `INVALID`.

7. Stop the server.

14.5 Assembly

The following section discusses the application assembly related actions for the Scheduler service.

In the application code, local name references are used for the Scheduler service so before you can use the Scheduler Task beans in WebSphere Business Integration Server Foundation runtime environment, you have to tell your application where they are in the JNDI name space. There are two ways to accomplish this task:

1. Use WebSphere Studio Application Developer Integration Edition and make appropriate changes to the Deployment Descriptors prior to importing the Enterprise Application Archive.
2. Use the Application Server Toolkit to make the changes.

14.6 Configuration

Detailed information regarding Scheduler configuration and administrative tasks can be found in the WebSphere Business Integration Server Foundation InfoCenter. The link to the InfoCenter is provided below.

```
http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/
info/ee/scheduler/tasks/tsch_managescheduler.html
```

Primarily there are three high-level tasks that must be performed in order to configure scheduler:

- Create the database for the scheduler.
- Configure the scheduler.
- Enable the scheduler service.

14.7 More information

This section aims at covering a few common discussion topics that are usually associated with any J2EE application. More importantly, the focus of this section will be Configuration, Problem Determination, Monitoring, Security Considerations.

14.7.1 Problem determination

If you encounter problems using the Scheduler service and you suspect that the source of the problem is somewhere within the service, then you can also use the following trace string to monitor what is happening in the execution time:

```
com.ibm.ws.scheduler.*=all=enabled
```

14.7.2 Security considerations

If you use security on WebSphere Application Server V5.0 Enterprise, the default identity that the Scheduler service uses with the task action session bean is system. This means if you do not specify any deployment-related security settings on the bean, the identity of the bean will be the same as the one used for starting WebSphere.

If the task action session bean (TaskHandler) uses some secured objects that require a different caller identity, then the security identity must be enabled on the Startup Bean and The Run-as on the bean's process must be configured to use the correct identity.

14.7.3 Clustering

Scheduler service can be clustered for high-availability. No steps are required to cluster a scheduler. If a scheduler resource is created a the cell level, the scheduler is automatically clustered across the cell.

When multiple scheduler daemons are configured to the same table (as is the case in a clustered environment), any of the daemons can find a task and set the timer in its Java Virtual Machine (JVM). The task is executed in the virtual machine where the timer first fires.

It is important to realize that if a scheduler daemon fails for some reason, that doesn't stop the scheduler service. If a daemon fails while processing a task, the task may be fired again hence implementing “at least once” behavior on scheduled task.

Consequently, each scheduler daemon configured in the cell try and execute the same scheduled task. However, each daemon verifies whether another daemon has previously fired the task. However if a daemon fails while firing off the task (and before updating the database for task status), the same scheduled task may be picked up by another scheduler daemon. Thus, scheduler service ensures “run at least once” behavior. It is not possible to implement, run “once and only once” behavior.

Scheduler daemon internals are discussed in more details in the WebSphere Business Integration Server Foundation InfoCenter:

http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/info/ee/scheduler/concepts/csch_schedulerdaemon.html

14.7.4 Performance considerations

The polling interval for a Scheduler service can have an impact on performance. The longer the interval, the lower the performance impact, and the lower the time accuracy with which a scheduled task is executed.

Both custom and WebSphere based calendars are treated equally by the Scheduler. There is a slight performance advantage when using the IBM-implemented calendars (CRON and Arithmetic).

When implementing scheduler base tasks, it should be kept in mind that scheduler is not meant for split-second type of accuracy. The reason for that is scheduler daemon has to poll the database at regular interval. The polling sequence itself introduces a margin of error on the time accuracy with which a task can be fired. For example, if a polling interval of 30 seconds is used, the margin of error is about one minute.

Scheduler can benefit organizations by automating administrative tasks and/or manually starting tasks. Typical tasks that can be implemented using Scheduler service include periodic back-up operations or clean-ups, auditing tasks and report generation. Generally, the tasks do not require millisecond accuracy. Scheduler can also be shared by multiple applications, further simplifying overall administration.

14.7.5 Future direction

The new EJB V2.1 specification includes the Timer Service component. This service provides some of the features of WebSphere Business Integration Server Foundation scheduler service. Future versions of WebSphere Business Integration Server Foundation may include a Scheduler service that builds on top of the standard timer service.

15

Chapter 15.

Asynchronous beans

The Asynchronous bean framework provided by WebSphere Business Integration Server Foundation V5.1 provides J2EE components with access to managed threads and allows J2EE contexts to be propagated to a separate thread. By using Asynchronous beans, your J2EE components will be able to run code on simultaneous threads and asynchronously.

An Asynchronous bean is a Java object or Enterprise Java Bean (EJB) that can be executed asynchronously by a J2EE application using the J2EE context of the bean's creator. The following J2EE contexts can be inherited:

- Internationalization context
- WorkArea context
- Application profile
- Security context

The ability to transfer a J2EE context to the newly spawned thread is very important because it provides full access to the full J2EE programming model and API.

Asynchronous beans enable the construction of stateful, "active" and event-driven J2EE applications. Some of the benefits of using Asynchronous beans include:

- **The ability to partition tasks so they can run in parallel**

  You can execute a complex database task using multiple threads. An example is a calculation over a large set of rows where the set can be partitioned and each partition can execute in parallel. The EJB can block until all threads have finished before aggregating the results.

- **Integration of non-JMS messaging middleware**

  Applications can integrate a third-party messaging solution that does not support JMS but has a Java API.

- **Ability to dynamically listen for JMS queues and topics**

  Threads can be used to block and receive messages regarding queues or topics that were unknown at deployment.

- **The use of background processing for performance**

  An application that performs persistent message logging can use a background task to write batched insert operations to a database in a single transaction. Foreground tasks can store the log message in a synchronized data structure and continue processing without waiting for the message to be persisted.

  Instead of each log operation resulting in a “begin, insert msg, commit” operation that can slow the application down significantly, a background thread can batch the insert operations together (for example `begin, insert msg1,msg2,msg3,msg4, commit`). This lowers the impact of this logging on both the application server and the database.

15.1 Prerequisites

Here are some useful resources that will help you get started with Asynchronous beans:

- WebSphere Business Integration Server Foundation V5.1 InfoCenter at http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp, then navigate to WebSphere Business Integration Server Foundation and choose **Asynchronous beans** under Highlights and Features.
- WebSphere Studio Application Developer Integration Edition V5.1 Help at **WebSphere Business Integration Server Foundation** → **Highlights and Features** → **Asynchronous beans**.
- IBM Redbook: *WebSphere Application Server Enterprise V5 and Programming Model Extensions*, SG24-6932, Chapter 7, “Asynchronous beans.”

15.2 Design

This section describes some design considerations to take into account when developing Asynchronous beans.

Asynchronous beans: simple Java objects or EJBs?

An Asynchronous bean can be either a Java object or an EJB. Applications that use the servlet-only approach may use Java objects whereas applications using EJBs may use stateless session beans or entity beans.

There are several differences in behavior between the two choices; these are summarized in Table 7-1.

Table 15-1 Java or EJB Asynchronous beans comparision

	Java Beans	**EJB**
Transactions	If created by a servlet then java:comp/UserTransaction is available. If created by an EJB then only TX_NOT_SUPPORTED is allowed and a ‘buddy’ EJB must be used for full global transaction support.	The support is specified by the descriptor for the EJB and the J2EE specification.

	Java Beans	EJB
Security	The credentials on the thread that created the Asynchronous bean are used when the bean is invoked.	The credentials on the thread that created the Asynchronous bean are used. However, the descriptor for the bean can override this with the Run-as role attribute.
Application Profile	The profiles active for the creating component are used.	The profiles active for the creating component are used but they may be augmented by specifying additional ones on the target EJB method.
Java: comp scope	The Java:comp of the component that created the Asynchronous bean are always available to the Asynchronous bean.	The java:comp of the creating component is ignored. The java:comp of the async EJB is always used.

There is not much difference between the two performance-wise because the performance is roughly equivalent to a local method call in both cases.

The types of Asynchronous beans

The three types of Asynchronous beans are as follows:

- Work

  A work object implements the com.ibm.websphere.asynchbeans.Work interface and runs parallel to its caller using the WorkManager.startWork() method. Applications implement work objects to run code blocks asynchronously.

- AlarmListener

  An alarm listener is an object that implements the com.ibm.websphere.asynchbeans.AlarmListener interface and is called when a high-speed transient alarm expires. Depending on whether an alarm is a once only or repeating event, actions can be taken to improve performance. If the alarm is a once only event or the last alarm, the alarm should be cancelled, otherwise if the alarm is repeating, the Alarm.reset() method should be called to schedule the alarm to fire again later.

- EventListener

  Event listeners are asynchronous by nature. An application can create an event listener and subscribe it to monitor the occurrence of a certain event. When the event occurs, the listener will be notified and will handle the event.

  Listeners can implement any interface, however, the application that originates the event needs to know which method corresponds to the event on the listener's interface. The event originator will do this by acquiring a "proxy" from the EventSource. Calling a method on the proxy will cause the same method to be invoked on all listeners that are registered and implement the requested interface.

  The listener's method will be executed in its own thread but it will run under the J2EE context of the component that registered the listener itself. It will not use the J2EE context of the application that is firing the event.

15.3 Sample scenario

In this section, we will design an Asynchronous J2EE application that uses the Work, AlarmListener and EventListener Asynchronous beans. Also, this application uses the Startup Bean service. For more information about Startup Beans, refer to Chapter 13, "Startup beans" on page 375.

The purpose of our application is to use a Startup bean to start Asynchrnous work to initialize the cache so the server does not appear to be 'hanging' during application startup. After we have loaded the application cache using a CMP bean, we also need a way to refresh our cache to get any updated data. To do this we will create an Alarm in the Startup bean to go off every 30 seconds. If the user then refreshes the JSP, they will receive the latest, up-to-date data held in cache.

Migration

The Asynchronous beans implementation is the same as it was in WebSphere Application Server Enterprise V5 there are no specific migration tasks.

Development

In this chapter we are not going to develop the whole sample application using Asynchronous beans. The application is available as an additional material, downloadable from the IBM ITSO Web site, together with the PDF version of this book.

Instead, we show how to import the fully written application into WebSphere Studio Application Developer Integration Edition, then the development section points out the Asynchronous beans development related key points.

Test environment

The test environment supports the unit test for the application. This section we provide detailed information about how to set up the test environment for the Asynchronous Beans sample application.

Assembly

There are a few application settings you have to be aware of when developing or assembling an application using Asynchronous Beans.

Configuration and deployment

The configuration part shows what the necessary configuration steps for the WebSphere Business Integration Server Foundation runtime environment before you can deploy and run your application.

The deployment of an application using Asynchronous Beans is not different than any other enterprise application deployment.

15.3.1 Understanding the sample application

The sample application is a standard J2EE application with one CMP Entity Bean (Department), one singleton Java object (EntityBeanCacheSingleton), one startup session bean (AppBootstrapBean) and one JSP (ListDepartment.jsp).

The application is initialized when the application server starts. The WebSphere runtime invokes the start() method of the AppBootstrapBean, which in turn creates an instance of a singleton class EntityBeanCacheSingleton. The singleton class will call the findAll() method in the home interface of the entity bean Department and will store the result in the cache. So, when the WebSphere server is up and running, the application is also initialized with CMP cache in the singleton. Whenever the user invokes ListDepartment.jsp, it will get data from the cache held by the singleton class.

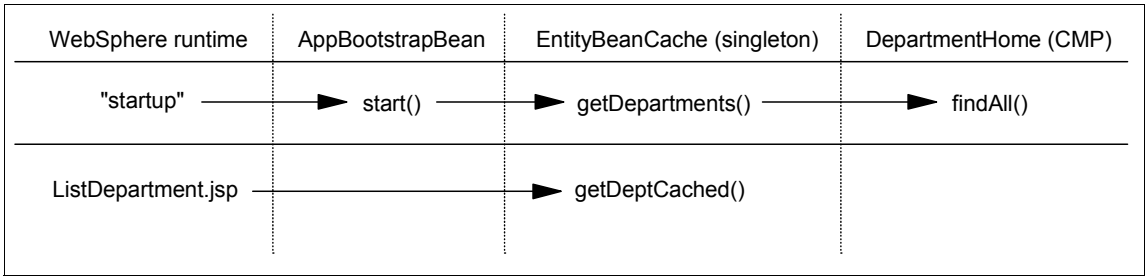


Figure 15-1 Interaction diagram for the sample

15.4 Development

This section tells you how to develop the sample application using Asynchronous Beans.

The AppBootstrapBean Startup bean

In this sample application, we want to offload the task of looking up CMP Entity Beans from the AppBootstrapBean on the main thread of execution.

- We create a Work object (InitializeCacheWork). it will invoke the EntityBeanCacheSingleton, which will in turn do the heavy-duty work of finding all instances of the entity beans and store the cache in memory.
- In order to simulate the data change behavior in our sample application, we will be implementing an AlarmListener UpdateDbAlarmListener, which will be called every 30 seconds when an associated alarm fires. The AlarmListener will use the Department CMP to insert a record into the database and then fire the updateEntityCache event in the UpdateCacheEventListener class.

We use one Startup Bean in this sample to initialize the singleton class and thus the CMP cache. The code in Example 15-1 is the start method of the bean.

Example 15-1 Startup Bean, start() method

```
public boolean start() {
   try {
      //get the Work Manager from the JNDI
      InitialContext ic = new InitialContext();
      WorkManager wm = (WorkManager)
ic.lookup("java:comp/env/wm/WorkManager");
      //Start an Asynchronous work to initialize the cache
      InitializeCacheWork pw = new InitializeCacheWork();
      wm.startWork(pw);
      //get the AsynchScope
      AsynchScope as = wm.findAsynchScope("ItsoScope");
      if (as == null) {
         as = wm.createAsynchScope("ItsoScope");
      }
      //get the AlarmManager
      AlarmManager am = as.getAlarmManager();
      UpdateDbAlarmListener updateListener = new UpdateDbAlarmListener();
      //create an Alarm to go off every 30 seconds
      Alarm a1 = am.create(updateListener, this, 30000);
   } catch (Exception e) {
      //...
      return false;
   }
   return true;
}
```

The EntityBeanCacheSingleton object

When the instance of this class is created, it will call the findAll() method in the home interface of the entity bean Department and store the results in a static data structure.

We will also implement asynchronous notification to refresh our CMP cache in case the underlying data changes. It works in the following way:

- First, we define an UpdateCacheEventListener interface with an method/event updateEntityCache.
- The EntityBeanCacheSingleton object needs to be notified when such event fires, so it implements the EventListener interface UpdateCacheEventListener.
- To facilitate intra-application notification, WebSphere provides a special type of EventSource, which is included in each enterprise application. This EventSource can be found using JNDI lookup in any servlet or EJB code in the application.

Whenever the data in the database changes, updateEntityCache event will be fired and the Singleton object will be called to update its copy of the cache of CMP data.

Example 15-2 EntityBeanCacheSingleton.java

```
//...
import com.ibm.websphere.asynchbeans.*;

public class EntityBeanCacheSingleton implements UpdateCacheEventListener{

   static private EntityBeanCacheSingleton _instance = null;
   private Vector deptCache = null;

   // constructor method for EntityBeanCacheSingleton
   protected EntityBeanCacheSingleton() {
      try {
      deptCache = this.getDepartments();
      //register this as a listener to asynchronous event
      InitialContext ic = new InitialContext();
      EventSource appES = (EventSource)
ic.lookup(EventSource.APPLICATION_NOTIFICATION_EVENT_SOURCE);
       appES.addListener(this);
      } catch (Exception e) {
         //...
      }
   }

   // creating a singleton instance of the EntityBeanCacheSingleton
   static public EntityBeanCacheSingleton instance() {
      if (null == _instance) {
         _instance = new EntityBeanCacheSingleton();
      }
      return _instance;
   }

   private Vector getDepartments() {
      Vector v = new Vector();
      try {
         // Get a local reference
         InitialContext ic = new InitialContext();
         DepartmentLocalHome cbHome =
            (DepartmentLocalHome) ic.lookup("java:comp/env/ejb/Department");
         Collection c = cbHome.findAll();
         Iterator i = c.iterator();
         while (i.hasNext()) {
            DepartmentLocal dl = (DepartmentLocal) i.next();
            DepartmentData dd = new DepartmentData();
            String dno = ((DepartmentKey) dl.getPrimaryKey()).getDeptno();
```

```
            dd.setDeptNo(dno);
            dd.setDeptName(dl.getDeptname());
            dd.setMgrNo(dl.getMgrno());
            v.add(dd);
         }
      } catch (Exception e) {
         //...
      }
      return v;
   }

   // this method updates the cache
   public void updateEntityCache() {
      setDeptCache(getDepartments());
   }

   public Vector getDeptCache() {
      return deptCache;
   }

   public void setDeptCache(Vector deptCache) {
      this.deptCache = deptCache;
   }
}
```

The ListDepartment JSP

The ListDepartment.jsp will calls the EntityBeanCacheSingleton, and displays data from the cache. It is a very simple JSP that iterates through a vector of data and writes them out to the response output. The JSP only serves testing purposes, it does not have any Asynchronous Beans specific code implemented in it. For more information check the source code of the JSP.

The Department CMP Bean

Department CMP Bean is a standard EJB 2.0 entity bean with container-managed persistence. In this sample, we created only local home and remote interfaces for it, since it is not accessed remotely by other clients from outside the container. We also created a custom finder findAll() in its home interface (DepartmentLocalHome.java) using the Enterprise Java Bean Query Language (EJB QL). For more information about EJB QL, refer to the IBM developerWorks article: *Introduction to container-managed persistence and relationships,* Part 3 at:

http://www-106.ibm.com/developerworks/edu/ws-dw-wscomp3-i.html

15.5 Test environment

This section provides information on how to import the Asynchronous Beans sample application to WebSphere Studio Application Developer Integration Edition and how to test it in the test environment.

Setting up the sample application project

1. Launch WebSphere Studio Application Developer Integration Edition with a new workspace for the Asynchronous Beans sample application. Make sure you enable the server targeting support.
2. Import the **ItsoAsyncBeans.ear** application, make sure you select Integration Server V5.1 for target server.
3. Create the database mapping for the EJBs. On the J2EE perspective, J2EE Hierarchy view, right-click **EJB Modules** → **ItsoAsynchBeansEJB** and select **Generate** → **EJB to RDB Mapping**.
   a. Select **Create a new back-end folder** and click **Next**.
   b. Select **Top Down** and click **Next**.
   c. For the target database, select your database, for testing purposes we used **Cloudscape, V5.1**, Database name: `ASYNCB`, Schema name: `EJB`. Click **Finish**.

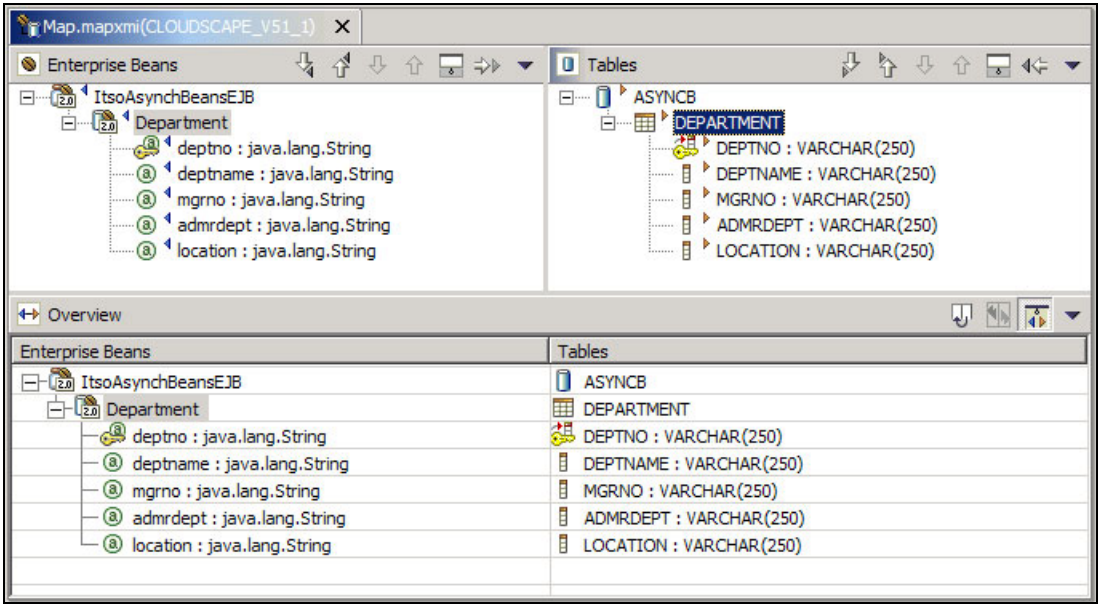


Figure 15-2 The generated Map.mapmxi file for Cloudscape

d. Close the generated Map.mapmxi file.

4. Generate the deployed code for the EJBs. Right-click **EJB Modules** → **ItsoAsynchBeansEJB** and select **Generate** → **Deploy and RMIC Code**.

   Click **Select All**, to select all the EJBs, then click **Finish**.

5. Open to the Server perspective and create a new **WebSphere version 5.1** → **Integration Test Environment** test server, with the name: `AsyncBean server`.

   a. Add the ItsoAsyncBeans application to the server.

   b. Use the Create tables and data sources for the server to generate the necessary database components for the test environment.

   c. You can open the server configuration and check the sample WorkManager configured for the server by default, the sample application is going to use this WorkManager. Switch to the Work Managers tab on the server configuration and check if the DefaultWorkManager item exist under the Node settings.

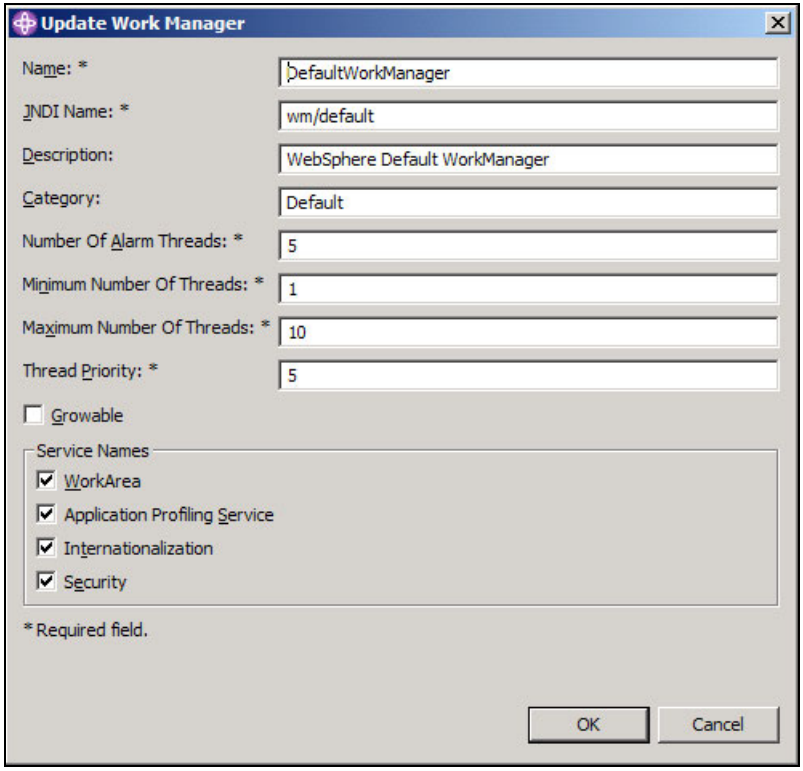


Figure 15-3 Work Manager settings

If you want to create a new Work Manager for your application, this is the place where you can define one.

6. Open the Data perspective, then create a new Database connection, according to the following figure.

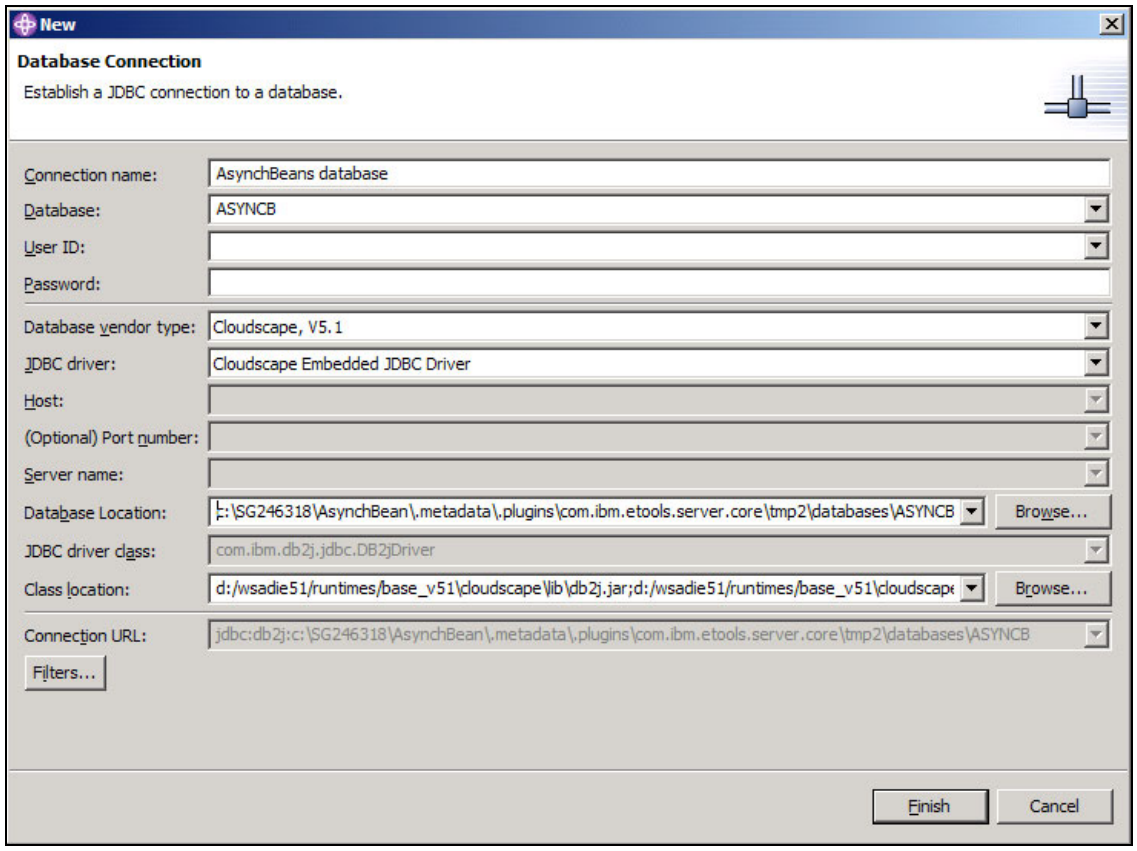


Figure 15-4 New database connection definition

Make sure that the Database location is correct, use **Browse** to find the database under your workspace directory. You may find the database under a different tmp directory, make sure you select the latest tmp (with the highest number) in the directory.

7. Navigate to the **ItsoAsyncBeansEJB** → **META-INF**, right-click the **populate.dll** file, then select **Run on Database server ...**

   a. A new wizard appears; make sure all statements are selected, then click **Next**.

   b. Select **Automatically commit** changes, then click **Next**.

c. Select **Use existing connection**, select the **AsynchBeans database** connection, then click **Finish**.

> **Note:** You have to run the populate.ddl script every time before you start your test server, to start with a clean database. Otherwise you will get database errors, when the update tries to create a new item in the database (to simulate data changes) that already exist.

8. Under the DB Servers view, navigate to **AsyncBeans database** → **ASYNCB** → **EJB** → **Tables**, right-click the **EJB.DEPARTMENT**, then select **Sample Contents**. The inserted records should show under the DB Output view on the Results panel.
9. Make sure you disconnect from the ASYNCB database after you have finished the work or before you start the application server.

Testing the sample application

This section will guide you htrough, how to test the sample application in the WebSphere Test Environment.

1. Start the test application server and monitor the console. Once the server started, you will see the following messages on the console.

Example 15-3 Console output WebSphere Studio

```
SystemOut     O $$$$$$$$$$$$ APPBOOTSTAPBEAN HAS BEEN CALLED
SystemOut     O $$$$$$$$$$$$ STARTING ASYNCHRONOUS WORK TO INITIALIZE CACHE
ApplicationMg A WSVR0221I: Application started: ItsoAsynchBeans
StaffServiceI I STFF0032I: The Staff Service started successfully.
WSRdbDataSour I DSRA8203I: Database product name : DBMS:db2j
WSRdbDataSour I DSRA8204I: Database product version : 5.1.45
WSRdbDataSour I DSRA8205I: JDBC driver name  : Cloudscape Embedded JDBC Driver
WSRdbDataSour I DSRA8206I: JDBC driver version  : 5.1
HttpTransport A SRVE0171I: Transport http is listening on port 9,080.
SystemOut     O $$$$$$$$$$$$ THE CMP CACHE INITIALIZED
SystemOut     O
HttpTransport A SRVE0171I: Transport https is listening on port 9,443.
SchedulerServ I SCHD0031I: The Scheduler Service is starting.
SchedulerServ I SCHD0032I: The Scheduler Instance BPEScheduler is starting.
SchedulerDaem I SCHD0038I: The Scheduler Daemon for instance BPEScheduler has
started.
SchedulerServ I SCHD0033I: The Scheduler Instance BPEScheduler has started.
SchedulerServ I SCHD0001I: The Scheduler Service has started.
RMIConnectorC A ADMC0026I: RMI Connector available at port 2809
WsServer      A WSVR0001I: Server server1 open for e-business
```

The messages indicate that the AppBootstrapBean has been started and the asynchronous work has been initialized.

2. After the server start, 30 seconds later you should see the following messages on the console.

Example 15-4 Console output WebSphere Studio

```
SystemOut     O $$$$$$$$$$$$ UPDATEdbALARM FIRED.
SystemOut     O $$$$$$$$$$$$ HANDLE ALARM: INSERTING NEW ROW INTO TABLE, AND
.....
SystemOut     O $$$$$$$$$$$$ FIRING ASYNCHRONOUS UPDATE CACHE EVENT......
SystemOut     O $$$$$$$$$$$$ HANDLE ASYNCHRONOUS EVENT: UPDATING CACHE......
SystemOut     O $$$$$$$$$$$$ THE CMP CACHE UPDATED.
```

The comments on the console tells you that the alarm was fired and the cache was refreshed.

In order to simulate changes in the database, a new record is being created every time the alarm fires.

3. If you open the DepartmentList.jsp on the test server, right-click the **DepartmentList.jsp** and select **Run on server...** You should see something similar to the following figure.

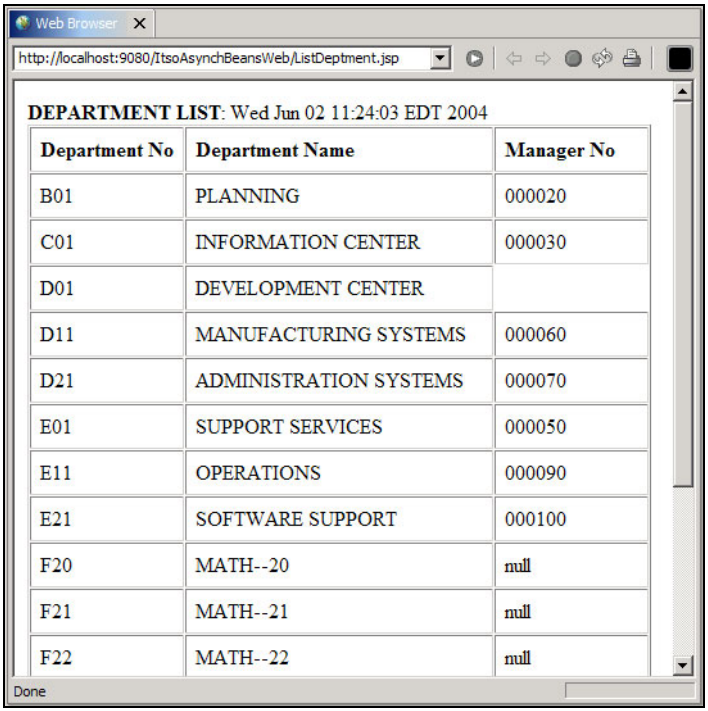


Figure 15-5 Sample data from the ListDepartments.jsp

4. When you are done with the testing, you can stop the test server.

15.6 Assembly

Asynchronous Beans have a few configuration settings for the enterprise applications to be set.

Asynchronous Beans use Work Manager(s) in WebSphere Business Integration Server Foundation to manage the work they are performing. Work Managers are defined in WebSphere Business Integration Server Foundation JNDI namespace. Asynchronous Beans need to lookup and use these Work Managers. Instead of hardcoding the Work Manager name in the code, developers should use resource references in their code.

You can find the same practice in the sample application, where the EJB resource reference: wm/WorkManager refers to the wm/default JNDI name in the Intergration Server V5.1 test server.

If you use any WorkManager from your EJB, Web or client module and you are using resource references, you can use the Application Server Toolkit or WebSphere Studio Application Developer Integration Edition to modify these resource references to reflect the correct names.

15.7 Configuration

The starting point for the configuration is the ItsoAsynchBeans EAR file. We will first create a data source and WorkManager for the application using the Administrative Console. Subsequently, we will install the application in WebSphere Application Server and test it.

Create a WorkManager for the sample application

In this section we are going to create a Work Manager in WebSphere Business Integration Server Foundation for the sample application.

1. Start the application server, then open and login to the Administrative Console.
2. Expand **Resources** → **Work Manager.** Click **New** to create a new item. Use the following details for the entry:
   - Name: `ItsoAsynchBeansWM`
   - JNDI name: `wm/ItsoAsynchBeansWM`
   - Number Of Alarm Threads: `5`
   - Minimum Number Of Threads: `1`
   - Maximum Number Of Threads: `10`
   - Thread Priority: `2`

   Check the **Security** check box in the Service Names field. This will ensure the security context, if any, will be propagated to the Work implementation.

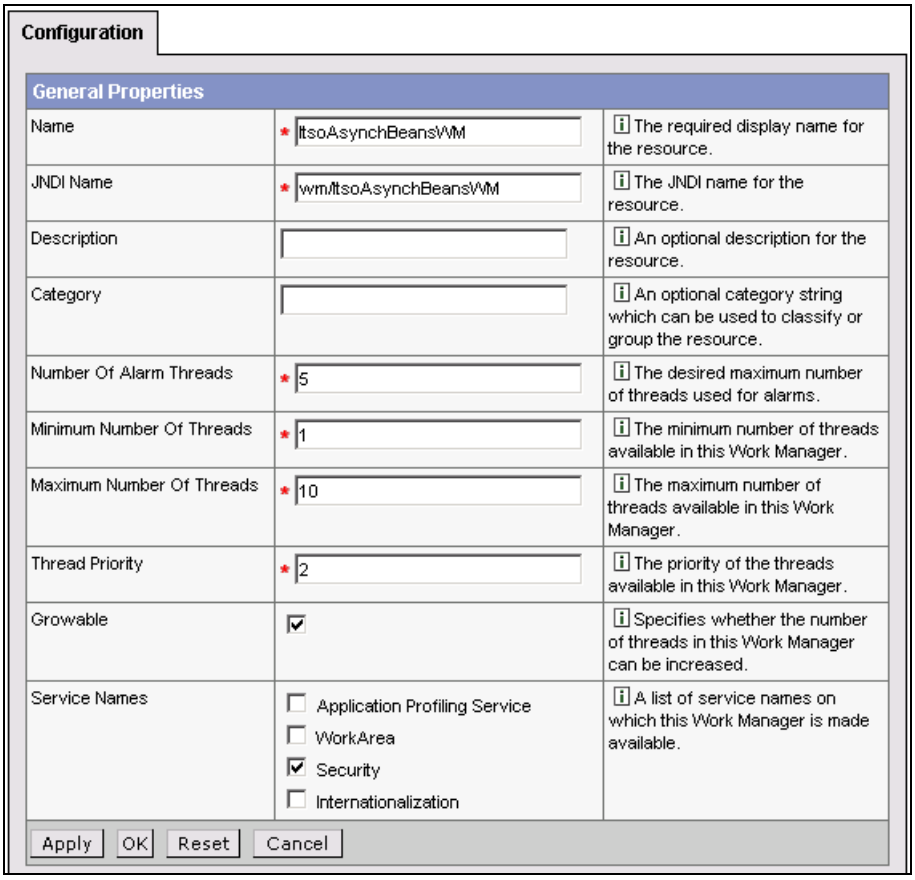


Figure 15-6 Work Manager definition

3. Save the configuration for WebSphere.
4. Restart the application server.

Important: For the Asynchronous Beans sample application, you will have to create a datasource for the sample application. You can use either Cloudscape as we did in the test environment or you can also set up a DB2 database. Either way, follow the steps below.

1. Create the database for your application and populate it with some data if you like.
2. Start WebSphere Application Server Enterprise, launch the Administrative Console, then log in.
3. Create a new data source for the server.
4. Save the configuration for WebSphere.

15.8 Deployment

This section describes the deployment steps for an application that has Asynchronous Beans components.

Important: For the sample application In WebSphere Studio Application Developer Integration Edition generate the RDB to EJB (database) mapping, generate the deployed code for your enterprise application, then export the EAR file of your application.

At this point, you will need the EAR file of your application, then follow the steps below.

1. Start installing the application for WebSphere in the Administrative Console.
2. When you get to the panel, show on the next figure, make sure you set the resource reference right.

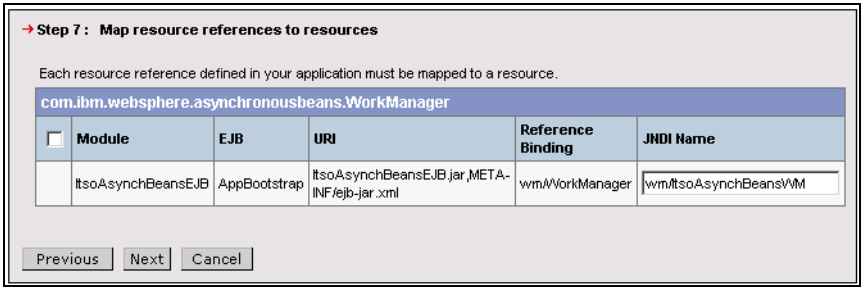


Figure 15-7 Setting the resource reference to the Work Manager's JNDI name

3. At the end of the deployment, save the configuration for WebSphere. At this point, your application should be ready to start and ready for testing.

16

Chapter 16.

Container Managed Persistence over Anything

It is well known that WebSphere Application Server supports Container Managed Persistence (CMP) for Enterprise Java Beans (EJB) over many different databases. In the EJB V2.0 specification, under the Goals section, one of the goals is to provide improved support for the persistence of entity beans. The process of creating and maintaining a Container Managed Persistence entity bean to persist in any relational database is well known.

Whenever there is a need to persist data over a non-relational database, the only option for the developer is to create a Bean Managed Persistence (BMP) Entity Bean and implement persistent logic inside the bean.

In WebSphere Business Integration Server Foundation V5.1, a new feature is called: CMP over Anything. It allows the mapping of CMP EJBs to any back-end application, beyond the traditional relational database. This feature allows the use of a CMP Entity Bean with virtually any type of support, for example:

- Stored Procedure (JDBC and SQL)
- Web services
- JCA adapters (CCI)
- EJB (RMI-IIOP)

16.1 Container Managed Persistence over Anything architecture

Figure 16-1 shows a high-level architecture of Container Managed Persistence over Anything in WebSphere Business Integration Server Foundation.

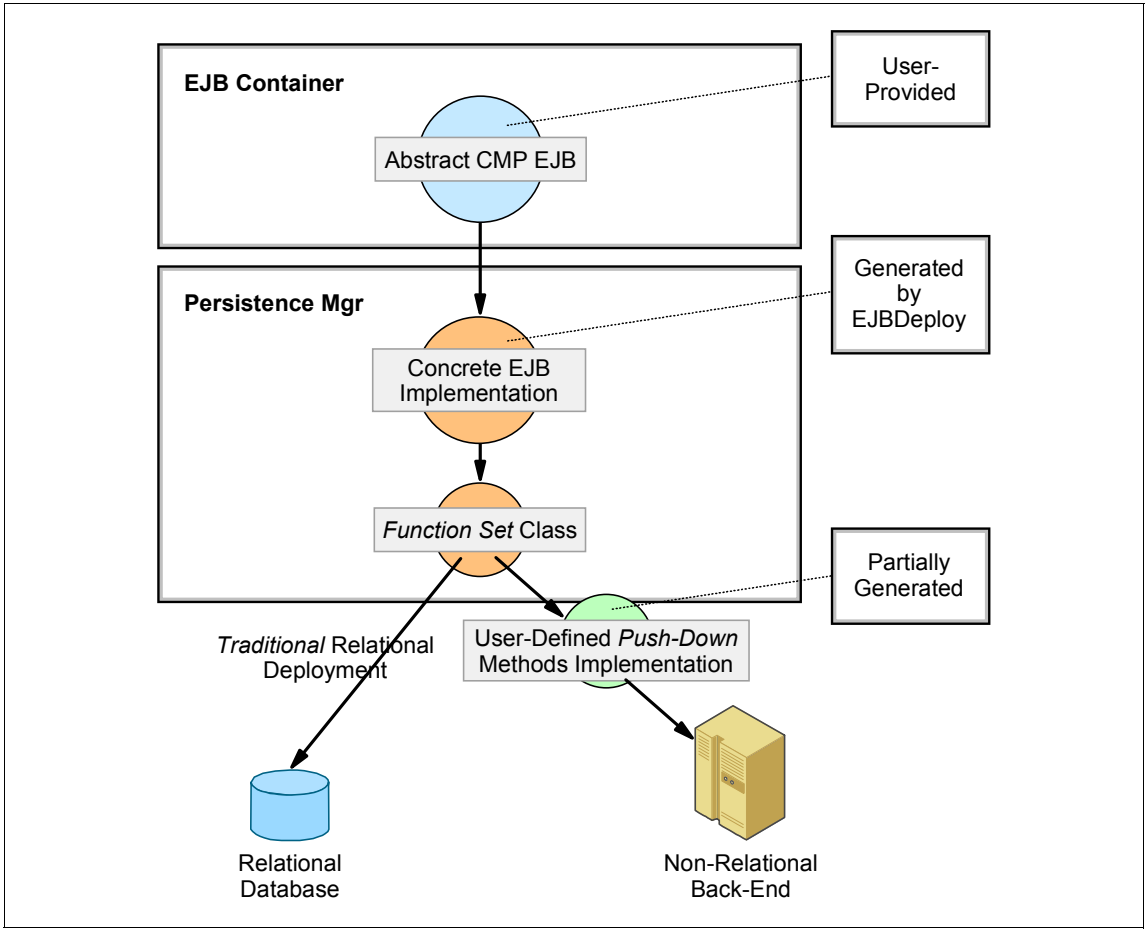


Figure 16-1 CMP over Anything conceptual diagram

The CMP over Anything programing model is similar to the standard CMP Entity Bean programing model. In the bean implementation class, optional push-down methods are defined as abstract methods. The Deployer then implements the UserDefinedPushDownMethodsImpl class. The clients can call a push-down method just like any other EJB method. The client is unaware of the back-end datastore; it is completely transparent to the client. The bean developer is

unaware of back-end data store. Only the Assembler/Deployer has to know about the back-end configuration.

In other words, WebSphere Business Integration Server Foundation introduces support for push-down methods. These methods are considered container-managed business methods. In the CMP framework, we can use the class called UserDefinedPushDownMethodsImpl to place connectivity logic with a non-relational back-end data store. The Java source code for this class is generated by the WebSphere Application Server deployment tools and contains methods corresponding to each CRUD method and each push-down data logic method defined in the WebSphere Business Integration Server Foundation EJB deployment descriptor.

This means that methods can be defined on a CMP bean. The store logic is not implemented by the developer but by the container and deployment tools. This is accomplished by defining the method as abstract, and then marking new WebSphere Business Integration Server Foundation extended deployment descriptor settings for the PushDownMethodElements.

The new command line code generation tool reads these extensions and generates the appropriate entries.

WebSphere Business Integration Server Foundation also includes a generic UserDefinedPushDownMethods interface (not related to the back end) and a UserDefinedPushDownMethodsImpl class that implements the interface which is specific to a given back end. When a method has been marked userDefined, the FunctionSet is delegated to the corresponding method of the UserDefinedPushDownMethodsImpl. The implementer of the UserDefinedPushDownMethodsImpl does not need to worry about processing CCI Records; the FunctionSet code will take care of that.

16.2 Sample scenario

This chapter has two sample scenarios using CMP over Anything. The first is an implementation using CMP over Stored Procedure in a database. The second is an implementation using Web services.

Development

Development of CMP over Anything does not differ from the development of a normal entity bean. Follow the steps to create a container-managed entity bean with push-down methods to support the CMP over Anything feature.

1. Generate the CMP Entity EJB with the usual wizards.
2. Define any Business logic push-down methods as abstract in the bean class.

3. Promote the abstract push-down methods to the interface, if needed.
4. Open EJB Deployement Descriptor and use the Pushdown tab to perform mapping.
5. Generate the push-down implementation and possibly complete the code.
6. Generate the code for deployment.

Assembly

There are no specific assembly tasks related to CMP over Anything in enterprise applications. Some of the information for CMP over Anything is stored in the EJB deployment descriptor; you can find those items in “Development” on page 437.

Unit test

WebSphere Studio Application Developer Integration Edition V5.1 is not capable of testing applications with CMP over Anything. You have to use either a standalone WebSphere Business Integration Server Foundation application server or the application server from WebSphere Studio Application Developer Integration Edition via the IBM Agent Controler.

Deployment

There are a few deployment-specific steps for applications using CMP over Anything.

Testing and runtime

There are various options for testing the application in the runtime environment. You can use the Universal Test Client from WebSphere Studio Application Developer Integration Edition when you deploy and test your application remotely using the IBM Agent Controller.

You can also install the Universal Text Client in the runtime environment for testing purposes. You can find the application in the WebSphere Studio Application Developer Integration Edition directory; make sure you change the classloader policy to `single` before you start using the client.

16.2.1 CMP over a database stored procedure

A stored procedure acts as an extension of the back-end store to the clients. Clients can access the back-end store through stored procedures in the function call using CALL statements. WebSphere Business Integration Server Foundation extends the existing J2EE CMP programming model to utilize the stored procedure feature for pushing down the CMP EJB methods to the back-end store. The mapping is done using the Application Server Toolkit or WebSphere Studio Application Developer Integration Edition V5.1. This will help the bean

developers to map the bean logic methods and the CMP EJB accessor methods to the stored procedures.

There is a PushDownMethodElement defined for each method that needs implementation with the stored procedure in WebSphere Studio Application Developer Integration Edition deployment descriptor extensions, which acts as a trigger. The logic that accesses the stored procedure is provided by the user-written class which plugs into the EJB support for WebSphere. Because of this, the bean's logic is simplified and uses the existing tested stored procedures.

Implementing the BeanUserDefinedPushDownMethod interface provides access to the stored procedure once the CMP EJB is deployed.

Developing CMP over a stored procedure

This section provides detailed steps to develop an entity bean using stored procedures at the back-end database for persistence.

This sample requires a database with a defined stored procedure. We used IBM DB2 for this scenario. You can find more details about creating the sample database and the stored procedure in "Building a stored procedure in DB2 for the CMP over Anything sample" on page 568.

1. Launch WebSphere Studio Application Developer Integration Edition with a new workspace and make sure you enable server targeting support.

   It is recommended that you use an empty workspace for the samples introduced in the book, for example: C:\projects\cmpa_sample.

2. Open the J2EE perspective, create a new J2EE 1.3 Enterprise Application Project with the name `cmpa`, using the server targeting Integration Server V5.1. Create a new EJB project for the application with the name `cmpaEJB`. Also create a new Web project for the application, with the name `cmpaWeb`.

Developing the CMP Entity Bean as usual

At this point, the EJB creation process is exactly the same as for the CMP Entity Bean. Once the bean is created, additional ejbCreate() methods will be developed.

1. Create the EJB using the details from the following figure, then click **Next**.

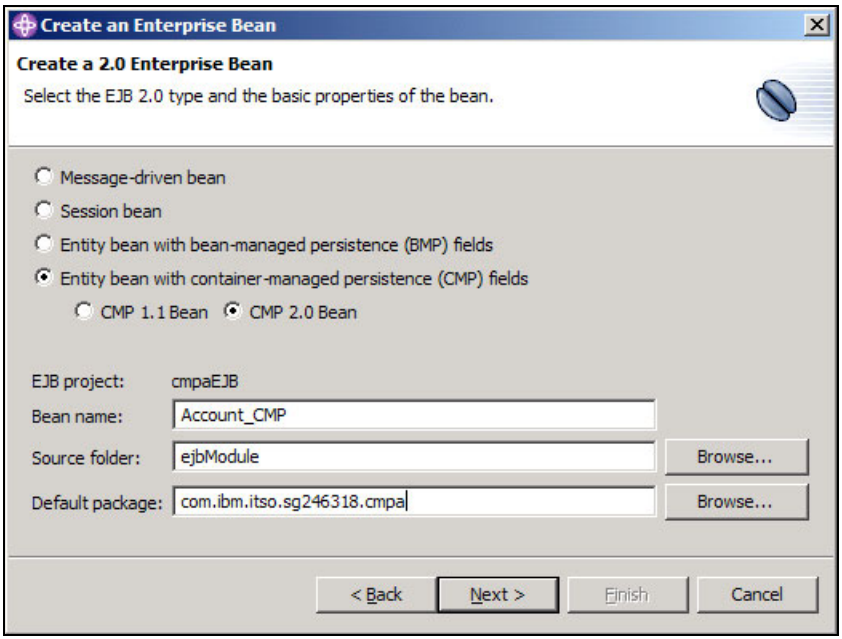


Figure 16-2 Creating a new EJB

2. Create the following CMP attributes:
   - `account - int - Key field`
   - `custno - int`
   - `name - java.lang.String`
   - `balance - float`
3. Click **Finish**.

Note: Do not create deployment code at this time. If you do so, WebSphere Studio Application Developer Integration Edition will show errors when creating the push-down method.

The reason is that we are deviating from the J2EE specification by creating an abstract business logic method. So, if you create deployment code, WebSphere Studio Application Developer Integration Edition will show a J2EE imcompatibility error.

The deployment code should be generated only after updating the push-down method with the logic needed and generating the code for the push-down method.

If you create CMP attributes for your entity bean, you can do that following the regular steps. Once the attributes are set, you can use the mapping tool to map the attributes to database fields; in most cases, you would probably use meet in the middle mappings if the database already exists.

Creating an abstract method as a push-down method

Once the CMP is created, open the bean class and add an abstract method. Do not create the implementation of the method. Just create the parameter it can take and the return type.

1. Open the Account_CMPBean.java source and insert the following line at the end of the class:

   ```
   public abstract java.lang.String getMyBalance();
   ```

2. Save and close the file.

Promoting the local method for the push-down method

Note: This method will be defined as the push-down method later.

1. Open EJB Deployment Descriptor, select the **Account_CMP** bean under the Enterprise JavaBeans section and click the **Pushdown** tab.
2. Click **Add** at the bottom of the window to add a new push-down entity. This will open up the push-down entity window.

   Select the back-end type **JDBC**. In our case, we have selected **JDBC** since we are going to call a Stored procedure. Select the CMP Entity Bean **Account_CMP** from the bean drop-down list. Click **Next**.

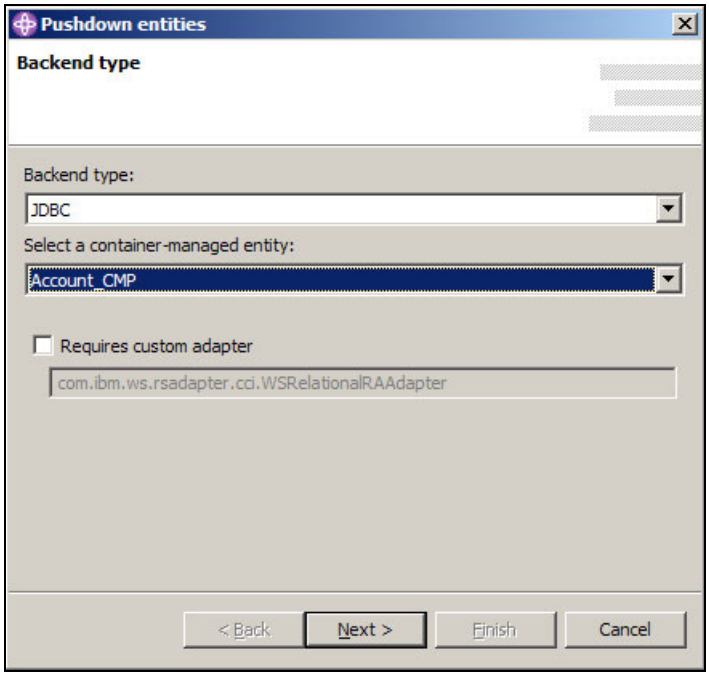


Figure 16-3 Back-end type for the entity bean

3. A new window opens up showing the methods available in the bean which can be used as push-down methods. Since there is only one abstract method available, it is preselected for a push-down method; in our case, it is the getMyBalance() method.

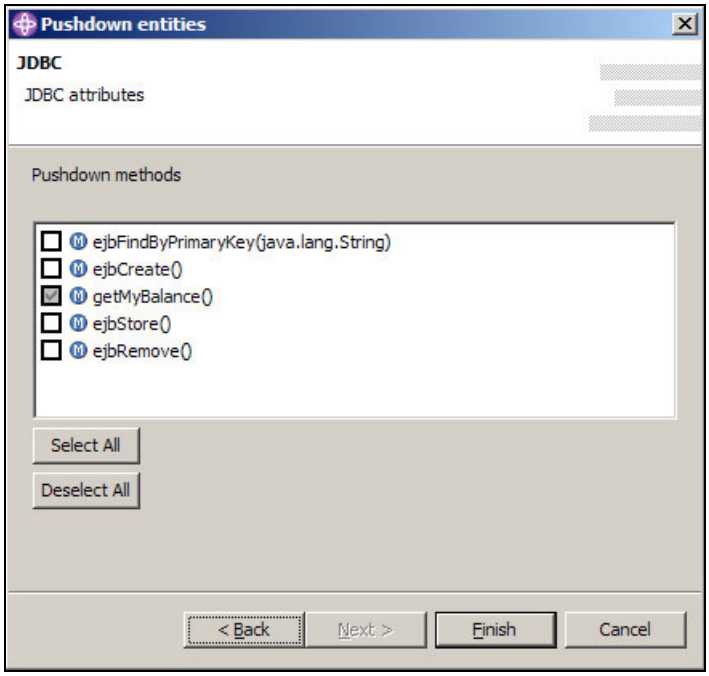


Figure 16-4 JDBC attributes for the push-down method

4. Click **Finish**. This will create a skeleton code for the push-down method.
5. Select the new push-down method created, **getMyBalance()**, under the Push-down methods section, then click **Edit** to add the JDBC SQL statement.

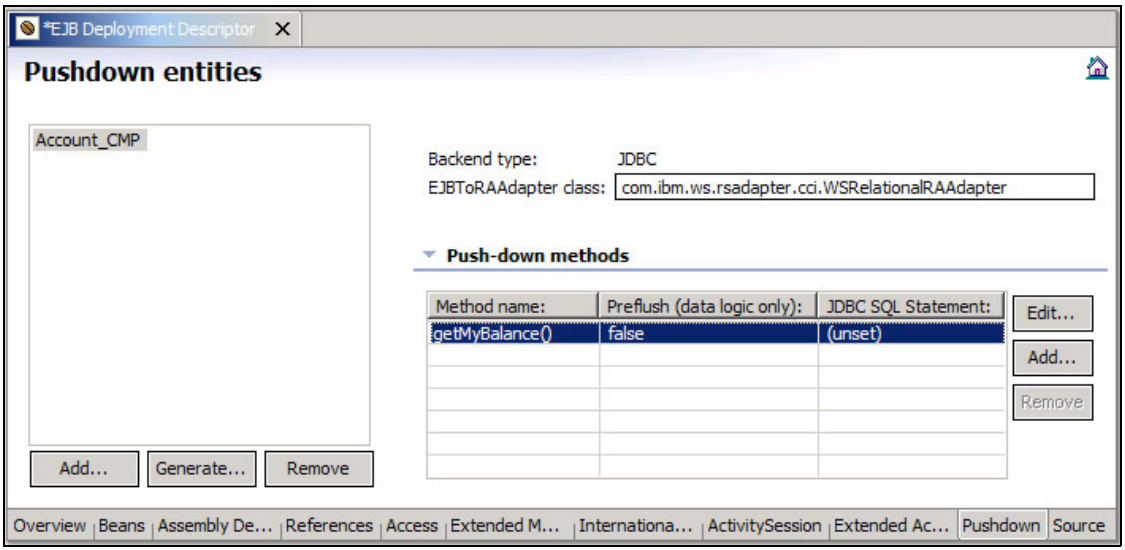


Figure 16-5 Push-down entities

6. Change Preflush to `true` and provide the stored procedure name with the required parameters, in our sample:

```
{call ITSO.ACCOUNT_SP(?,?)}
```

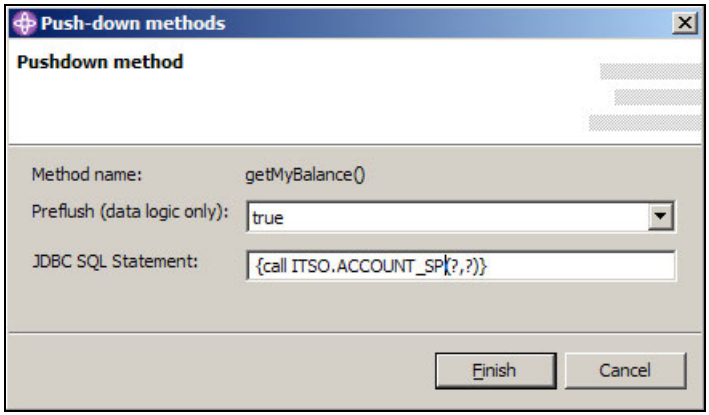


Figure 16-6 Push-down method details

7. Click **Finish**.
8. Click **Generate** in the Push-down entities window.
9. Make sure the **Account_CMP** bean is selected then click **Finish**.

10. Save and close the deployment descriptor file.
11. Once the push-down method is generated, we can create the deployment code. The push-down method skeleton has been created. We need to update the push-down method logic before generating the deployment code.

    When the push-down method implementation skeleton was generated, WebSphere Studio Application Developer Integration Edition generated and opened the file Account_CMPBeanUserDefinedPushDownMethods.java. If the file is not open, open it from the com.ibm.ITSO.websphere_deploy.jdbc package.

    The generated push-down skeleton method has a comment section where you need to insert the actual code. There are two places to insert code; these are indicated with comments.

    The first comment is as follows:

    ```
    // Begin customer-written code to populate callableStatement
    // Place code here like: callableStatement.setXXX(X, argX);
    // End customer-written code to populate callableStatement
    ```

    Replace it with the following code:

    ```
    callableStatement.setString(1,bean.getCustno());
    callableStatement.registerOutParameter(2, Types.FLOAT);
    ```

    You will also have to add the following import statement to make it work:

    ```
    import java.sql.Types;
    ```

    The second comment is as follows:

    ```
    // The returnValue variable should be set to the appropriate value
    ```

    Replace it with the following code:

    ```
    returnValue = "The value from Stored Procedure = " +
    callableStatement.getDouble(1);
    ```

12. Save and close the file.
13. If you have not generated the EJB to RDB mapping then do it now. In this example, we used DB2 8.1, with database name `CMPA` and schema name `EJB`.
14. Generate Deployment and RMIC code for the EJB.

Deploying the sample in the runtime environment

1. Export the enterprise application, using the name `cmpa-sp.ear`.
2. Make sure you have the database set up and configured as described in “Building a stored procedure in DB2 for the CMP over Anything sample” on page 568.
3. Start your WebSphere Business Integration Server Foundation application server.

4. Launch the Administrative Console and log in.
5. Create a J2C Authentication alias with the following details:
   – Alias: `AccountAlias`
   – User ID: `db2admin`, or your database administrator's ID
   – Password: the user's password

   Click **Apply**.
6. Create a JDBC driver and a Data source for the application database.
   a. Select **Resources** → **JDBC Providers**.
   b. Set the scope to `Node`, then click **New**.
   c. Select the JDBC provider **DB2 Universal JDBC Driver Provider (XA)**, then click **Apply**.
   d. Select **Data sources** at the bottom and click **New** to create a new datasource. Provide the following information:
      • Name: `AccountDS`
      • JNDI name: `jdbc/Account`
      • Component-managed Authentication Alias: `AccountAlias`
      • Container-managed Authentication Alias: `AccountAlias`

      Click **Apply**.
   e. Select **Custom Properties** at the bottom.
   f. Set the databaseName property to reflect the database name, in our example: `ITSO_CMP`.
7. Save the configuration for WebSphere.
8. Install the resource adapter for CMP over Anything EJBs.
   a. Select **Resources** → **Resource Adapters**.
   b. Set the scope to `Node`, then click **Install RAR**.
   c. Browse for the cmpaAdapter.rar file under the <WebSphere_root>/installableApps directory. Click **Next**.
   d. Provide the following information:
      • Name: `CMPA Resource Adapter`

      Click **Apply**.
   e. Select the new resource adapter then click **J2C connection factories** at the bottom of the page.

f. Click **New** to create a new connection factory; provide the following information:

- Name: `AccountCF`
- JNDI name: `jdbc/Account`
- Component-managed Authentication Alias: `AccountAlias`
- Container-managed Authentication Alias: `AccountAlias`

Click **OK**.

9. Save the configuration for WebSphere.
10. Start the installation of the application cmpa-sp.ear, then select **Applications** → **Install New Application**.
11. Browse for the cmpa-SP.ear and click **Next** and **Next** again.
12. Follow the installation steps and stop at *Provide default datasource mapping for modules containing 2.0 entity beans* and at Map datasources for all 2.0 CMP beans.
13. At “Provide default datasource mapping for modules containing 2.0 entity beans“, click the **+** next to Apply Multiple Mappings to open the list of available datasource JNDI names.
14. Select the **eis/jdbc/Account_CMP** next to *Specify existing Resource JNDI name*, check the box next to the cmpaEJB item, then click **Apply** under the JNDI name.
15. At *Map datasources for all 2.0 CMP beans* click the **+** next to *Apply Multiple Mappings* to open the list of available datasource JNDI names.
16. Select **eis/jdbc/Account_CMP** next to the *Specify existing Resource JNDI name*, check the box next to the *Account_CMP* item, then click **Apply** under the JNDI name.
17. At the last step, click **Finish**.
18. Save the configuration for WebSphere.
19. Start the new application, cmpa, in WebSphere.

Testing the sample

In this section we will use the Universal Test Client installed under WebSphere (the application can be found under WebSphere Studio Application Developer Integration Edition).

1. Launch a Web browser and open the URL: `http://localhost:9080/UTC/`.
2. Select **JNDI Explorer**, then drill down to **[Local EJB beans]** → **com** → **ibm** → **itso** → **sg246318** → **cmpa** and click the **Account_CMPLocalHome** item.

3. Use the Account_CMPLocal findByPrimaryKey(Account_CMPKey) method to find a bean. If you have the database populated as described in “Building a stored procedure in DB2 for the CMP over Anything sample” on page 568 then use the int value `11` for the lookup. Once the bean is found, click the **Work with Object** button; the new object appears on the navigation bar.
4. Use the String getMyBalance() method of the Account_CMPLocal object to return the balance for the customer. The result should be the following string:

```
The value from Stored Procedure = 60.0 (java.lang.String).
```

17

Chapter 17

Application profiling

Application profiling enables you to identify particular units of work to the WebSphere Application Server runtime environment. The runtime can tailor its support to the exact requirements of that unit of work. Access intent is currently the only runtime component that makes use of the application profiling functionality. The application profile itself is a set of access intent policies that should be selectively applied for a particular unit of work (a transaction or ActivitySession).

In WebSphere Business Integration Server Foundation V5.1, application profiling consists of tasks and profiles. A *task* is a named unit of work within a distributed application whereas a *profile* is simply a mapping of a task to a set of access intent policies that are configured for entity beans.

17.1 Prerequisites

The WebSphere Business Integration Server Foundation InfoCenter provides valuable information regarding various development, deployment and administration aspects. The InfoCenter is located at:

http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp

It is a good idea to familiarize yourself with the following topics in the WebSphere Business Integration Server Foundation InfoCenter.

- There is additional information available about application profiling administration and development in the InfoCenter. Using WebSphere Business Integration Server Foundation InfoCenter, navigate to **All topics by feature** → **Applications** → **Application services** → **Application profiling**.
- It is important to gain familiarity with development tasks associated with Application Profiling and Access Intents. Next few sections assume this knowledge and refer to the InfoCenter where possible. Using WebSphere Business Integration Server Foundation InfoCenter, navigate to **Developing** → **Task overviews** → **Application profiling**

Familiarize yourself with the following topics in WebSphere Studio Application Developer Integration Edition help:

- EJB Deployment tool. This can be navigated by going into WebSphere Studio Application Developer Integration Edition Help Content and then navigating to **WebSphere Studio** → **Developing** → **EJB applications** → **EJB application development overview** → **Tools for developing EJB applications** → **EJB deployment tool**
- Defining Access Settings. This can be navigated by going into WebSphere Studio Application Developer Integration Edition Help Content and then navigating to **WebSphere Studio** → **Developing** → **EJB applications** → **Editing EJB deployment descriptor** → **Defining access settings** → **Adding bean-level access intent for EJB 2.0 entity beans**
- Extended Access. This can be navigated by going into WebSphere Studio Application Developer Integration Edition Help Content and then navigating to **WebSphere Studio** → **Developing** → **Enterprise services** → **Extended Access (Application Profiling)**

17.2 Overview

WebSphere Business Integration Server Foundation runtime can make optimizations for an application based on “hints” on the behavior of the

application. Application Profiling and Access Intents feature of WebSphere Business Integration Server Foundation allows for declaratively or dynamically configuring these hints at assembly time or at runtime.

Using Access Intents, EJB container can be configured to provide optimal performance based on a specific type of EJB use. Various Access Intent hints can be declared at assembly time to indicate to WebSphere resources (such as the container and the persistence manager) to provide the appropriate Access Intent services for every EJB request.

EJB 2.0 introduces the Container Managed Relationships but the EJB container does not have enough information to optimize access. A good example of such shortcoming is EJB read ahead, that is, how far to read ahead in the Object Relationship graph. Read-aheads, combined with other forms of application profiling and access intents, can have dramatic impact on the runtime performance.

The concept of Application profiling and Access Intent further extends beyond Container Managed Persistence. Application profiling policies are hints that also apply to Relational Resource Adapter, for example. Following are some common applications of Application Profiles and Access Intents:

- Persistence manager makes decisions about isolation level, cursor management, and so on.
- EJB container influences the management of EJB collections.
- Relational Resource Adapter (RRA) provides prefetch hints in defined increments to control the number of rows read from database at a time. WebSphere does not provide paging nor does the RRA provide prefetch. Instead, a prefetch hint is passed to the database telling how many rows are going to be read so that the database can optimize the access.

Profiling levels

Profiling can be configured at different levels, depending on your needs. Table 17-1 shows the different Profiling configurations available as they are seen in the Application Server Toolkit and their granularity level.

Table 17-1 Profiling configurations

PME	Scope
Dynamic Query	Entity
Access Intent	Method
Application Profiling	Unit of Work

You can set different Profiling configurations on the same EJB, so apparently there will be collisions. But a clear priority hierarchy has been defined upon these three levels of Profiling. Notice that there are three types of Profiling, each with a different level of granularity.

Access Intent

Access Intent lets you associate an Entity CMP EJB method with an Access Intent Policy. An Access Intent Policy is a set of properties defining how the EJB container should access the persistence layer.

This feature is provided in WebSphere Application Server as well as in the WebSphere Business Integration Server Foundation. It is a step forward in performance tuning but yet limited.

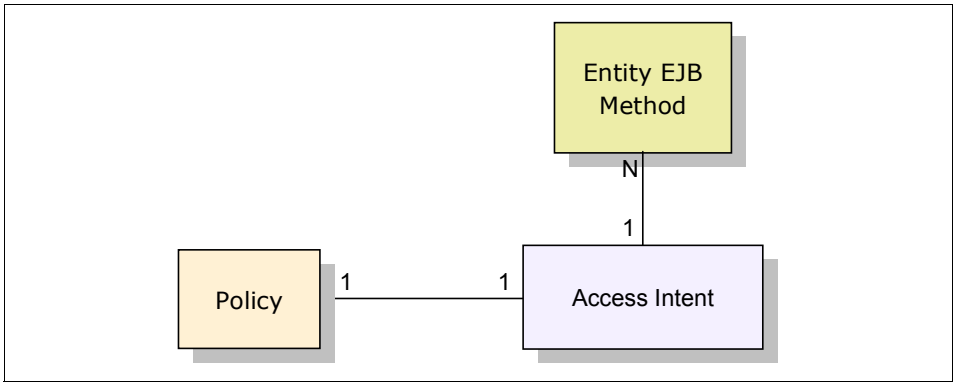


Figure 17-1 Access Intent

Because Access Intents are defined at method level, you can assign different Access Intent Policies to EJB's create, remove, setter and getter methods, achieving some performance tuning. If you are unable to decide which Access Intent to apply in deciding upon the caller or client of the Entity EJB, this issue is covered by Application Profiling.

Application Profiling

Application Profiling lets you associate task names to Session and Entity EJB methods and group a set of tasks under a set of Access Intents. This provides the capability of associating at runtime an Entity EJB method with a specific Access Intent depending on the task under which it is called. So the binding is caller dependent and resolved dynamically at runtime.

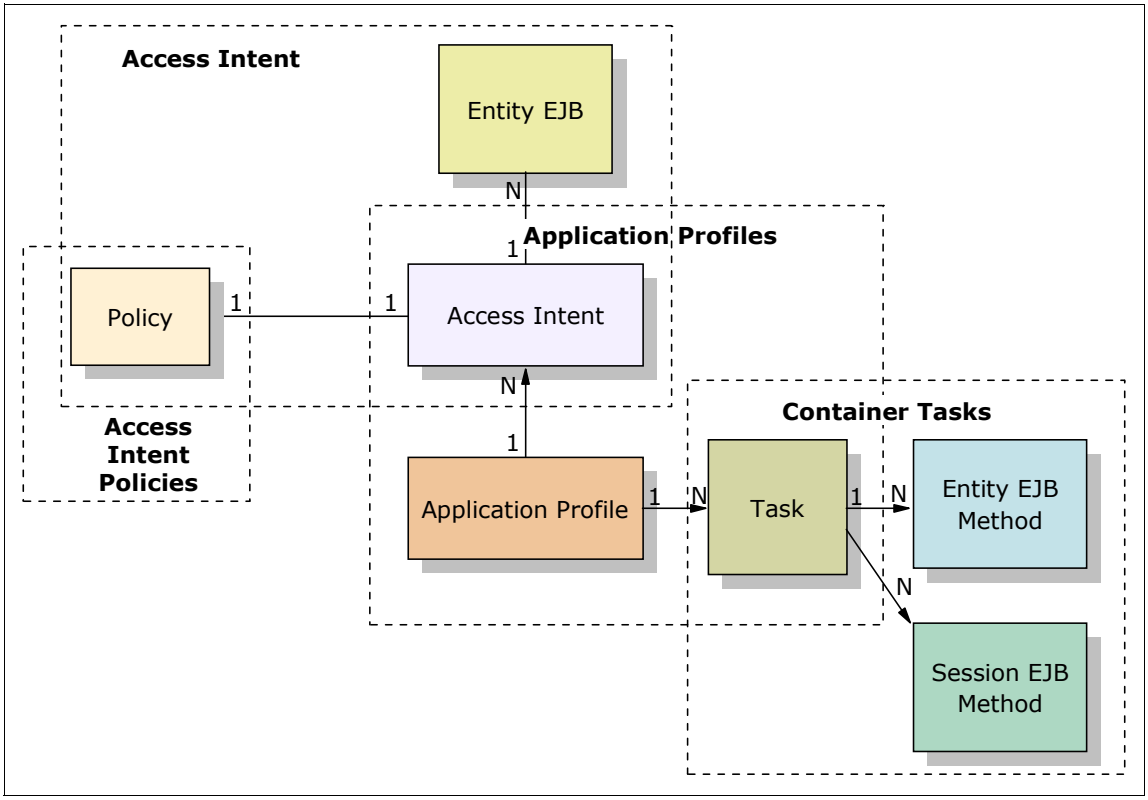


Figure 17-2 Application Profiling diagram

A task is associated with an Application Profile at deploy time. At runtime, the task name gets propagated and its arrival identifies the caller. An Application Profile associated with that caller task defines the Access Intents to be applied.

Default Access Intent

There is another level of configuration for Access Intent. You can set an Entity EJB to be accessed with a specific Access Intent when it is loaded as a result of a Dynamic Query. In fact, this configuration is used by default if no other Access Intent is specified. This Access Intent is defined at bean level.

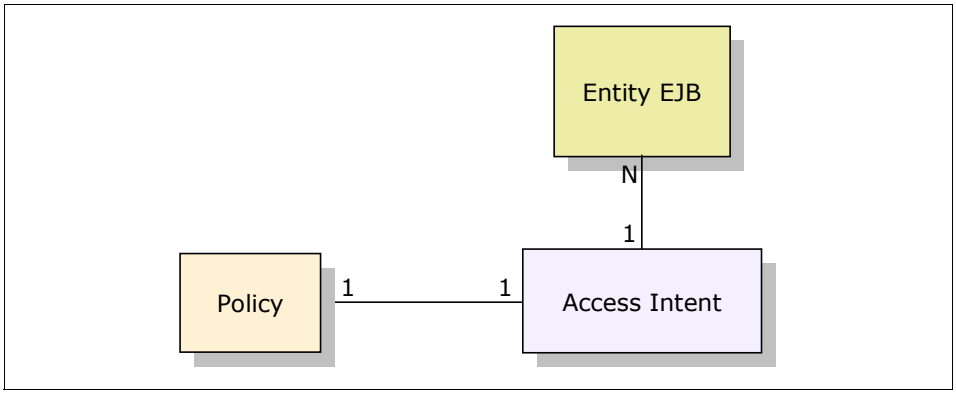


Figure 17-3 Default Access Intent, Dynamic Query Access Intent diagram

Note: Access Intents are defined at bean level. Access Intents are defined at method level. Compare the Access Intent diagrams in Figure 17-1 on page 452, Figure 17-2 on page 453, and Figure 17-3.

Profiling prioritization

When different Access Intents are defined on the same entities, prioritization is applied to resolve which Access Intent to apply. Figure 17-4 on page 455 shows the decision algorithm used to apply an Access Intent when many Access Intents have been defined or are applicable for the current request.

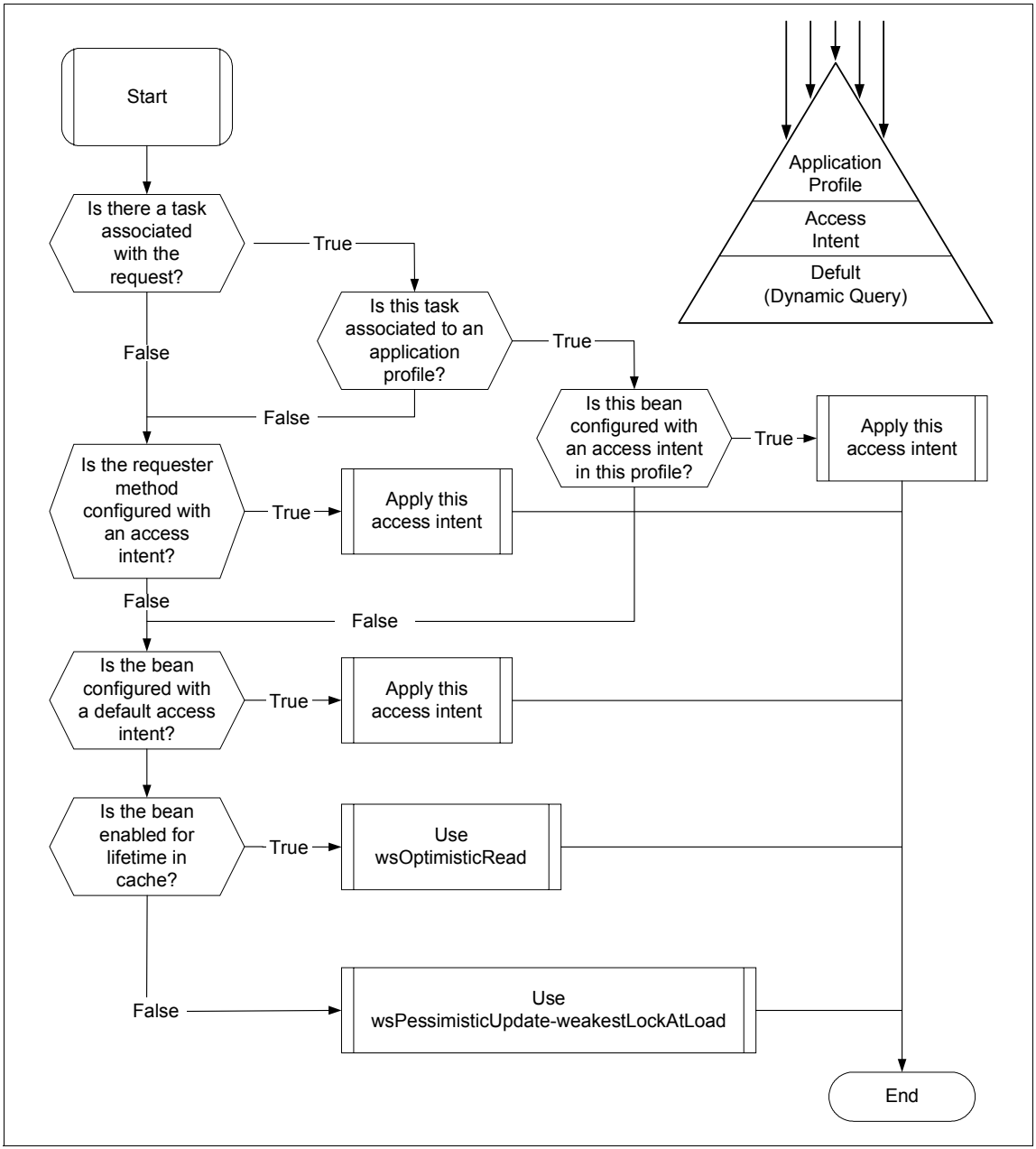


Figure 17-4 Application Profiling prioritization

17.3 Planning

Application Profiling and Access Intent, hereafter referred to as Application Profiling or just Profiling, are meant to be configured at deployment time through the extended Deployment Descriptors. For flexibility, an API is exposed for Entity BMP EJBs. This section shows you how to configure Application Profiling and how to work with the exposed API.

Steps for setting up Application Profiling

This section provides an overview and discussion of the major steps needed to set up application profiles. Figure 17-5 highlights these steps.

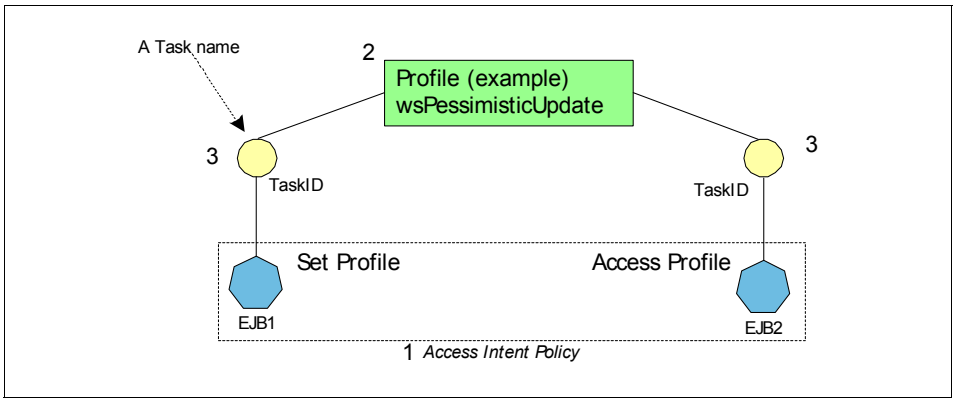


Figure 17-5 Steps for setting Application Profiles

1. **Create custom Access Intent policy**

   Before setting up the profile, there has to be scenario that could benefit from access intents and application profiling. You need a policy where you can express whether you intent to read or update, whether you intent to use an optimistic or pessimistic approach, whether you intent to use pre-fetching mechanism or not. WebSphere Business Integration Server Foundation comes with 7 predefined Access Intent Policies. You can use those as they are, or "tweak" them at the time you use them to provide for necessary variations. You can also create your own policies on EAR file by using Application Server Toolkit.

2. **Create Application Profile**

   Next a custom application profile must be created using the Application Server Toolkit. The application profile is a definition that includes

   – a number (one of more) of access intents applied to one or more EJBs,

   – a number of task names that identify a logical "business task".

3. **Container Task**

   Container task definition points to a "task name" you associated with a profile. Container task definition can be applied to EJB methods (session EJB). When a client calls that particular method on the session EJB, the profile that is associated with the task is going to be used, and therefore the access intents specified in the profile are going to be used for the Entity EJB.

   Important: It is important that the method on which you created the container task initiates a transaction, because the application profile definition "flows" with the transaction context to the EJBs that are called by the session bean.

An association graph between client methods, Tasks, Profiles, Access Intents and Entity EJBs methods is shown in Figure 17-2 on page 453.

17.3.1 Access Intent Policies

There are several different Access Intent Policies available in WebSphere Business Integration Server Foundation, these are listed below. Each policy is really a configurable data access parameter specified in Access Intent Policy for the persistent manager:

1. Concurrency Control
2. Access Type
3. Read-ahead - Container Managed Relationships
4. Collection Scope
5. Collection Increment
6. Resource Manager Pre-fetch Increment

The following sections discuss each Access Intent Policy in detail

Concurrency Control

Concurrency control allows a choice of pessimistic or optimistic access.

Pessimistic Access

Pessimistic Access implies that the Entity EJB is locked for the duration of the transaction. At a minimum, no one, except the call, can modify that EJB instance. In caller's transaction, it is guaranteed that the data it is dealing with is not stale. This level of locking provides data integrity. On the downside, pessimistic access provider lesser concurrency. Typically, pessimistic access results in a higher isolation level on the RDB.

Optimistic Access

Optimistic Access reduces the demands in terms of data integrity. During an optimistic transaction, no locks are maintained on the instance. This implies that another transaction can modify the instance, and commit those changes too. When the optimistic transaction tries to commit, it will verify that someone else hasn't changed the data in the meantime. If the data did change, the commit will fail, and the end user has the option of resubmitting the transaction.

Optimistic access improves concurrency, but introduces complexity in application design where programmers must take into account situations where exceptions might be thrown because of data integrity issues.

Note: Optimistic concurrency always implies that dead locks are possible, but are less likely.

Access Type

When defining application profiles, it is possible to choose between read or update access intents.

Read access intent

Read intent can be optimistic or pessimistic. If pessimistic read intent is specified and an update method is called on a CMP EJB, the container throws: com.ibm.ws.ejbpersistence.utilpm.UpdateCannotProceedWithIntegrityException exception. For BMP EJBs, the EJB should throw javax.ejb.EJBException exception. Typically read intents result in lower contention.

if pessimistic read is used, it will not be possible to escalate the intent to perform an update. For example, if a transaction specific read intent, and then it calls a method that actually updates (like a setter method), the container will throw an UpdateCannotProceedWithIntegrityException exception.

The container prevents from loading the same EJB in the same transaction using incompatible access intents. For example, if EJB is loaded with pessimistic read intent, and then subsequently loaded with an update intent, the InconsistentAccessIntentException is thrown.

Note: Under pessimistic concurrency control, transactions started with read intent are not capable of supporting an update.

Update access intent

Update intent can be optimistic and pessimistic, as the read intent before. Pessimistic update is the most “stringent” form of intent, that is, it is the most demanding in terms of isolation.

There are three combinations that are available for pessimistic update:

- Weakest Lock at Load

  Although update intent is specified, a container acquires read lock first. If the transaction then updates, lock escalation will be pursued. This allows better concurrency than if an update lock was attempted at the beginning, but it may result in dead locks. This is the default Access Intent for all CMP EJBs.

- Exclusive

  This is the most restrictive option and provides highest level of locking, hence resulting in the most restrictive isolation levels. It should be used careful when used with large result sets, for example in finder methods. It is used to prevent phantom reads and non-repeatable reads.

- No Collision

  No collision behaves like Exclusive option but it allows for EJB instances to be cached in memory. There, the isolation level requirement are not so stringent. In other words, with this option there is no concurrency control, that is, it selects without locks and update without checks. This option can be used safely only if database tables, where EJBs are persisted, are not shared with any other software other than WebSphere, otherwise data loss might occur.

Important: Do not confuse the *lock escalation* concept in an Access Intent context with *lock escalation* in a database context. In a database context, escalation of locks refers to the internal mechanism of the database that reduces the number of locks. In a single table, locks may be escalated to a table lock from many row locks. In an Access Intent context, *lock escalation* is a synonym of *lock conversion* in a database context. It happens when a more restrictive lock than the one held is needed. A read lock on an object may be converted to an update lock.

Read-ahead

This parameter is applied to multiple Entity EJBs. It is only available for Entity EJBs with a Container Managed Relationship (CMR) defined on them. Possible values are the names of any CMR field on the current Entity EJB. At the time a findByPrimaryKey() method is invoked, the container uses the read-ahead hint to pre-fetch data for the specified related objects.

For example, as shown in Figure 17-6, when a findByPrimaryKey() operation is performed on Manufacturer table, the container could run a join query to get not only the Manufacturer information, but also the information for all the related cars and their sales information.

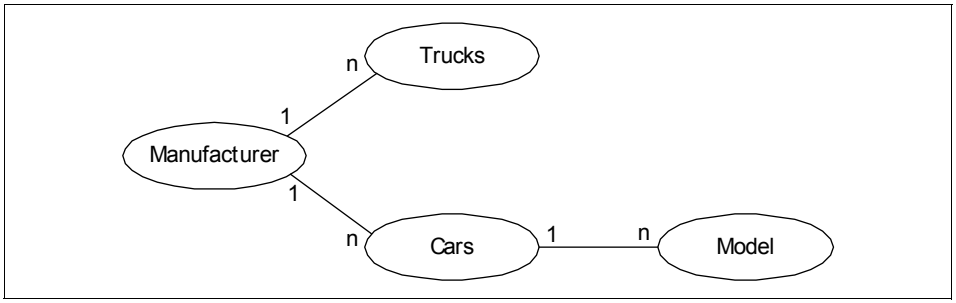


Figure 17-6 Data model representing Read-ahead Access Intent scenario

Tip: This operation is beneficial only when the task looking up the Manufacturer information is also going to use related EJBs.

The read-ahead intent is relatively complex in that it deals with multiple Entity EJBs. When navigating through a CMR graph by calling a get method on a CMR field, that method call effectively corresponds to a finder method on the related EJBs. The application profile associated with the source of the relationship will be propagated to the target as well. In WebSphere Business Integration Server Foundation, the same EJB cannot be loaded twice with different intents. Hence the target EJBs must be configured to have the same access intent for the finder method to work properly. If this condition cannot be met, read-ahead for the target EJBs cannot be configured.

For example, in Figure 17-7, if you want to read-ahead from Manufacturer to Trucks and that an optimistic intent is defined for the Manufacturer. If trucks was not configured with a optimistic intent, the following may happen:

- department is loaded
- the container optimistically loads some trucks
- in the same transaction, there is a findByPrimaryKey() on a truck that was already optimistically loaded, but this time we have to use the default pessimistic intent.

The last operation above will cause an exception. That is why Application Server Toolkit will force you to define an optimistic intent on the Trucks EJB and then you will be able to configure the read-ahead intent.

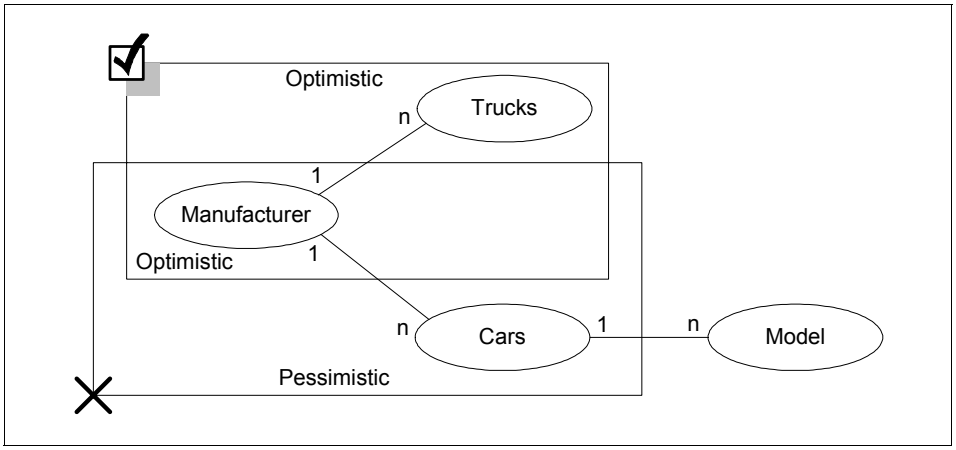


Figure 17-7 Access Intents and Read-ahead

Tip: In general, you must specify consistent access intent on the related EJBs.

Collection Scope

Collection scope intent is used to control the lifecycle of a "lazy" collection. When you call a finder method optimistically, you get a "lazy" collection back from the container. The collection can be loaded in memory "in chunks", so that as you iterate through it, the elements that you are looking for are already in memory.

However, a “lazy” collection normally expires at the transaction boundary. The default value for the collection scope hint is "transaction" boundary. The scope can be extended by specifying "activity session" for the collection scope and have the collection survive multiple transactions as long as they are all taking place within the same Activity Session.

- **Transaction** is the default value. The collection will no longer be usable at the end of the transaction that created it. References will be dropped and objects will be ready for garbage collection.
- **ActivitySession**. The collection may be accessed until the end of the Activity Session under which the collection was retrieved, spanning multiple transactions.

In cases where an attempt is made to iterate over a collection that has "expired", following exception is thrown:
`com.ibm.websphere.ejb.container.CollectionCannotBeFurtherAccessed`

Collection Increment

As mentioned previously, when a finder method is invoked optimistically, the container returns a “lazy” collection. Collection increment intent controls how many elements can be pre-fetched in a single operation. The number of elements loaded can influence application performance:

- Too many objects may clutter the cache.
- Too few may cause the iterator to wait for incoming data.

Tip: Collection increment is useful in optimistic approach.

Resource Manager Pre-fetch Increment

This parameter suggest how many rows should be retrieved from the underlying relational database in a single operation. If set to zero, the JDBC driver will ignore it. This parameter may be completely ignored by some database implementations that have their own optimization mechanism. It is a hint to the JDBC driver and may or may not be taken into account.

This number may differ from the collection increment discussed earlier. For example, it is possible to have a data mapping that is scattered across multiple tables. In that case, this access intent type is very useful.

Summary

The table below, Table 17-2, summarizes the various options available to configure access intent policies.

Table 17-2 Access Intent Polices Summary

Access Intent Type	Options
Concurrency Control	Pessimistic, Optimistic
Access Type	Read, Update, WeakestLockAtLoad
Read-ahead (Container Managed Relationships)	Allows pre-fetching related EJBs
Collection Increment	Allows pre-fetching certain number of elements of a collection of EJBs
Collection Scope	Transaction, Activity Session
Resource Manager Pre-fetch Increment	Hint to JDBC driver to allow pre-fetching number of rows

17.3.2 Predefined Access Intent Policies

WebSphere Business Integration Server Foundation ships with seven predefined access intent policies, as noted in Table 17-3.

Table 17-3 Predefined Access Intent Polices

Access Intent Policy	Description
wsWeakestLockAtLoad **[DEFAULT]**	Starts a pessimistic read, then escalate when update is required. This is the default policy.
wsPessimisticUpdate	Gets update lock at the beginning of the transaction. Prevents dead locks.
wsPessimisticRead	Read locks are held for the duration of the transaction. Updates are not permitted.
wsOptimisticUpdate	No locks are held. Updates are allowed. If data originally queried has changed since the read, the update will produce an exception and will not take place.
wsOptimisticRead	No locks are used. Updates are not allowed.
wsPessimisticUpdateNoCollision	No locks held but updates permitted. Provides no concurrency control. Can lead to data corruption if misused.
wsPessimisticUpdate-Exclusive	Read or update locks are held for the duration of the transaction on the entire range of data affected by the SQL statement. Misuse may result in dead locks.

The following are a few key points that should be kept in mind regarding predefined access intent policies:

- All optimistic access intents have the collection increment set to 25.
- All access intents have the read-ahead parameter set to black (no read-ahead)
- All pessimistic access intents have collection set to 1 (except for wsPessimisticUpdateNoCollisions where it is 25). In this case you need to make sure that WebSphere Persistence Manager is the only user for those database tables, or data loss and inconsistency may occur as previously indicated.
- The wsWeakestLockAtLoad is pessimistic update. However, it initiates data access in read mode, and tries to escalates the lock level as needed

Note: Optimistic locking results in better concurrency handling. However, the application have to be engineered to catch optimistic exceptions and to avoid deadlocks.

17.3.3 Isolation Levels and Access Intents

This section describes various database isolation levels that an access intent may be mapped to. A summary of supported database, predefined access intents, and the corresponding isolation levels is also provided.

Read Committed

A read committed transaction isolation level ensures only committed data is read, that is, no *dirty reads* allowed. However, no locks are maintained for the duration of the transaction. This implies that another process may change and commit the data you are reading, leading to a non-repeatable read. Also rows matching your criteria may be inserted or deleted, causing a *phantom read*. In database terminology, this isolation level is called cursor stability.

Repeated Read

The repeatable read transaction isolation level works like the read committed but it also ensures repeatable reads. In other words, another process will **not** be able to update a row you obtained until your transaction is complete. However, *phantom reads* are still possible, as another process might insert a row that matches the selection criteria; if you repeat the query, you would get the newly inserted row. In database terminology, this isolation level is called read stability.

Seralizable

A serializable transaction isolation level ensures repeatable reads and disallows phantom reads by preventing other processes from inserting or deleting rows that would modify your result set until your transaction has completed. In database terminology, this isolation level is called repeatable read.

Summary

The table below, Table 17-4, summaries the mapping between the various access intents and the corresponding isolation levels on the supported relational databases.

Table 17-4 Isolation Levels and Access Intents mapping for databases: Summary

	DB2	Oracle	Sybase	Informix	Cloud scape	SQL Server
wsWeakestLockAtLoad (default)	RR	RC	RR	RR	RR	RR
wsPessimisticUpdate	RR	RC	RR	RR	RR	RR
wsPessmisticRead	RR	RC	RR	RR	RR	RR
wsOptimisticUpdate	RC	RC	RC	RC	RC	RC
wsOptmisiticRead	RC	RC	RC	RC	RC	RC
wsPessmisticUpdateNoCollosions	RC	RC	RC	RC	RC	RC
wsPessimisticUpdateExclusive	S	RC	S	S	S	S

17.3.4 Access Intent Decision

Now that we have discussed Access Intent and various Isolation levels, a key point in making an application perform better is how to pick the right Access Intent. This decision has to be based on the intention of the transaction involved and the isolation level that the transaction requires.

Pessimistic Concurrency Control Decision

Under pessimistic concurrency control, transactions that start with read intent are not prepared to support an update with integrity because locks are not held at the time of SELECT. Hence, if a bean is loaded into transaction with read intent and a CMP field is subsequently updated (through a setter method), Persistent Manager must throw a CannotUpdateEntityException exception.

Table 17-5 Pessimistic Concurrency Control Decision table

Access Type	PM Action	1st Method in Transaction	Subsequent Methods in Transaction	Isolation level	Comments
update (default)	Select for Update	Reads	Updates	RR	
update	Select for Update	Updates	Reads	RR	

Access Type	PM Action	1st Method in Transaction	Subsequent Methods in Transaction	Isolatio n level	Comments
read	Select (not for update)	Reads	Updates	RC	throws exception
read	Select for Update	Updates	NA	RC	throws exception

Optimistic Concurrency Control Decision

Optimistic concurrency always implies that deadlocks are possible, because no real lock is required on the object until the transaction is about to commit. However, lock escalation is always possible with optimistic concurrency. Even if the transaction is started with an optimistic read intent, you still have the freedom to decide to update the instance at a later time.

Table 17-6 Optimistic Concurrency Control Decision table

Access Type	PM Action	1st Method in Transaction	Subsequent Methods in Transaction	Isolation level	Comments
update (default)	Select (not for update)	Reads	Updates	RC	deadlock potential
update	Update	Updates	Reads	RC	deadlock potential
read	Select (not for update)	Reads	Updates	RC	deadlock potential
read	Select (note for update)	Updates	Reads	RC	deadlock potential

Summary

Figure 17-8 on page 467 shows a better approach to decide which Access Intent should be chosen, taking into consideration the intention of the transaction and the isolation level that the transaction requires.

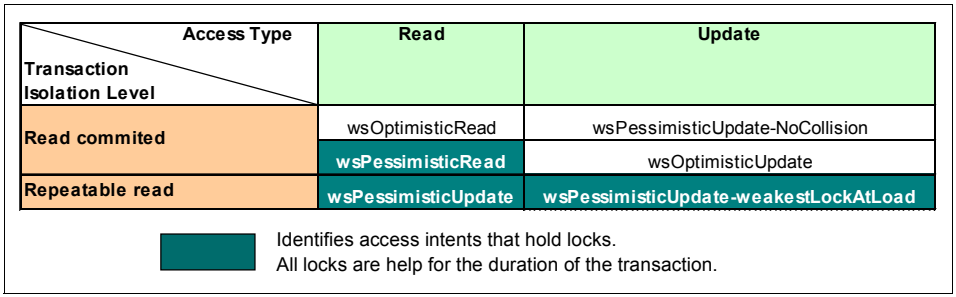


Figure 17-8 Access Intent decision table

17.3.5 Switching Access Intents within a Single Transaction

Under some circumstances, it may happen that client code attempts to access the same EJB or even the same EJB instance with different intents. These situations should be avoided because of the potential for runtime exceptions or deadlocks. In general, if you try to access the same EJB within the same transaction using different intents, the Persistent Manager will perform a number of checks, in order to prevent deadlocks as much as possible, and in order to prevent data loss or data inconsistencies.

If you try to use intents that imply different isolation levels for same EJB, the Persistent Manager will attempt to start a new connect to accommodate the various isolation levels. It is not possible to switch isolation levels on the same db connection. Multiple connections can be managed in s single transaction only if the database is XA capable. If you are using single-phase datasource, the situation described above will result in exceptions.

If the same instance of EJB is accessed using an incompatible access intent type, an exception may be thrown, that is, InconsistantAccessIntentExcepton. For example, Pessimistic Read cannot switch to Pessimistic Update, or Optimistic intents cannot be converted into Pessimistic intents.

One very common situation where same instance of an EJB is accessed differently in the same transaction is when Container Managed Relationships are involved, this is illustrated in Figure 17-9.

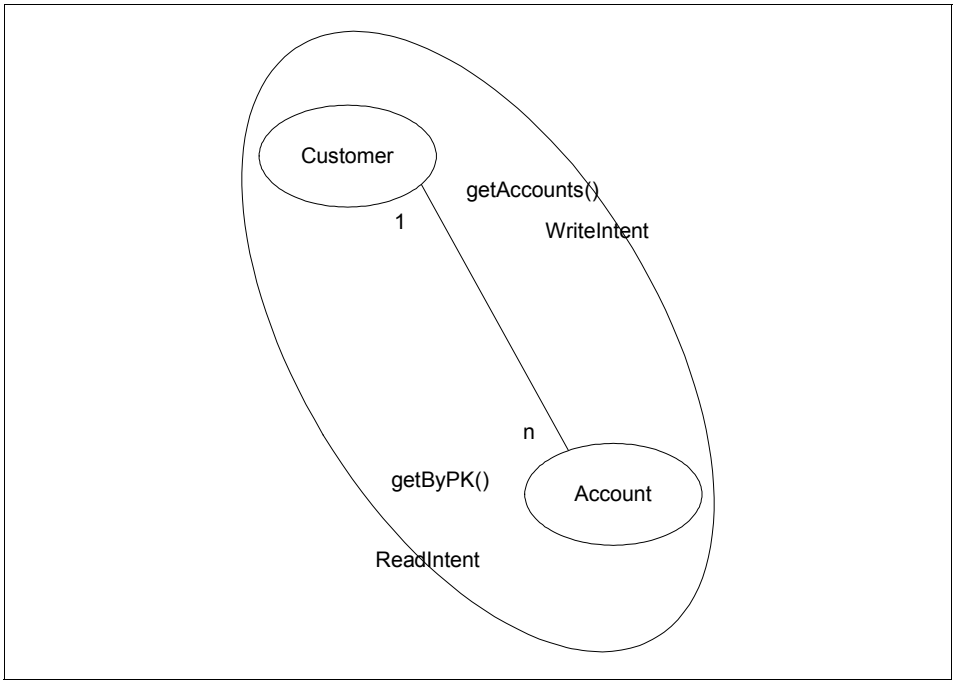


Figure 17-9 Container Managed Relationship with access intent

One client may look up an account using getByPrimaryKey(), on which a Pessimistic Read intent is defined. Later, but within the same transaction, the client accesses the customer that happens to own that account. The customer is accessed with a write intent. When getAccounts() is called, the collection of accounts include the account that the client already holds in read only mode. This results in a InconsistentAccessIntentException exception. Sample code that would cause this condition is shown below:

Example 17-1 Switching Access Intents sample code

```
accountHome.getByPrimaryKey("001"); // read intent
//...
customer.getAccount(); // includes account "001", inconsistent intent
//...
```

Reminder: If you load the same instance of EJB with different access intents that imply the same isolation level, you may get:

- a deadlock
- InconsistantAccessIntentException exception

These situations are much more common on the base application server rather than in WebSphere Business Integration Server Foundation. In the base application server, you would be using access intents specified on the individual methods of the EJB, rather than intents specified on the Application Profile (and therefore applicable to the entire transaction scope).

17.4 Assembly

This section guides you through the configuration of application profiles for an enterprise application. You can use either WebSphere Studio Application Developer Integration Edition or the Application Server Toolkit to define your application profile tasks and policies together with Access Intents.

The steps below are not configuring application profiling to a specific application, they are simply sample steps to show you how to do it for any application.

1. Open the EJB deployment descriptor for the EJB module.
2. You can set up access intents for entity beans on the bean and on the method level under the Access tab. Access Intent is a base application server functionality in WebSphere Application Server.
3. Switch to the Extended Access tab. During the next couple steps we are going to set up application profiling for an application.

   You have two options to start with for configuring application profiling for your application: Container-Managed Tasks (CMT), Application-Managed Tasks (AMT). Application Profiling Tasks are set up for EJB methods (CMT) or EJBs (AMT). You can also combine the two different task types in one application.

   a. Container-Managed Tasks show on the upper left side of the page.

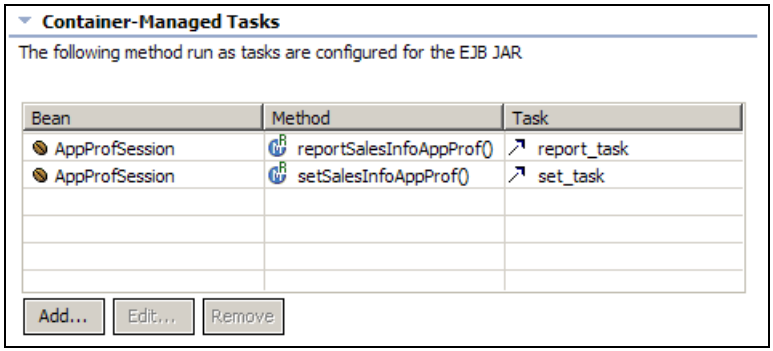


Figure 17-10 Container-Managed Tasks

To create a new task, click **Add...**

i. On the first panel select the bean, then select the method you want to assign the task to.

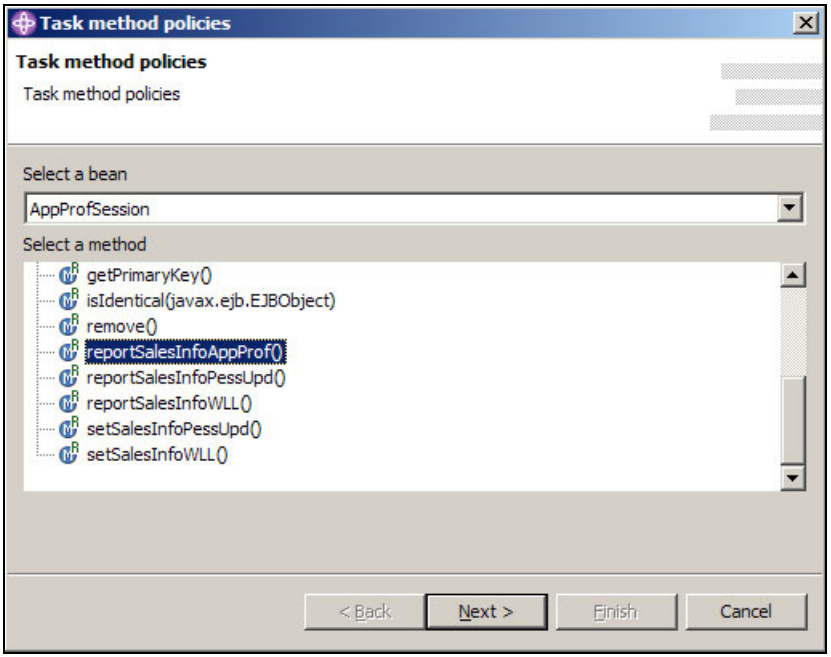


Figure 17-11 New Container-Managed Task 1.

Click **Next**.

ii. Name the task that you are going to configure later, then click **Finish**.

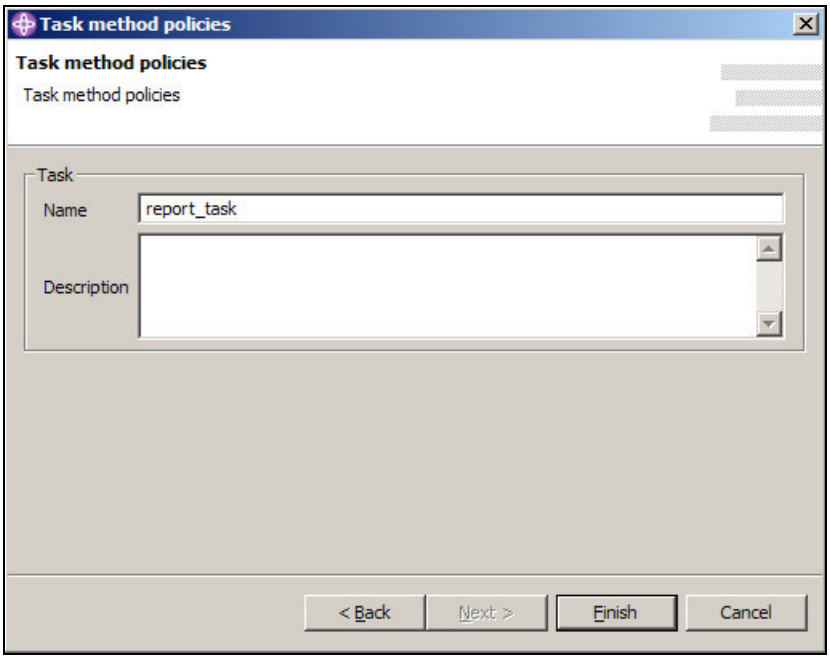


Figure 17-12 New Container-Managed Task 2.

b. Application-Managed Tasks shows on the upper right side of the page.

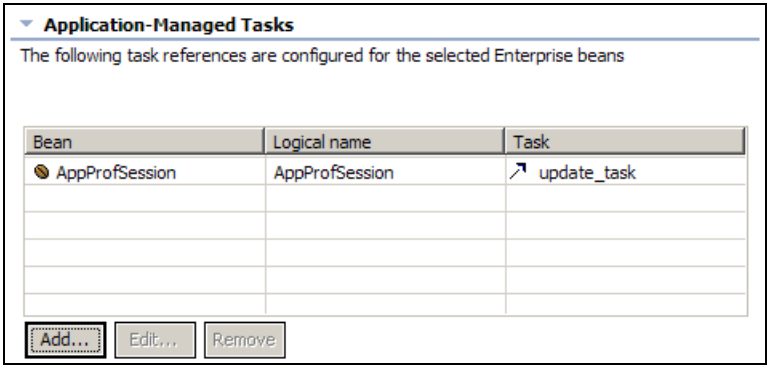


Figure 17-13 Application-Managed Tasks

To create a new task, click **Add...**

On the first panel select the bean, set the logical name for the task reference and the name for the task, at the end click **Finish**.

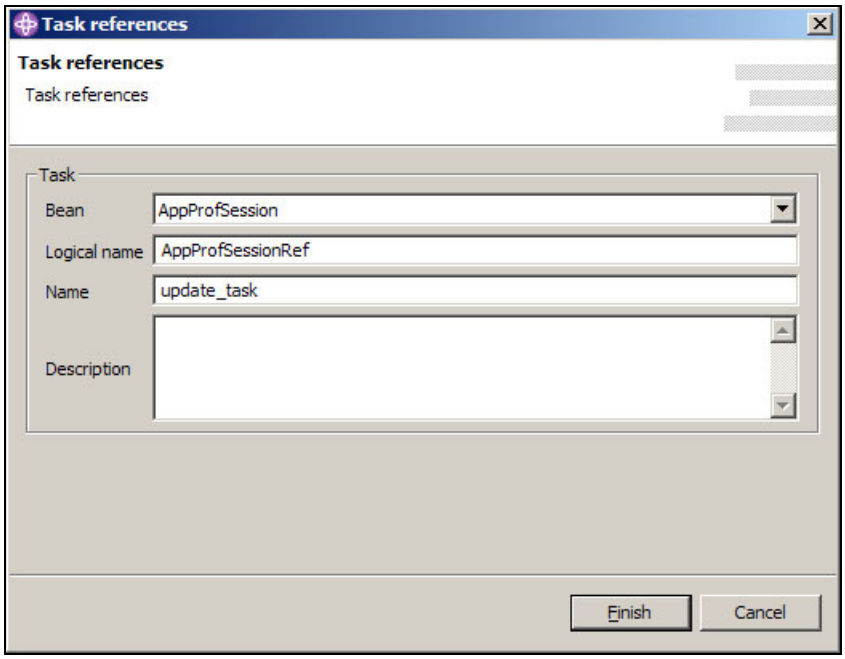


Figure 17-14 New Application-Managed Task 1.

4. The next step is to configure the tasks named in the previous step. The tasks are shown under the Application profiles section.

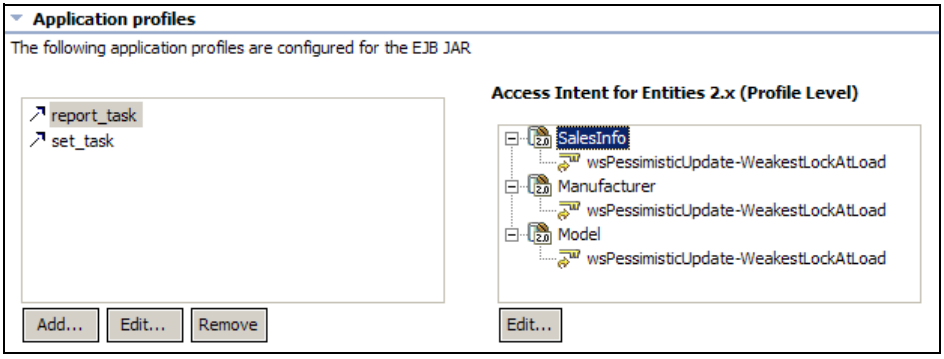


Figure 17-15 Application profiles

To define a new task, click **Add...**

a. First, set the name of the task, then click **Next**.

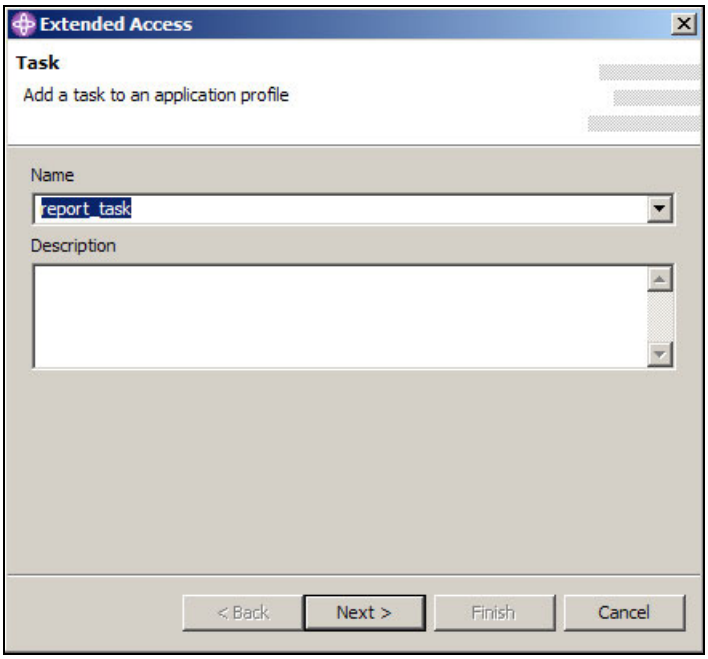


Figure 17-16 New application profile task 1

b. Select the managed beans for the task, then click **Finish**.

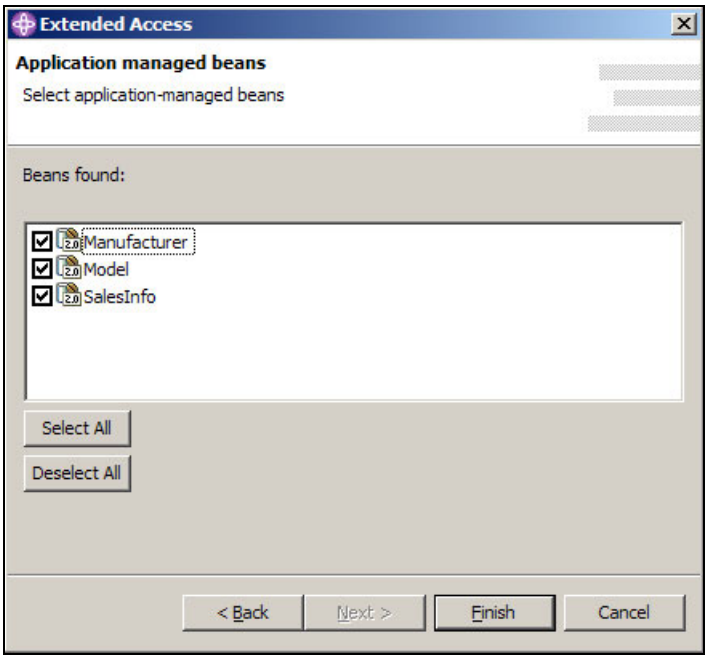


Figure 17-17 New application profile task 2

Once the task is set, you will find the Access Intent set for the entities under the *Access Intent for Entities 2.x (Profile Level)* section.

You can modify the access intent for a certain entity in the task.

a. Select the task under the *Application profiles* section, then select the entity bean under the *Access Intent for Entities 2.x (Profile Level)* section. Click **Edit...** underneath the beans.

b. On the panel, select the Access intent name, then click **Next**.

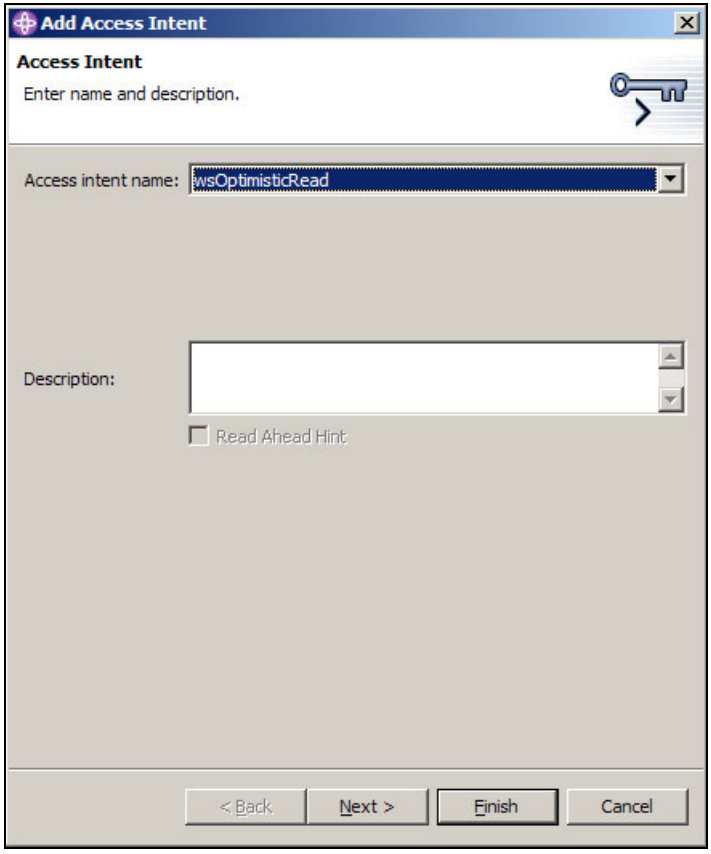


Figure 17-18 Configuring access intent for entity bean 1

c. Set further details for the access intent on the next panel. Check the box(es) that you want to configure then enter or select a value. Click **Finish** at the end.

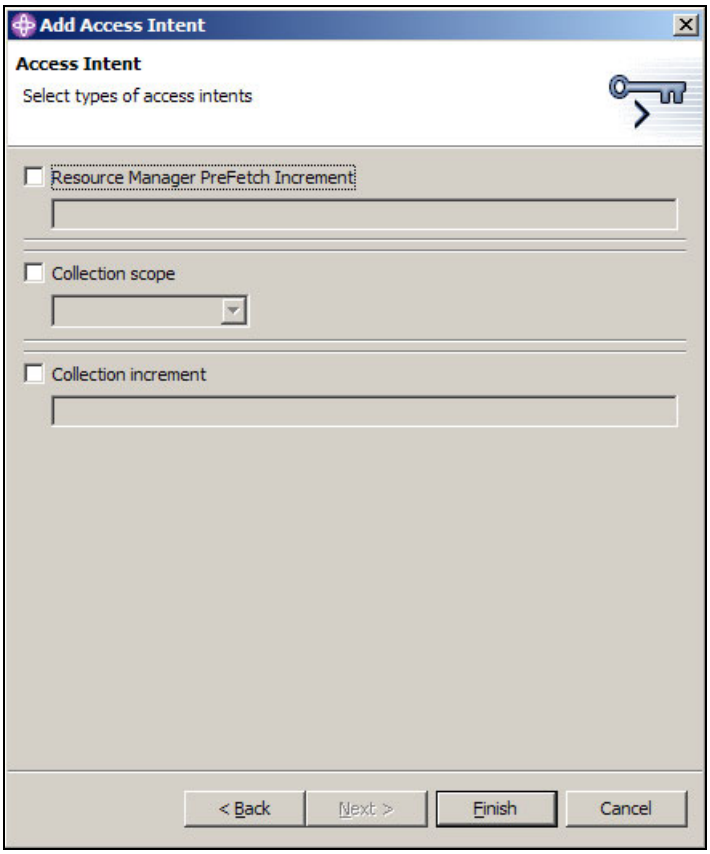


Figure 17-19 Configuring access intent for entity bean 2

d. You can repeat the steps for all the entity beans to set up or change the access intent.

5. If you need to define a custom access intent policy for your application, click Add under the *Defined Access Intent Policies* section, bottom of the page.

   Provide a name for the custom access intent, then set up the attributes for it.

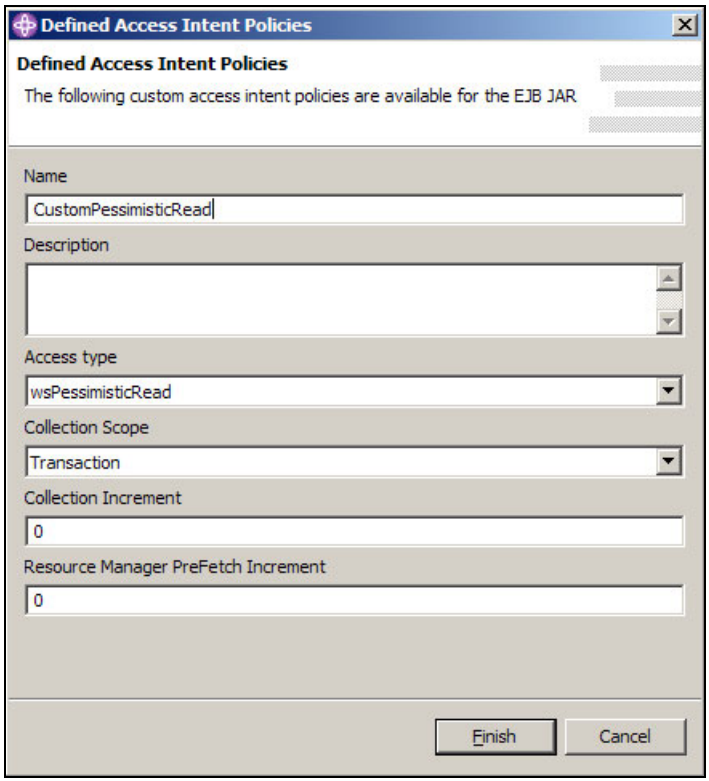


Figure 17-20 Custom defined access intent policy

After defining a new custom access intent policy, it is going to be available for selection for the entity beans at the *Access Intent for Entities 2.x (Profile Level)* section.

6. Save the EJB deployment descriptor and close the file.

18

Shared Work Area service

In this chapter, we will discuss the Shared Work Area. A Shared Work Area (SWA) is a service in WebSphere Business Integration Server Foundation. With this service, developers can easily pass user-defined information in a J2EE environment. The Work Area will be propagated automatically, like the security and transaction context.

18.1 Prerequisites

The Shared Work Area service has had some enhancement, since the WebSphere Application Server Enterprise V5 implementation, although the concept and the basics are the same.

For basic information about the Shared Work Area service, please refer to the redbook *WebSphere Application Server Enterprise V5 and Programming Model Extensions*, SG24-6932. This chapter will address some of the new features in the WebSphere Business Integration Server Foundation V5.1 release.

18.1.1 Work area partition service

The Work Area partition service is an extension of the Work Area service that allows users to create multiple customer Work Areas. WorkArea Partition can be thought of in the same way as the UserWorkArea and has the same API as the UserWorkArea. Any user that currently uses the Work Area service and the UserWorkArea partition can continue using it in the same manner. A WorkArea Partition differs from the UserWorkArea in a number of ways.

- The UserWorkArea is publicly available, through java:comp/websphere/UserWorkArea, thus allowing multiple users on the same thread to access the context in the WorkArea. A Work Area partition is created by a certain user and is only known to that user, unless of course that user makes its partition publicly known. The Work Area partition service does not strictly enforce that a partition is accessed by the partition creator. It can be accessed by anyone who knows the certain partition name. Users can choose to publish a certain partition name or not. On the other side, Work Area partition service will try to hide a partition as much as possible, actually, it does not allow a person to determine or query all the names of partition that have been created.
- A WorkArea Partition can be configured to allow for bidirectional propagation of its context. That is to say, changes made to a WorkArea's context by a downstream process will be propagated back upstream to the originator of that WorkArea.

The WorkAreaPartitionManager.createWorkAreaPartition() method can only be used from J2EE clients. Administrators can use the Administrative Console to create a Work Area partition on the server side. On the server side a Work Area partition must be created during server startup because each partition needs to be registered with the appropriate Web and EJB collaborators before the server is started. When creating the Work Area partition, there are some configuration properties available, which are defined as described in the following sections.

Bidirectional propagation of WorkArea context

In the previous manner, applications can create a Work Area, insert information in it, and perform a remote invocation. The Work Area is propagated with a subsequential remote method invocation. The server side methods which are invoked by client can use or simply ignore the information included in the Work Area as appropriate. If the method being called has a nested Work Area associated with it, a copy of the nested Work Area and all its ancestors are propagated to the target application.

There is a new configuration property *Bidirectional propagation of WorkArea context*, which can be specified when creating the Work Area partition service either in Administrative Console or using the API. If the bidirectional property is specified during creation time, the changes made by server side will be propagated back to calling application, meaning that changes made to the Work Area context by a downstream process will propagate back to upstream.

Note: UserWorkArea partition can not be configured to be bidirectional, it has been "hard configured" as non-bidirectional; therefore the changes only can be passed one way under UserWorkArea partition.

Deferred attribute serialization of WorkArea context

By default, the serialization and deserialization are automatically performed by the Work Area services. On each set of operations, the Work Area service serializes and deserializes the attributes. The mechanism gives the server complete control over the attribute so that any changes to a mutable object are not reflected in the Work Area's copy of the attribute unless a user specifically resets the attribute into the Work Area. However, from performance point of view, this will lead to excessive serialization/deserialization operation and can result in performance degradation under heavy load.

The *deferred attribute serialization* is a cache service to address this issue. With this property specified (using the Administrative Console or the API), attributes set into the Work Area service are not automatically serialized during the set operation. Rather, the Work Area stored a reference to the attribute as a cache. When a get operation is performed on an attribute, the reference to the object is returned to the requester and no deserialization is performed. So, when does the serialization/deserialization occur? When the deferred attribute serialization property is enabled, the Work Area service will always store the reference of mutable objects attributes except when:

- An attribute is reset or removed.
- The Work Area is explicitly completed by the application.
- Server component ends execution of the request during which the Work Area was begun.

- Client process which began the Work Area terminates.

Note: Be careful; if the changed attribute has not been reset in the Work Area, sequent invocation will not see the changed value. In other words, any direct changes to that mutable attribute are not propagated because the Work Area still holds an old serialized version of that attribute.

Partition context propagation across process boundaries

The Work Area service enables application developers to implicitly propagate information beyond the information passed in remote calls. Applications can create a Work Area, insert information into it, and make remote invocations. If a client has multiple different Work Area partitions when it makes remote invocation to server side, the information included in each partition on the client thread propagates to the application server. Which information (context) will be demarshalled at server side, depends on appropriate partition name both defined at client and server side. It means that only the context associated with a partition that is resident on both the client and server is demarshalled. The information (context) associated with a partition that does not reside on server side is still there but will not accessible, it will be propagated to a different server during the next remote invocation.

18.1.2 Distributed Work Areas

Following are some details about the distributed Work Areas.

Non-bidirectional Work Area partitions (UserWorkArea partition)

As with the previous UserWorkArea, if the calling application has a Work Area or nested Work Area, a copy of the Work Area and nested Work Area are automatically propagated to the remote method, which can modify or ignore the information in the Work Area as appropriate. The changed information then will be propagated to any other sequent method invocations. However, no changes made to a nested Work Area on a target object are propagated back to the calling application. The caller's Work Area is never affected by changes made in the remote method.

Bidirectional Work Area partitions

In WebSphere Application Server Enterprise V5, the Shared Work Area (SWA) had an important characteristic: the data can only passed one way. That means, if the remote method makes changes to the data, the changes will not be propagated back to the caller. However, if a partition is defined as bidirectional, and the target application does not need to begin a nested Work Area, any new

changes set into the Work Area will propagate not only subsequent remote invocations but also back to the originator applications. If the target application does not want the changed context to propagate back to the calling application, then the target application must begin a nested Work Area to scope the context to its process. The changed context in the nested Work Area will normally propagate to subsequent remote invocation if it has any.

18.2 Managing Work Area partitions

Work area service is managed in the Administrative Console. You have the following tasks available for the Work Area Partition service.

- Creating a new Work Area
- Enabling the Work Area service
- Managing the size of the Work Area

Creating a new Work Area

1. Start the Administrative Console and login.
2. Navigate to **Servers** → **Application Servers** → **<*server_name*>** → **Work Area Partition Service**.
3. Click **New** to create a new Work Area Partition.
4. Fill out the fields, the Name is mandatory, then click **OK**.

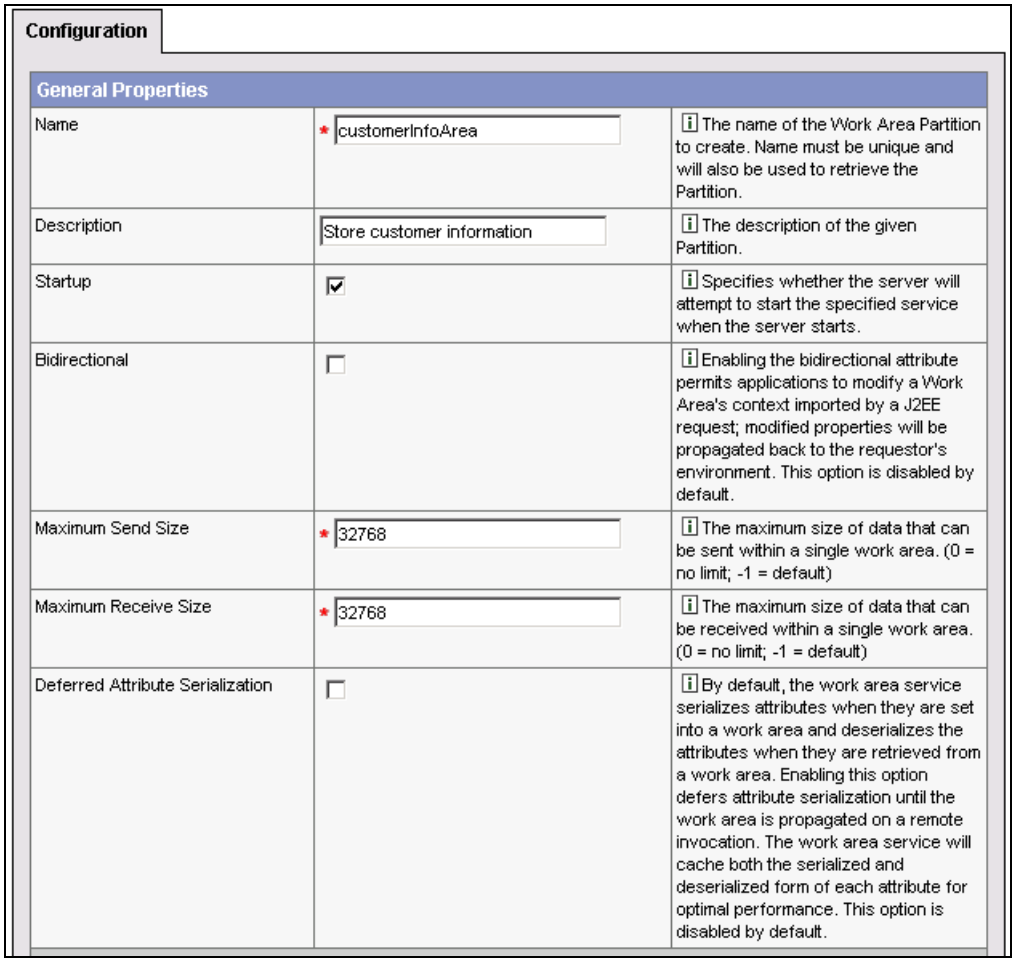


Figure 18-1 Shared Work Area service configuration for the server

5. Once the new item is created, save the configuration for WebSphere.

Enabling Work Area service

1. By default, the Work Area service is enabled on both clients and servers. But you still can choose this task to enable/disable the Work Area on a server.
   a. In the Administrative Console navigate to **Servers** → **Application Servers** → **<*server_name*>** → **Work Area Service**, see Figure 18-1.
   b. Select (deselect to stop) the **Startup** checkbox.
2. Enable/disable WorkArea service on a J2EE client.

Set the com.ibm.websphere.workarea.enabled property to true or false before starting the client. For example, you can add the following line to the launchClient.bat (Windows) or the launchClient.sh (Unix):

```
-Dcom.ibm.websphere.workarea.enabled=true|false
```

Managing the size of Work Area

Users can use this option to limit the data size sent/received both on the client and server side. Applications can set maximum sizes on each Work Area to be sent or received. By default, the maximum size of a Work Area that is sent by a client and received, then possibly re-sent, by a server is 32,768 bytes.

1. Set the sent/received data size of WorkArea on a server.

   a. In the Administrative Console navigate to the **Servers** → **Application Servers** → **<*server_name*>** → **Work Area Partition Service**, see Figure 18-1 on page 484.

   b. Enter a value for maxSendSize or maxReceiveSize to modify the size of the Work Area that this server can accept.

2. Set the sent/received data size of WorkArea on a J2EE client

   Set the com.ibm.websphere.workarea.maxSendSize property to the number of bytes before starting the client. For example, you can add the following line to launchClient.bat (Windows) or launchClient.sh (Unix):

   ```
   -Dcom.ibm.websphere.workarea.maxSendSize=1000
   ```

18.3 Sample scenario

This chapter is using the Travel Agency scenario introduced for the whole book. The sample application consists of a J2EE client named *TravelTestClient* and and EJB component on the server side. TravelTestClient creates two Work Area partitions: one is *customerInfoArea* which is then used to store customer information and the other is *travelInfoArea* which is then used to return some server side information back to client.

The application simulates a customer made travel reservation. The Customer inputs some information on the client, and passes the information to the server. On the server side a façade EJB gets the information, then makes the appropriate invocation to process the user's request, then it generates a confirmation number that is sent back to client.

Note: Only the original thread that makes the invocation can get the right information from WorkArea.

18.4 Development

For this chapter we have created a sample J2EE V1.3 application with all the code, we are not going to provide step-by-step instructions about how to build such an application. The description you can find here is how to use the Shared Work Area service in a J2EE application.

To develop an application using the Shared Work Area service, you need add two JAR files to your Java build directory: acwa.jar and distexcep.jar.

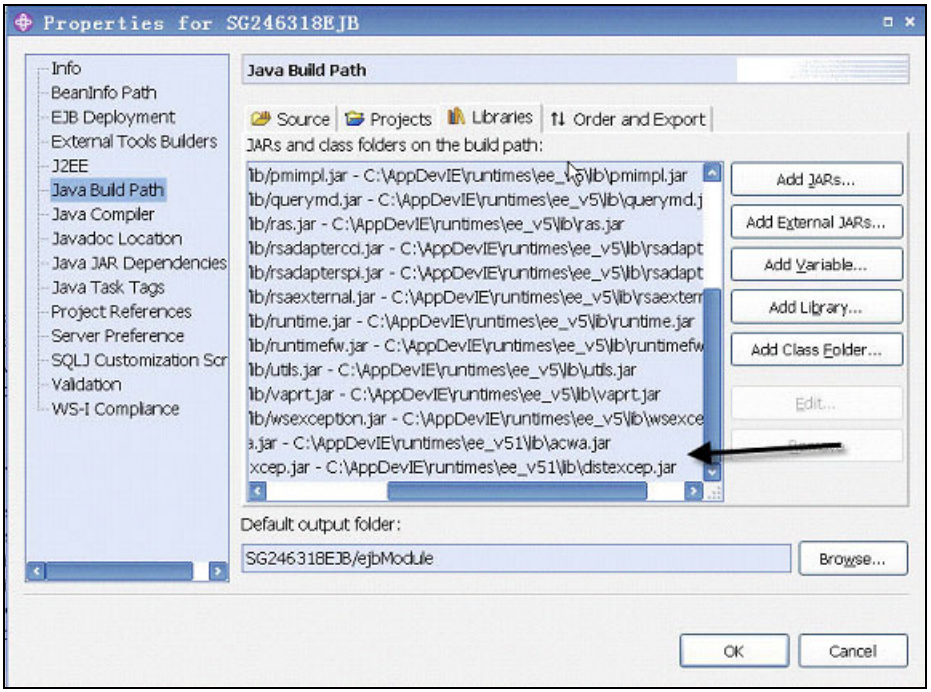


Figure 18-2 Java Build Path for the server application

The EJB module

The sample application has an EJB module and an application client module. The EJB module has two session EJBs: SessionFaçade, BusinessLogic.

The SessionFaçade is a façade component that routes the customer request to the appropriate business logic invocation. Only the SessionFaçade has remote interface, the subsequent logic beans only have local interface. We get the BusinessLogic local instance in ejbCreate() method to gain some performance benefit.

```
private BusinessLogicLocalHome home = null;
```

```
private BusinessLogicLocalHome home = null;
...
public void ejbCreate() throws javax.ejb.CreateException {
   try {
      InitialContext ctx = new InitialContext();
      // Get the BusinessLogic bean instance
      home = (BusinessLogicLocalHome) ctx.lookup(businessName);
      business = home.create();
   } catch (NamingException e) {
      System.out.println("SessionFacde Bean : Lookup the BusinessBean
failed!");
      e.printStackTrace();
   }
}
```

The actual business invocation will be delegated to BusinessLogic bean method.

```
// Business logic invocation
public void bookingProcess(){
   business.makeReservation();
}
```

There are two serializable classes: *CustomerInfo.java* and *TravelInfo.java*. These two classes will store the information exchanged between client and server side. In the TravelInfo.java, we defined a travelConfirmNumber parameter to store the confirmation information generated by the server. Because we will specify the travelInfoArea partition as "bidirectional", we assume this information will be propagated back to client thread automatically.

As usual, in the BusinessLogic bean, we get the partition service instance in ejbCreate() method to gain some performance benefit.

```
import com.ibm.websphere.workarea.NoSuchPartitionException;
import com.ibm.websphere.workarea.PropertyModeType;
import com.ibm.websphere.workarea.UserWorkArea;
import com.ibm.websphere.workarea.WorkAreaPartitionManager;
...
private static final String jndiName =
"java:comp/websphere/WorkAreaPartitionManager";
private WorkAreaPartitionManager manager = null;
private UserWorkArea custInfoArea = null;
private UserWorkArea travelInfoArea = null;
private CustomerInfo custInfo = null;
private TravelInfo travelInfo = null;
...
public void ejbCreate() throws javax.ejb.CreateException {
   try {
      InitialContext ctx = new InitialContext();
      manager = (WorkAreaPartitionManager) ctx.lookup(jndiName);
```

```
    } catch (Exception e) {
       e.printStackTrace();
    }
    try {
    custInfoArea = (UserWorkArea)
manager.getWorkAreaPartition("customerInfoArea");
    travelInfoArea = (UserWorkArea)
manager.getWorkAreaPartition("travelInfoArea");
    } catch (NoSuchPartitionException e) {
       e.printStackTrace();
    }
}
```

The WorkAreaPartitionManager instance looks up the WebSphere JNDI for the java:comp/websphere/WorkAreaPartitionManager. Then, we get the two Work Area partitions which are defined on the server using the WorkAreaPartitioniManager.getWorkAreaPartition("*partition_name*") method.

How the makeReservation() business method works:

1. Retrieve all the information: UserWorkArea.retrieveAllKeys() method will return all the informations included in the Work Area.
2. Make some changes to TravelInfo class, to simulate that we have some follow on invocation to generate the confirmation number for user travel booking.
3. Reset the property in the Work Area and try to propagate this information back to the client side. We use the UserWorkArea.set() method to achieve this.

Example 18-1 Propagating back to the client side

```
// The reservation business method
public void makeReservation() {
   System.out.println("BusinessLogicBean: Testing...");
   // Get information from CustomerInformation work area
   try {
      String[] custInfoProperties = custInfoArea.retrieveAllKeys();
      if (custInfoProperties != null) {
         for (int i = 0; i < custInfoProperties.length; i++) {
            // … print out information here
         }
      } else {
         System.out.println("BusinessLogicBean: No Properties found!...");
      }

      // You can make some other invocations like WPC to process the user
      // request based on the user information
      // Then, Store the confirm information in the bidirectional work area
```

```
        // Get information from TravelInformation work area
        String[] travelInfoProperties = travelInfoArea.retrieveAllKeys();
        if (travelInfoProperties != null) {
           for (int i = 0; i < travelInfoProperties.length; i++) {
           //print out information here
           }
     // We store the confirmation number in the work area so that it can be
     // propagated back to caller this number is for test purpose,
        // meaning it can be some other return values on other invocation
           travelInfo.setTravelConfirmNumber("1234567890");
           travelInfoArea.set("travel", travelInfo, PropertyModeType.normal);
        }
        } else {
           System.out.println("BusinessLogicBean: No Properties found!...");
        }
     } catch (Exception e) {
     }
  }
```

The application client

This section will discuss how the TravelTestClient application works.

1. To get the Work Area partition manager and SessionFacade session instances the client uses the JNDI lookup to find the two services: java:comp/websphere/WorkAreaPartitionManager and java:comp/env/ejb/SessionFacade.
2. Once got the Work Area manager instance, we are going to create two Work Area partition at client side because only the context associated with a partition that is resident on both the client and server is demarshalled. See below code snippet:

   ```
   //Find the work area partition manager
   Properties props = new Properties();
   props.put("maxSendSize", "32768");
   props.put("maxReceiveSize ", "32768");
   props.put("Bidirectional ", "true");
   partition = manager.createWorkAreaPartition("partition_name", props);
   ```

3. Get the information that was changed in the server.

   If the Work Area partition was defined as bidirectional, the changed information will be propagated back to client thread automatically. We use UserWorkArea.retrieveAllKeys() to get the information back.

   ```
   String[] travelInfoProperties = travelInfoArea.retrieveAllKeys();
   ```

The following code example is a snippet from the TravelTestClient.java source, for more information read the comments inserted into the source.

Example 18-2 TravelTestClient.java

```
//...
UserWorkArea custInfoArea = null;
UserWorkArea travelInfoArea = null;
WorkAreaPartitionManager manager = null;

SessionFacade facade = null;
final String facade_jndiName = "java:comp/env/ejb/SessionFacade";
final String workAreaPartitionManager_jndiName =
"java:comp/websphere/WorkAreaPartitionManager";

// Prepare some parameters for test, leave the travelConfirmationNumber blank for test.
CustomerInfo custInfo = new CustomerInfo("SAM", "Male", "12345678");
TravelInfo travelInfo = new TravelInfo(new Date(System.currentTimeMillis()), "LA", true, true,
true, "");

// Find the work area partition manager instance
try {
   InitialContext ctx = new InitialContext();
   manager = (WorkAreaPartitionManager) ctx.lookup(workAreaPartitioinManager_jndiName);
   SessionFacadeHome home = (SessionFacadeHome)
PortableRemoteObject.narrow(ctx.lookup(facade_jndiName), SessionFacadeHome.class);
   facade = home.create();
} catch (Exception e) {
   e.printStackTrace();
   System.exit(1);
}

// Create the work area partition on the client side
Properties props = new Properties();
props.put("maxSendSize", "32768");
props.put("maxReceiveSize", "32768");
props.put("Bidirectional", "false");
try {
   custInfoArea = manager.createWorkAreaPartition("customerInfoArea", props);
   // Let travelInfoArea partitioin bidirectional
   props.setProperty("Bidirectional", "true");
   travelInfoArea = manager.createWorkAreaPartition("travelInfoArea", props);
} catch (PartitionAlreadyExistsException e) {
   e.printStackTrace();
} catch (IllegalAccessException e) {
   e.printStackTrace();
}

// Set the information in the work area
try {
   // Begin a non-directional work area to store customer information
   custInfoArea.begin("CustomerInformation");
```

```
   System.out.println("TravelTestClient: Work Area " + custInfoArea.getName() + "Successfully
Created!");
   custInfoArea.set("user", custInfo, PropertyModeType.normal);
   //Begin a bidirectional work area to store customer information
   travelInfoArea.begin("TravelInformation");
   travelInfoArea.set("travel", travelInfo, PropertyModeType.normal);
} catch (Exception e) {
   e.printStackTrace();
}

// Test this information with the server
try {
   facade.bookingProcess();
} catch (Exception e) {
   e.printStackTrace();
}

// Get the information from the TravelInformation work area for the confirmation number.
// Since this work area is defined as bidirectional
try {
   String[] travelInfoProperties = travelInfoArea.retrieveAllKeys();
   if (travelInfoProperties != null) {
      for (int i = 0; i < travelInfoProperties.length; i++) {
         System.out.println("TravelTestClient: found <" + travelInfoProperties[i] + "> in
WorkArea: " + travelInfoArea.getName());
         travelInfo = (TravelInfo) travelInfoArea.get(travelInfoProperties[i]);
         System.out.println("TravelTestClient: the detail travel information is: " +
travelInfo.getTravelDate().toString());
         System.out.println("TravelTestClient: the detail travel information is: " +
travelInfo.getTravelDestination());
         System.out.println("TravelTestClient: the detail travel information is: " +
travelInfo.getTravelConfirmNumber());
      }
   } else {
      System.out.println("BusinessLogicBean: No Properties found!...");
   }
} catch (Exception e) {
   e.printStackTrace();
}
//...
```

18.5 Testing

You can easily test the Shared Work Area service using the sample application provided with this book.

You can import the application to WebSphere Studio Application Developer Integration Edition and run it in the WebSphere Test Environment.

> **Note:** Make sure you add the acwa.jar and the distexcep.jar to both the EJB and Application client module.

You can also deploy the application on the WebSphere Business Integration Server Foundation.

In both cases, you have to use the Administrative Console to create the following Shared Work Area items for the server.

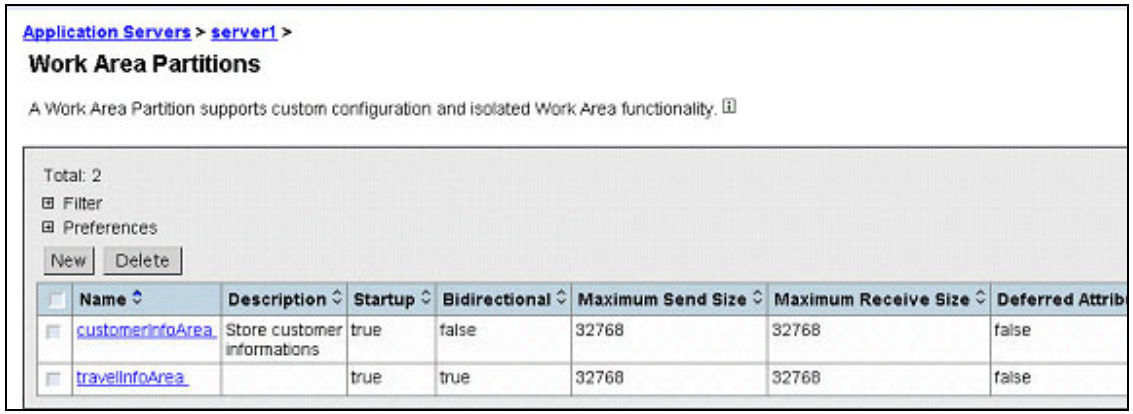


Figure 18-3 Shared Work Area items

You have to restart the server to have the Shared Work Area service running with the new items. You should see the following lines in the SystemOut.log file:

```
WorkAreaParti A ACWA0017I: WorkArea Partition: customerInfoArea, is ready
on server1
WorkAreaParti A ACWA0017I: WorkArea Partition: travelInfoArea, is ready on
server1
```

For testing, you will need to run the application client from a console window, using the same .ear file with the launchclient.bat (launchclient.sh on Unix).

After a successful testing, you should see the following results in the server SystemOut.log file:

```
BusinessLogicBean: Testing.....................................
BusinessLogicBean: found <user> in WorkArea: CustomerInformation
BusinessLogicBean: the detail user information is: SAM
BusinessLogicBean: the detail user information is: Male
BusinessLogicBean: the detail user information is: 12345678
```

```
BusinessLogicBean: found <travel> in WorkArea: TravelInformation
BusinessLogicBean: the detail travel information is: Wed May 12 00:13:46
EDT 2004
BusinessLogicBean: the detail travel information is: LA
```

The client should return the following results:

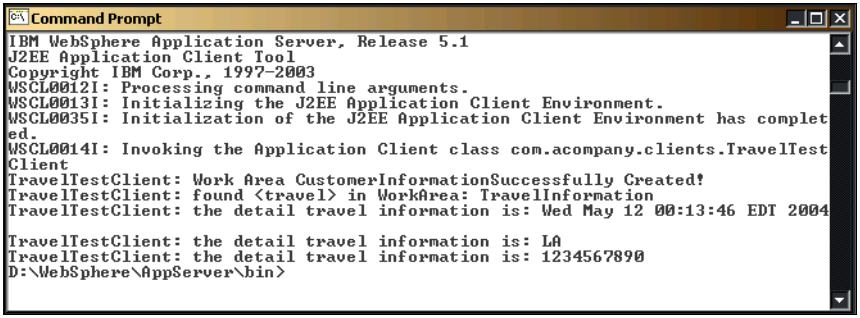


Figure 18-4 Client application log

Tracing

In order to enable tracing for the Shared Work Area service, enable the WebSphere system component trace using the following expression:

```
com.ibm.ws.workarea.*=all=enabled
```

19

Dynamic Query

Chapter 19

The Dynamic Query service is provided by WebSphere Business Integration Server Foundation in the area of data access through CMP EJBs. The base application server supports the standard EJB QL specifications, with some extensions that increase the flexibility of the syntax of EJB QL. However, one of the characteristics of EJB QL is its static nature. Once an EJB QL query is associated with a finder method and the EJB deployed, the query can only be modified by redeploying and re-installing the EJB. The Dynamic Query service removes this limitation. It allows applications to formulate queries at runtime and have them submitted and executed on the fly, providing a behavior that resembles that of traditional dynamic SQL.

The Dynamic Query service works at the object schema level, just like regular EJB QL. This is a form of object query, and not a database query. It works for CMP Entity EJBs, and provides the ability to perform queries on both CMP and CMR fields. Hence dynamic EJB query is largely independent of the way objects are schema mapped to the relational database tables.

19.1 Prerequisites

WebSphere Business Integration Server Foundation InfoCenter provides valuable information regarding various development, deployment and administration aspects. InfoCenter is located at:

http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp

Familiarize yourself with the following topics in the WebSphere Business Integration Server Foundation InfoCenter:

- Installing WebSphere Business Integration Server Foundation:

  http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/info/ee/ae/welc_installing.html

- Using Dynamic Query service:

  http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/info/ee/query/tasks/tque_dynamic.html

- There is additional information available on Dynamic Query Service development practices in the InfoCenter. Using WebSphere Business Integration Server Foundation InfoCenter, navigate to **All topics by feature** → **Applications** → **Application services** → **Using EJB query**.

- It is important to gain familiarity with the EJB query language. The next few sections assume this knowledge and refer to the InfoCenter where possible. In the WebSphere Business Integration Server Foundation InfoCenter, navigate to **Developing** → **Task overviews** → **Using EJB query**.

  For EJB Query language specifics, click **Developing** → **Task overviews** → **Using EJB query** → **EJB query language**.

- Familiarize yourself with Dynamic Query API, because references to various functions will be made in the Development section below. In the WebSphere Business Integration Server Foundation InfoCenter, navigate to **Reference** → **Javadoc** → **Enterprise Extensions API**. Then click **com.ibm.websphere.ejbquery**.

Familiarize yourself with the following topics in WebSphere Studio Application Developer Integration Edition help:

- EJB Deployment tool. This can be navigated by going to WebSphere Studio Application Developer Integration Edition Help Content and then navigating to **WebSphere Studio** → **Developing** → **EJB applications** → **EJB application development overview** → **Tools for developing EJB applications** → **EJB deployment tool**.

- The Dynamic Query Cheat Sheet can be perused by going to WebSphere Studio Application Developer Integration Edition Help Content and then

navigating to **WebSphere Studio** → **Developing** → **Enterprise services** → **Tools for building enterprise services** → **Cheat sheets**.

The sample application used in this chapter assumes the knowledge of DB2 UDB 8.1. It is further assumed that the user's development environment has DB2 UDB 8.1 installed. Sample Dynamic Query application contains database scripts that are DB2 specific. Furthermore, setup instruction for the development environment can be found in 19.3.4, "Development environment setup" on page 504.

19.2 Sample scenario

In order to illustrate the usage of Dynamic Query service in WebSphere Business Integration Server Foundation, a sample Web application is developed. This Web application simulates ordering subsystem of an online e-commerce Web site. The application contains several Entity EJBs that map to back-end database tables. Additionally, the application consists of Session EJBs, servlets and HTML forms. The front-end of the application was modified for illustrating the Dynamic Query service. Figure 19-1 highlights the application's architecture and highlights the interactions between application components.

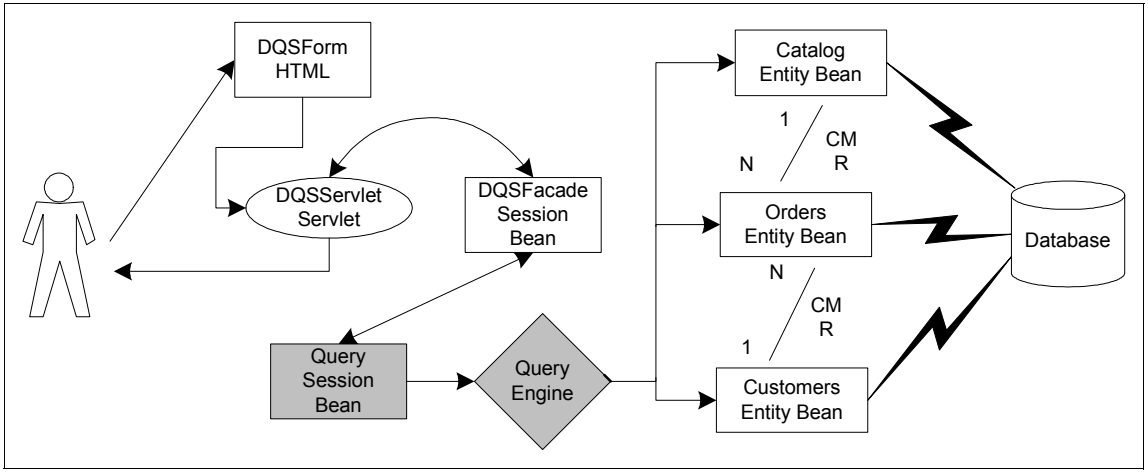


Figure 19-1 Dynamic Query Service: Sample Application Architecture

Using this Web application, users can execute pre-defined Dynamic Queries or compose Dynamic queries on the fly and execute them. This sample application will illustrate the following key extensions to the base EJB 2.0 EJB QL language:

- How to use multiple element in the SELECT clause.
- How to use WebSphere extension aggregate functions.

- How to invoke business method on EJB that are activated as part of executing the query.

Migration

There are no migration tasks for the Dynamic Query service in WebSphere Business Integration Server Foundation V5.1.

Development

The effort involved in developing code for Dynamic Query service is as follows:

- Properly formulating Dynamic Query and have a better understanding of what data is required for business purposes.
- Instantiate Query Session bean's and invoke one of the remote interface method of Query Session bean.
- Exception handling as Query Session bean throws Dynamic Query specific exceptions that should be caught by the application.

All components of the sample application, illustrated in Figure 19-1 on page 497, are discussed in more detail in "Development" on page 498.

Unit test

Application containing Dynamic Query service code can be tests within the WebSphere Test Environment in WebSphere Studio Application Developer Integration Edition v5.1. For details on WebSphere Test Environment refer to 19.4, "Unit test" on page 514.

Assembly

It is important to note that Dynamic Query service provides both local and remote interfaces to the Query Session Bean - the core of Dynamic Query service. When using the local interface, the local JNDI name for the QueryBean must be specified; and for remote interface, use the remote JNDI name.

Additionally, the default JNDI name for the query bean can be different for development and production environment. Application Server Toolkit can be used to specify the right JNDI name.

Deployment

When it comes to packaging, deployment and usage of application containing Dynamic Query service code, it should be treated as standard J2EE application. There are no specific steps related to the deployment of application containing Dynamic Query service code in the WebSphere Business Integration Server Foundation runtime.

In WebSphere Business Integration Server Foundation, Dynamic Query Services is provided as a special stateless session bean, named QueryBean. The QueryBean is packed in `query.ear` file that is automatically installed on the default server during the WebSphere Business Integration Server Foundation installation. If new application servers are created, the `query.ear` file must be manually installed on each additional application server in the same fashion any other application is installed.

Testing

Dynamic Query service code can be tested in the unit test environment. However, for functional and integration tests, query components should be further tested in the runtime environment.

You can install the Universal Test Client in the runtime environment to perform testing.

19.3 Development

The following section describes how to make use of the Dynamic Query Service in a J2EE application. The sample application used here is described in section 19.2, “Sample scenario” on page 497. First a general discussion of Dynamic Query API is given in this section. The development environment setup for Dynamic Query is discussed next followed by a discussion of sample application development.

For development purposes, using WebSphere Studio Application Developer Integration Edition, `query.ear` must be imported into the Project where QueryBean is used. Client interfaces to the QueryBean, must also be imported. Refer to section 19.3.4, “Development environment setup” on page 504 for more details.

Table 19-1 shows the JAR files that comprise the Dynamic Query service.

Table 19-1 Dynamic Query service JAR files

JAR File	Usage
query.jar	Query parser, runtime, Query Session bean
qryclient.jar	Client stubs and classes
query.ear	Query Session Bean application (located in installableApps, it should be installed in application server runtime environment)

When QueryBean is installed in application server, or imported into development environment, the default JNDI name for the QueryBean is `com/ibm/websphere/ejbquery/Query`. In the runtime environment, this name can be changed by system administrators.

Tip: Applications invoking the QueryBean can either used the full JNDI name, given above, or use EJB reference. Using the EJB reference, there are no code changes required if the default QueryBean JNDI name is changed by System Administrator at deployment time. Application deployer can simply update the QueryBean EJB reference in deployment descriptor. The use of QueryBean EJB reference is recommended.

Sample Dynamic Query application makes use of EJB reference.

19.3.1 Dynamic Query service

The Dynamic Query service is implemented using “push down” technology. The technology involves taking an object query statement and using metadata that describes the mappings of EJB attributes to database tables and columns, to translate it to an SQL statement that can be executed by the database management system. In effect, the EJB query is pushed down to the database, which is designed to perform this type of query evaluation, and groups and sorts very efficiently. As would be expected, not all of an EJB query can be pushed down. The application server must still evaluate some query criteria, such as results of method invocations. By delegating most of the work, including navigation of relationships, to the database, excellent performance can be provided. Lastly, this push down technology is designed to be independent of the database vendor. It works with any relational database supported by WebSphere.

The Dynamic Query service is provided by the stateless Query session bean. The client of the Query session bean may be a remote client or it may be a local client, depending on whether the client makes use of the bean’s remote or local interfaces. A remote client accesses the query bean through the bean’s remote interface, and remote home interface. A remote client can be any Java program such as an application, JSP, applet, or servlet. A local client accesses the query bean through the bean’s local interface and local home interface. A local client is collocated in the same JVM with the query bean and can be another enterprise bean such as a session bean, entity bean or Message-Driven bean, as shown in Figure 19-2 on page 501. The Query Engine can perform queries on entity beans, on CMP fields, and on CMR fields, represented by Entity EJB X and Entity EJB Y in the figure.

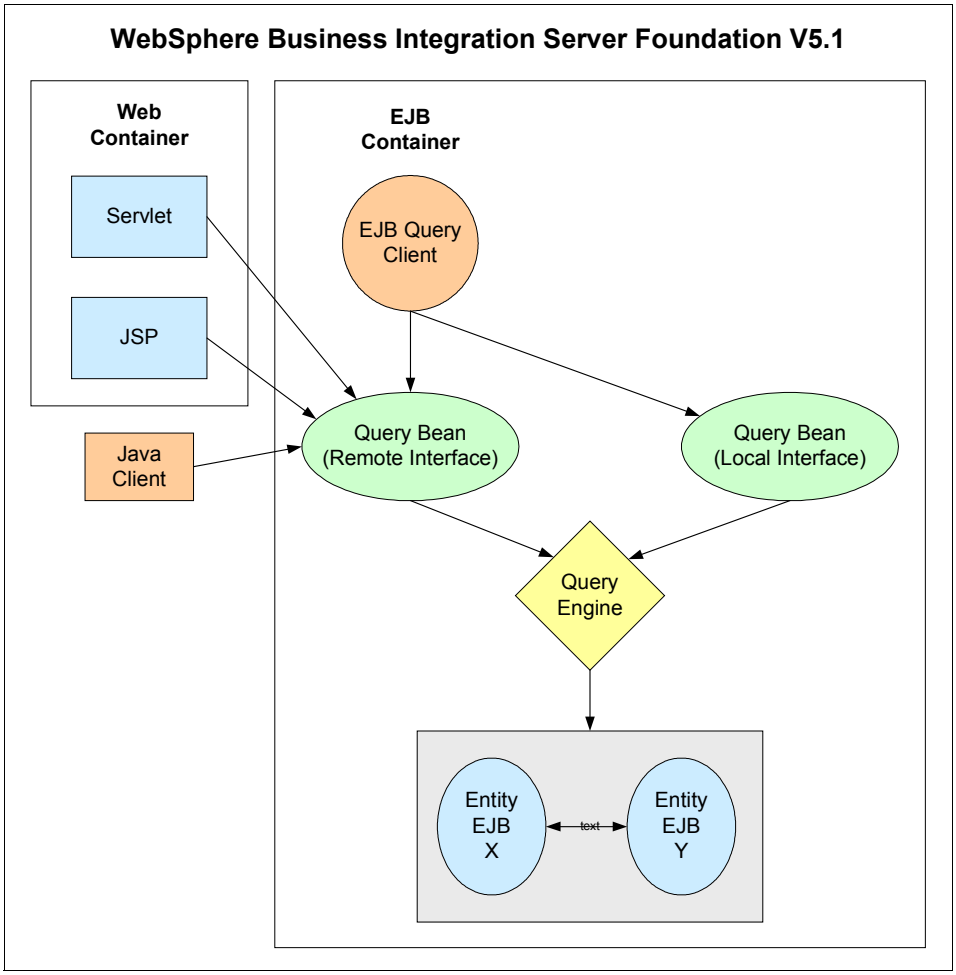


Figure 19-2 Dynamic Query service

19.3.2 Design concerns and recommendations

There are two major concerns to explore before deciding to use Dynamic Queries with your enterprise applications. One of these concerns is related to performance, while the other is related to security.

- If you have a query that has a high frequency of execution, you should define it as a finder or select method and consider using SQLJ as a deployment option for best performance. The Dynamic Query service always uses JDBC and must parse and process the EJB query at runtime.

- If you need security control over which queries a user can execute, you need to define the queries as finder or select methods and use EJB method authorization. The Dynamic Query service does not have fine-grain security control at this time. You can control who is permitted access to the remote query bean and the local query bean, but once authorized a user can execute any valid query and return any data in the server.

19.3.3 Dynamic Query Bean API

The query bean has both a remote and a local interface to support both remote and local clients.

The QueryBean interface has three client methods:

- executeQuery(): this method parses and executes the query in a single operation. The executeQuery method in the remote interface has some extra arguments over the method provided by the local interface.
- prepareQuery(): this method invokes the query parsing and plan creation process. It receives exactly the same three common input parameters described above (queryStatement, parameterVars, and queryDomain). This method returns the optimized query plan generated from the query string input parameter. Normally a client would not directly invoke the prepareQuery or executePlan methods. The return type of this method is a string containing the created query plan, which is a query statement parsed, validated, and optimized, and it is presented as an input to the executePlan method.

  Note: There might be times when the same query is being executed multiple times. Therefore, some performance optimization can be received by invoking prepareQuery once, and then invoking executePlan multiple times, rather than invoking executeQuery multiple times. Doing this helps save parsing effort.

- executePlan(): this method executes a query plan that is in a string text form. It receives two input parameters for the local interface, and an additional two for the remote interface.

For API details on QueryBean interface, refer to WebSphere Business Integration Server Foundation v5.1 InfoCenter:
http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/info/ee/javadoc/ee/com/ibm/websphere/ejbquery/package-summary.html

Local client programming model

For a local client to use the Dynamic Query service, the client implementation should perform the following steps:

1. Look up the QueryLocalHome.
2. Populate the parameter list.
3. Create an instance of the Query bean.
4. Formulate the query string.
5. Run the query.
6. Iterate through the result set. Each iteration will retrieve a tuple. For each tuple, the client should get the name of the field and its corresponding value. If one of the values is an EJB reference, then that will be an EJB local reference. You just need to cast it to the correct type using normal Java casting. It has to be narrowed to the correct type using java.rmi.PortableRemoteObject.

The usage of Dynamic Query local client interface is given in WebSphere Business Integration Server Foundation v5.1 InfoCenter. Refer to the following link for details:
http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/info/ee/query/xmp/xque_localclient

Note: When using a local client with Dynamic Query, the query must be invoked from within a transaction scope. At transaction termination, local query iterator is invalidated.

Remote client programming model

For a remote client to use the Dynamic Query service, the client implementation should perform the following:

1. Look up the QueryBean home.
2. Populate the parameter list.
3. Create an instance of the Query bean.
4. Formulate the query string.
5. Specify the desired number of rows to be retrieved.
6. Run the query.
7. Iterate through the result set. Each iteration will retrieve a tuple. For each tuple, the client should get the name of the field and its corresponding value. If one of the values is an EJB reference, that it has to be narrowed to the correct type using java.rmi.PortableRemoteObject.

The usage of Dynamic Query remote client interface is given in WebSphere Business Integration Server Foundation V5.1 InfoCenter. Refer to the following link for details:

http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.wasee.doc/info/ee/query/xmp/xque_remoteclient.html

Important: When an object is selected by the query, the return value is an EJB local reference, which needs casting to the correct type using normal Java casting, when using a local client. When using a remote client, the EJB reference has to be narrowed to the correct type using java.rmi.PortableRemoteObject.

19.3.4 Development environment setup

When developing applications that use the Dynamic Query service, you need to start by setting up WebSphere Studio Application Developer Integration Edition to develop and run applications using dynamic queries.

Sample Application Workspace setup

One of the first step that should be done when developing Dynamic Query applications, like other J2EE applications, is to set up the WebSphere Studio Application Developer Integration Edition workspace.

The following instructions can be used to set up the Studio workspace for the sample Dynamic Query application.

1. Launch WebSphere Studio Application Developer Integration Edition with a new workspace, make sure you enable server targeting for the workspace.
2. Import `ACompanyDQS.ear` file into workspace. Make sure to not check **Preserve project names, classpath....** checkbox; check ACompanyUtility.jar module, and to select **Expanded: extract project contents for development** radio button. Make sure that the target server is set to **Integration Server V5.1.**

   Click **Finish** button to import the EAR file.
3. Switch to the J2EE perspective, open the EJB Deployment Descriptor for the ACompanyEJB module. Verify that the JNDI name for the datasource is set to *jdbc/redbookDS* and Container authorization type is *Per_Connection_Factory*. The Backend ID should be set to *DB2UDBNT_V8_1*.

Dynamic Query dependencies

It is worth noting the most important tasks to set up development environment for Dynamic Query Service is the inclusion of query.jar and qryclient.jar in WebSphere Studio Application Developer Integration Edition project.

In order to define EJB reference to QueryBean remote interface in DQSFacade session bean, the query.ear must be imported into the workspace. Import the `query.ear` as an EAR project located in the <WSADIE_root>/runtimes/ee_v51/installableApps/ folder.

Note: Note that the query.ear and the query.jar are not the same. The first is an enterprise application to support the Dynamic Query service, the latter holds the API for programming.

EJB References

Sample Dynamic Query application includes EJB references in both DQSFacade session bean and DQSServlet. Perform the following steps to make sure that EJB references are defined properly

1. Switch to Business Integration perspective and to J2EE Hierarchy view.
2. Expand **Web Modules** → **ACompanyWeb** and verify that **EjbRef DQSFacade** exists. Refer to Figure 19-3
3. Expand **EJB Modules** → **ACompanyEJB** → **Session Bean** and verify that **EjbRef ejb/Query** reference exists, refer to Figure 19-3

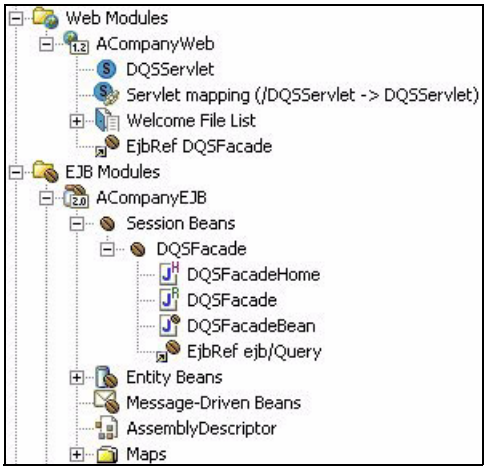


Figure 19-3 EJB References

Sample application database setup

The sample Dynamic Query application, ACompanyDQS Enterprise App, contains CMP Entity Beans. These Entity beans require a persistent storage to map to. The following steps will require IBM DB2 database.

1. Open a DB2 Command Window, change directory to the <DQSample_root>\ACompanyEJB\ejbModule\META-INF\database.
2. Open the setup.bat file in a text editor and replace the CHANGE_USERNAME (two occurrences) and the CHANGE_PASSWORD (one occurrence) with your database user name end password.
3. Issue the **`setup.bat`** command from the DB2 Command Window.
4. Make sure that the script ran successfully, at the end of each executed command you should find the results.

The script also creates tables for the database and populates them with data. At this point the REDBOOK database is ready for the sample application.

19.3.5 Development of Dynamic Query sample

Dynamic Query sample application, described in section 19.2, “Sample scenario” on page 497 and illustrated in Figure 19-1 on page 497, consists of Enterprise Beans, both Entity and Session beans to serve as back-end business logic, a servlet that controls the data flow between the back-end business logic tier and the user interface, and HTML form that servers as the user interface for interacting with the back-end business tier.

Note: Note that the use of JSP is recommended for User Interface design and tier. Usage of JSPs is a best practices for MVC-II design as it relates to Web application. Moreover, servlets should be used to control data flow and should contain minimal logic for data manipulation or for user interface presentation.

The sample application does not use JSPs for User Interface tier. The sample application is only meant for Dynamic Query usage illustration purposes only.

The application contains the following J2EE components:

- Catalog Entity Bean
- Orders Entity Bean
- Customers Entity Bean
- DQSFacade Session Bean (Stateless)
- DQSServlet servlet

This section does not concentrate on the development of Entity Beans. Entity Beans included in sample application have been previously developed and are

only used in this sample application for illustration purposes. However, the section discusses how to invoke Dynamic Queries on the Entity Beans. For this purpose, a stateless Session Bean is developed, along with a servlet and HTML form.

DQSFacade Session Bean

The session bean is used to invoke QueryBean using pre-defined and custom Dynamic Queries, which in turn invoke appropriate Entity Beans. This Session bean includes two methods that are part of the remote interface

- **executeDQ(string)** invokes pre-defined Dynamic Queries based on method argument passed. This method first performs preliminary checks on the input arguments and throws appropriate exceptions. Then, depending on the input argument, the method invoke one of the three private methods, `executeQuery1()`, `executeQuery2()` or `executeQuery3()`. These methods specifically call the three different types of Dynamic Queries by invoking QueryBean's `executeQuery() method`.
- **executeDQ(string, string)** invokes free-form Dynamic Queries. This method has two arguments. First step in this method, is to perform checks on the input arguments; the method then invokes a private method `executeFreeFormQuery()` that invokes QueryBean's `executeQuery()` method.

Example 19-1 shows DQSFacade Bean code. Take a note of executeQuery1() method below to see how the initial lookup is performed on QueryBean's home interface; EJB reference is used instead of QueryBean's default JNDI name. Note that for performance reasons, QueryHome is defined as a static variable such that it is instantiated only once, hence performing the JNDI lookup once.

Note: For space constraint, the entire code for DQSFacadeBean.java is not shown in Example 19-1 on page 507. Refer to the Dynamic Query sample application code that ships with redbook for details.

Example 19-1 DQSFacadeBean Java code

```
//...
public class DQSFacadeBean implements javax.ejb.SessionBean {
    private static QueryHome qHome;
    private String dqsQuery = null;
    private javax.ejb.SessionContext mySessionCtx;
//...
    public Collection executeDQ(String queryType) throws DQSException {
        if (queryType == null)
            throw new DQSException("DQSFacadeBean.executeDQ(String): Invalid
Querty Type Argument, queryType.  Check JSP/HTML Form.");
```

```
        System.out.println("DQSFacadeBean.executeDQ(): queryType is '" +
queryType + "'");
        if (queryType.equals("q1")) { return executeQuery1(); }
        else if (queryType.equals("q2")) { return executeQuery2(); }
        else if (queryType.equals("q3")) { return executeQuery3(); }
        // Default Query type is QUERY_TYPE1
        return executeQuery1();
    }

    public Collection executeDQ(String queryType, String queryInput) throws
DQSException {
        if (queryType == null)
            throw new DQSException("DQSFacadeBean.executeDQ(String, String):
Invalid Querty Type Argument, queryType.  Check JSP/HTML Form.");
        if (queryInput == null)
            throw new DQSException("DQSFacadeBean.executeDQ(String, String):
Invalid Querty Type Argument, queryInput. Check JSP/HTML Form.");
        System.out.println("DQSFacadeBean.executeDQ(): queryType is '" +
queryType + "'");
        return executeFreeFormQuery(queryInput);
    }
    private Collection executeQuery1() throws DQSException {
        try{
            dqsQuery = DQSConstants.QUERY_TYPE1;
            System.out.println("DQSFacadeBean.executeQuery1(): Executing " +
dqsQuery);
            if (qHome == null){
                InitialContext ic = new InitialContext();
                qHome = (QueryHome) ic.lookup("java:comp/env/ejb/Query");
            }
            Query query = qHome.create();
            //Execute QUERY_TYPE1 Dynamic Query:
            //SELECT cat.cname, cat.cprice FROM Catalog cat WHERE cat.cid = '1'
            Object[] params = { new String("1") };
            QueryIterator it = query.executeQuery(dqsQuery, params, null, 0,
50);
            ArrayList arrl = new ArrayList();
            int rows = 0;
            while (it.hasNext()) {
                ArrayList ar = new ArrayList();
                System.out.println("DQSFacadeBean.executeQuery1(): Number of
fields: "+ it.getFieldsCount() );
                IQueryTuple tuple = (IQueryTuple) it.next();
                String name = (String) tuple.getObject(1);
                Double price = (Double) tuple.getObject(2);
               System.out.println("DQSFacadeBean.executeQuery1(): Column1: " +
name);
               System.out.println("DQSFacadeBean.executeQuery1(): Column2: " +
price);
```

```
                ar.add(name);
                ar.add(price);
                arrl.add(ar);
                rows += 1;
            }
            System.out.println("DQSFacadeBean.executeQuery1(): Query returned
rows " + rows);
            // Insert the size of Resultset at element 0 in ArrayList.
            // This Arraylist is passed to the servlet for rendering purposes.
            arrl.add(0, new Integer(rows));
            return arrl;
        }
        catch (QueryException e){
            e.printStackTrace(System.out);
            throw new DQSException("DQSFacadeBean.executeQuery1(): " +
e.getMessage());
        }
        catch (NamingException e) {
            e.printStackTrace(System.out);
            return null;
        }
        catch (CreateException e) {
            e.printStackTrace(System.out);
            return null;
        }
        catch (RemoteException e) {
            e.printStackTrace(System.out);
            return null;
        }
    }
    private Collection executeQuery2() throws DQSException {
//...
    }
    private Collection executeQuery3() throws DQSException {
//...
    }
    private Collection executeFreeFormQuery(String queryInput) throws
DQSException {
//...
    }
}
```

DQSServlet servlet

In MVC design, the role of a servlet, as discussed above, is to serve as a controller for redirecting requests to the business tier and forwarding the response to a JSP, which in turn lays out the response according to the user's

client application specifications, for example a Web browser, a mobile browser, etc.

For the Dynamic Query sample application, the servlet was used not only to redirect the requests to the business tier, but it was also used to display the user response in HTML format, assuming that the client application is a HTML compliant Web browser.

Tip: For production and robust Web application, a Struts like framework should be used instead that provides a better implementation of MVC-2 architecture.

DQSServlet contains two private methods:

- **performTask()** method provides the logic for checking the HTML data submitted as POST operation to the servlet. Based on the input, it invokes the appropriate remote methods, executeDQ() and the overloaded executeDQ() of DQSFacade Session Bean. It then invokes the formatOutput() method before sending the response back to the client.
- **formatOutput()** method is a private method that formats the results from DQSFacade method calls based on the type of query submitted. This method assumes four output style formats:
  - Three predefined dynamic queries
  - One free-form Dynamic Query
  - Any other query types, based on malformed input from user's HTML form, are ignored and displayed as error on the servlet's result page

Example 19-2 shows the DQSServlet code. Take a note of performTask() method below to see how the initial lookup is performed for the DQSFacade home; EJB reference is used instead of DQSFacade JNDI name. For performance reasons, DQSFacadeHome is defined as a static variable such that it is instantiated only once, hence performing the JNDI lookup once.

Note: For space constraint, the entire code for DQSServlet.java is not shown in Example 19-2. Refer to the Dynamic Query sample application code that ships with redbook for details.

Example 19-2 DQSServlet Java code

```
public class DQSServlet extends HttpServlet implements Servlet {
    private static DQSFacadeHome dqsFacadeHome;
//...
    public void performTask(HttpServletRequest req, HttpServletResponse resp)
```

```
        throws ServletException, IOException {
        PrintWriter out = resp.getWriter();
        StringBuffer outBuf = new StringBuffer();
        try{
            outBuf.append(DQSConstants.HTML_PAGE_HEADER);
            outBuf.append(DQSConstants.HTML_DQS_RESULT_HEADER);
            String qType = req.getParameter("queryType");
            if (qType == null){
                outBuf.append("ERROR: 'queryType' parameter not
found.<BR>\n</BODY></HTML>\n");
                out.print(outBuf.toString());
                return;
            }
            System.out.println("DQServlet.performTask(): "+qType);
            if (dqsFacadeHome == null){
                InitialContext ic = new InitialContext();
                dqsFacadeHome = (DQSFacadeHome) ic.lookup("ejb/DQSFacade");
                outBuf.append("Looked up home.<BR><BR>\n");
            }
            DQSFacade dqsf = dqsFacadeHome.create();
            Collection co = null;
            if (qType.equals("q0")) {
                String qInput = req.getParameter("queryInput");
                if (qInput == null) {
                    outBuf.append("ERROR: 'queryInput' parameter not
found.<BR>\n</BODY></HTML>\n");
                    out.print(outBuf.toString());
                    return;
                }
                // free-form dynamic query
                co = dqsf.executeDQ(qType, qInput);

            } else {
                // pre-defined dynamic query
                co = dqsf.executeDQ(qType);
            }
            outBuf.append(formatOutput(qType, co));
            outBuf.append(DQSConstants.HTML_PAGE_FOOTER);
        }
        catch (Exception e) {
            e.printStackTrace(System.out);
            outBuf.append(e.getMessage());
        }
        finally {
            out.println(outBuf.toString());
        }
    }

    private StringBuffer formatOutput(String queryType, Collection co) {
```

```
        StringBuffer s = new StringBuffer();
        /*
         * Format of Collection (ArrayList) object is as follows
         * ArrayList[0] contains Integer object indicating # of rows returned
in resultset
         * ArrayList[1..n] contains an ArrayList object.  This ArrayList object
represents
         * columns returned as part of the resultset.
         */
        ArrayList arrl = (ArrayList) co;
        Integer rows = (Integer) arrl.get(0);
        if (rows.intValue() <= 0) {
            s.append("No rows returned<br>");
            return s;
        }
        if (queryType.equals("q1")) {
//...
        } else if (queryType.equals("q2")) {
//...
        } else if (queryType.equals("q3")) {
//...
        } else if (queryType.equals("q0")) {
//...
        }
        // All other queryType values, anything other than q0, q1, q2, q3,
        // are invalid and are not supported by the servet.
        else {
            s.append("<p>Non-supported Argument</p>");
        }
        return s;
    }
}
```

DQSForm HTML form

The HTML form serves as a user interface for the sample Dynamic Query application. The form is shown in the Figure 19-4 on page 513.

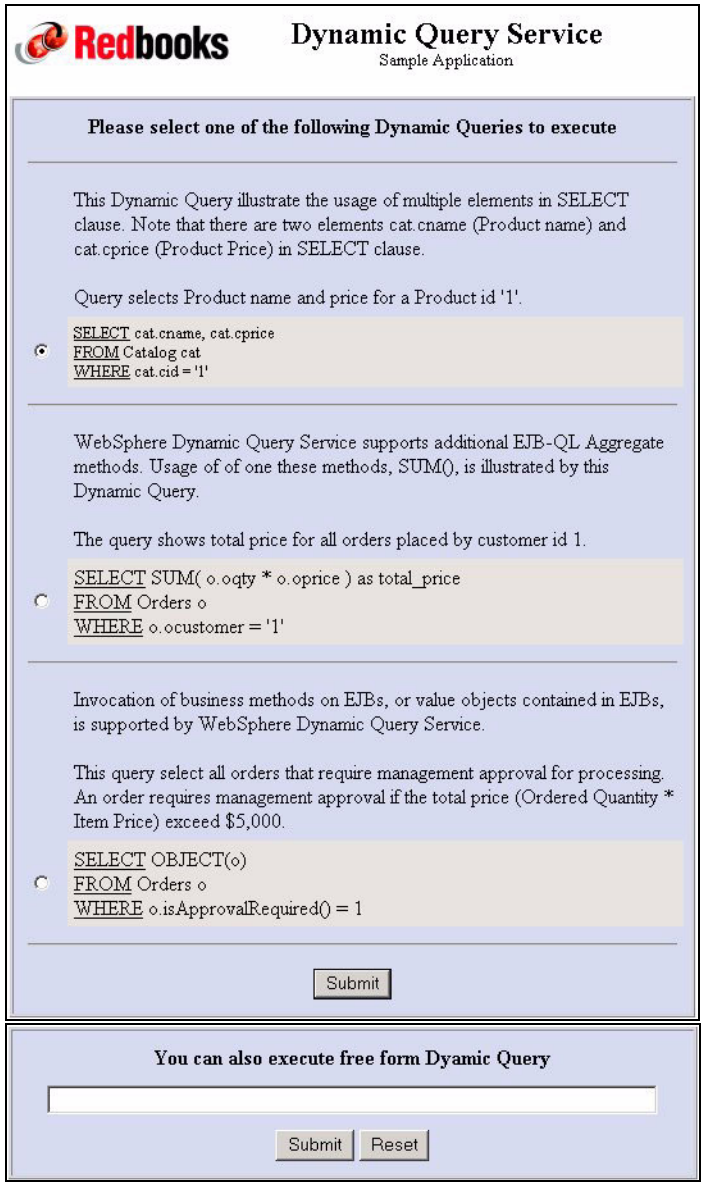


Figure 19-4 DQSForm HTML Form

19.4 Unit test

Dynamic Query sample application should be unit tested in WebSphere Studio Application Developer Integration Edition using the WebSphere Test Environment. The following sections discuss how to set up WebSphere Test Environment for the sample Dynamic Query application.

In order to unit test the sample application, the QueryBean session bean, packaged in query.ear, must be installed on the WebSphere Test Environment. This application, query.ear, should be automatically installed when application server is created in WebSphere Test Environment in the Server Perspective. However, it can be manually installed at any time using the Administrative Console within the WebSphere Test Environment. You must enable “Enable administration console” under the sample WebSphere Business Integration Server Foundation application server’s Configuration tab.

Note: In the runtime/production environment, QueryBean EAR query.ear, is automatically installed on the default server during the WebSphere Business Integration Server Foundation installation.

19.4.1 Configuring the application server

This section will show you how to configure the application server in the test environment.

1. Switch to the Server perspective, in the Server Configuration view. Create a Integration Test Environment V5.1, with the name: `DQ server`.
2. Open the server configuration for the test server, select the **Configuration** tab. Verify that Application class loader policy is set to `MULTIPLE`. This is required if a remote client is used for Dynamic Query.
3. Make the following additional changes for the server configuration:
   a. On Security tab, create a new JAAS Authentication Alias: `dqsAuth`. Assign your DB2 user name and password to this alias.
   b. On Datasources tab:
      i. Define a new JDBC Provider DB2 Universal JDBC Driver Provider (XA).
      ii. Define a WebSphere V5 datasource for the new JDBC provider. Set the JNDI name for this datasource as `jdbc/redbookDS`. Select the Component and Container managed authentication alias: **dqsAuth**.
      iii. Edit the properties shown for the datasource, set the databaseName to `REDBOOK`, then remove all the other properties.

c. On Variables tab, change the variables: DB2_JDBC_DRIVER_PATH, DB2UNIVERSAL_JDBC_DRIVER_PATH to reflect the java directory in the DB2 install path, for example: C:\IBM\SQLLIB\java.

d. On Configuration tab, you might want to enable Administration Console. You should have the Universal Test Client enabled for testing purposes.

4. Add the ACompanyDQS enterprise application to the test server.
5. Start the test server, make sure the server starts without any problem.

Tip: If you encounter problems with BPEDB (specifically an error message about upgrading your BPEDB by setting `upgrade=true` on your connection), use the attached script. This script must be executing using the JRE of your WBISF application server,

```
set JAVA_HOME=C:\WebSphere\WSADIE51\runtimes\ee_v51\java[should point to
the location of your WSADIE v5.1 installation]
set PATH=%PATH%;%JAVA_HOME%\bin
java -Djava.ext.dirs=%WBISF_V51_HOME%\cloudscape\lib
-Dij.protocol=jdbc:db2j: com.ibm.db2j.tools.ij
upgradeDatabaseCloudscape.ddl [make sure to update the path to your BPEDB
inside this script]
```

Refer to the code that ships with the redbook for information on upgradeDatabaseCloudscape.ddl. This script is bundled with the redbook code.

Important: Do not add a “query” EAR to the server. This is because query EAR is automatically installed in WebSphere Test Environment when the server is first created in WebSphere Test Environment.

19.4.2 Running the sample application

The following steps show how to run the sample Dynamic Query application:

1. Make sure that the application server is running and the ACompanyWeb and ACompanyEJB modules have started properly
2. Using a Web browser, open the following URL:
   `http://localhost:9080/acompany/DQSForm.html`
3. Select one of three predefined Dynamic Queries, or write your own Dynamic Query. This page needs a little bit more souping up as I left the interface quite simple. It will help to describe what each Dynamic Query does (I have documented this in detail in DQSServlet.java file):

- The first query shows that Dynamic Query can be used to retrieve multiple columns (standard finder methods only allows for 1)
- The second query shows the usage of SUM aggregate method that is only part of WebSphere Dynamic Query
- The third query shows how to invoke a business method defined on an Entity bean (Orders in this case). This method is part of the remote / local interface of the entity
- Additionally, free-form Dynamic Queries can be executed.

19.5 Configuration

The system administrator might need to install the query.ear application into the application server, if a new server is being created, since the WebSphere product install does this only for the default server. In the case of our sample, we are using a new server ACompanyServer, thus we need to install query.ear on ACompanyServer.

19.5.1 Installing query.ear

Since the WebSphere Business Integration Server Foundation installation only provides the Dynamic Query service with the default server (server1), if additional application servers are created, Dynamic Query service EAR must be installed on the new servers. To do that, you need to start the WebSphere Administrative Console, and choose to install a new application. You will need to point to the query.ear file available in `<WBISF_root>/installableApps/`.

Follow the steps to install the query.ear, but note the following:

1. Assign a new name to the application, since the name query is already available in the repository with the server1
2. Install this application on the new application server (and not server1).
3. After you finish the installation, you need to save your configuration.

After saving, the new application will be ready for deployment of applications that use the Dynamic Query service.

19.5.2 Application class loader policy configuration

In our sample application we are using the remote interface of the query bean. In order to use the remote interface of the query bean, you must configure your server to use Application Classloader Policy = MULTIPLE.

Important: Using a value of SINGLE may result in your application being unable to find the remote interface for the query bean home.

To configure WebSphere Business Integration Server Foundation runtime application server, select **Servers** → **Application Servers** → **<Server>**. This will open the editor for configuring the server's general properties. For the Application class loader policy, select the value MULTIPLE from the available drop-down list, and click **OK**. You will need to save the updated server configuration and restart the application server.

19.6 More information

The following section details additional information that should be kept in mind when developing applications using the Dynamic Query service.

19.6.1 Performance considerations

As mentioned in this chapter, dynamic queries can be performed on EJB objects and on CMP fields. The following section discusses some performance considerations regarding performing queries on objects and on fields.

Transactions and Dynamic Query

By default Dynamic Query makes calls against the database with the lowest level of locking possible. Using Dynamic Query, one can either select CMP fields or can select EJB objects. In cases when CMP fields are selected, there is no locking of the data selected. So in this case, you should be aware that the data you are selecting is volatile. Currently there is no way to enforce locking for the records when you are selecting CMP fields. This means that the use of Dynamic Query can be dangerous in some instances. Let's say you want to make a transaction based on information returned in the data set. Example 19-3 runs a query that returns all the account balances. The customers pay interest based on the account balance. However, this example is not safe. The reason is that the value of balance can change between the time when we run the query and the time when we assign a new balance. Moreover, the account might be deleted between running the query and looking up the account to set the balance. This is because the back-end rows are not locked when a data query runs.

Example 19-3 Selecting CMP fields

```
Select a.id, a.balance from Account a
For (x in resultset)
   if (a.balance  > 1000 && a.balance < 5000) {
```

```
            Account A = findByPrimaryKey(a.id);
            A.setBalance(a.balance*1.01);
   }
   if (a.balance  > 5000) {
            Account A = findByPrimaryKey(a.id);
            A.setBalance(a.balance*1.02);
   }
```

Therefore, the code for the balance update may be rewritten as Example 19-4 to overcome the problem of the possibility of account deletion. After the data is selected, findByPrimaryKey() method is invoked, and if the account has been deleted between running the query and the findByPrimaryKey(), AccountNotFoundException will be thrown. If the account still exists, the findByPrimaryKey() method has the side effect of locking the account for the transaction. The level of locking is based on the object's Access Intent. So there is still a possibility that the account balance will change during iterations, if the Account Bean Access Intent is optimistic.

Example 19-4 Selecting CMP fields with exception handling

```
Select a.id, a.balance from Account a
For (x in resultset)
try {
   Account A = findByPrimaryKey(a.id);
} catch (AccountNotFoundException e) {

}
if (A.getBalance() > 1000 && A.getBalance() < 5000) {
   A.setBalance(A.getBalance()*1.01);
}
if (A.getBalance() > 5000) {
   A.setBalance(A.getBalance()*1.02);
}
```

The only way to lock a piece of information is to return the instance that relates to it. The only way to implement Example 19-3 on page 517 safely is to select EJB objects instead of selecting CMP fields, and then perform the update, as shown in Example 19-5 on page 519.

Example 19-5 Selecting EJB instances

```
Select object(a) from Account a
For(x in resultset)
   If (a.getBalance() > 1000 && a.getBalance < 5000){
      a.setBalance(a.getBalance()*1.01);
and so on..
```

The side effect of selecting the EJB object instead of selecting the data fields is that all of the resulting instances are instantiated. This makes the performance much slower because it must return and lock each and every instance of Account. We will use an enormous amount of back-end resources. All work with Account will wait for this transaction to complete. If you have 10000 accounts to update their balances, you will end up instantiating 10000 objects in your runtime, which will bring the system performance down. Moreover, while all this is going on, you will have all these records blocked. The optimal use of query in this case would be to set a lock on the data that prevents updating but allow others to read.

A possible workaround to reduce the number of instances available in memory during runtime is to use the Query bean remote interface. The executeQuery method in the remote interface allows you to divide the result set into groups, by specifying the cursor position skipRows arguments, and the maxRows argument. But remember that you need to commit after each transaction to be able to release the lock on the selected objects.

One more issue worth noticing when using Dynamic Query is the possibility of using one the bean's method in the query where clause. Let's say we need to retrieve accounts whose balance is greater than 10000. In Example 19-6 the first query selects an Account object, which gets instantiated and used to retrieve the account balance. The second query selects data only. The Account bean is instantiated to evaluate getBalance() in the where clause, so you get a lock on the object based on the Account bean's default Access Intent. But since you are selecting CMP fields, the lock is released at the end of the transaction, not at the end of the query. The problem in this situation is in cases when the Access Intent is optimistic. The getBalance method will get invoked indicating that the balance is greater than 10000, then a moment later that balance might get changed by another user reducing the balance.

Example 19-6 Selecting objects with condition

```
select object(a) from Account a where a.getBalance()>10000
or
select a.id, a.balance from Account a where a.getBalance()>10000
```

19.6.2 Security considerations

As discussed earlier the Query session bean is responsible for executing queries on entity beans, and on CMP/CMR fields, through the use of its executeQuery() method. Security can be controlled on the use of this bean. That is you can control the use of the Dynamic Query service through granting or denying access to the Query bean. If you want to deny somebody the ability to perform queries, this is done by denying this user access to the Query session bean and its method. Once a user has access to the Query bean and the executeQuery() method, then the user can perform any query. You can control who is permitted access to the remote query bean and the local query bean, but once authorized a user can execute any valid query and return any data in the server. If you need security control over which queries a user can execute, you need to define the queries as finder or select methods and use EJB method authorization. The Dynamic Query service does not have fine-grain security control at this time.

A user may submit a query in which an object is being selected, or data array query. Access to a certain bean information can be controlled by granting/denying access to the bean and its methods. So if is an object query and the user does not have access to the bean, then the user won't be able to retrieve the desired data. For example, let's say the user submits a query such as:

```
SELECT OBJECT(e) FROM Employee e
```

If the user running the query doesn't have access to the Employee bean, then the user won't be able to execute this query, since this query requires instantiating the Employee bean.

Unfortunately WebSphere does not have security access control for CMP and CMR fields. So let's say that the user doesn't have access to the Employee bean, and submits a query such as:

```
SELECT e.name, e.salary FROM Employee
```

Submitting such a query will return the desired results to the user, although this user is not granted access to the Employee bean, since there is no security on the fields level, and since in such a type of query the Employee bean is not instantiated.

Another example that is worth watching for is a situation such as having two EJBs: an Employee EJB, which a user is not allowed to access, and a Department bean, which the user can access. Let's say the user submits a query such as :

```
SELECT e.name FROM Department d, IN (d.employees) as e WHERE d.deptNum = 10
```

This user will be able to retrieve the list of employee names, although this user is not allowed to access the Employee bean.

If the submitted query has a method call, then the user access privileges on that method are checked first before the method is invoked and the query is executed.

20

Object pools

The Object pools service can be used to improve the performance of multi-threaded applications that frequently use a common complex object. To do this, a Java application will have its own pool of objects that are already instantiated and waiting to be used. When the application needs a new object, it will simply fetch one from the pool and return the object to the pool when it is no longer required (instead of being destroyed). This returned object can then be later reused by another thread.

These Object pools are not intended for pooling Java Database Connectivity (JDBC) connections or Java Messaging Service (JMS) connections and sessions. The server tools provide specialized mechanisms for dealing with those types of objects. These Object pools are designed for pooling application-defined objects or Developer Kit types such as Vectors or HashMaps.

Java is a “memory-safe” language where the Java Virtual Machine (JVM) handles the allocation and deallocation of memory in a way that is transparent to the programmer. However, the instantiation and removal of objects can be expensive in terms of time and processing.

The advantages of using WebSphere’s Object pool service are as follows:

- Administration

  Object pools are managed by WebSphere’s administration mechanisms.

- Implementation level

  Object pooling is already implemented within the J2EE application container as an extension.

- Performance monitoring and tracing

  WebSphere's Performance Monitoring Infrastructure (PMI) and tracing infrastructure means you can monitor what is happening to the Object pool in the runtime environment.

- Variety

  Two Object pool types are offered: synchronized and unsynchronized. Definitions of these terms can be found in 20.5, "Runtime environment" on page 535.

- Flexibility

  If WebSphere's Object pool service is not suitable for your application, you can create your own Object pool implementation.

In *Java Theory and Practice: Garbage Collection and Performance* (IBM developerWorks Web site), Object pooling is described as beneficial for the most heavyweight objects on modern JVMs and recommended for special cases. To determine whether your application will benefit from Object pooling, you can run tests with simple code to determine how much time the JVM takes to instantiate given object types and perform garbage collection.

When testing, consider the size and complexity of the given object types and whether the object is frequently used. Using Object pools to manage simple object types may actually slow down your application.

20.1 Prerequisites

Following are some useful resources that will help you get started with Object pools:

- WebSphere Business Integration Server Foundation V5.1 InfoCenter

  `http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp?topic=/com.ibm.wasee.doc/info/welcome_ee.html`

- WebSphere Studio Application Developer Integration Edition V5.1 help can be found by clicking **WebSphere Business Integration Server Foundation** → **Highlights and Features** → **Object pools**.

- IBM Redbook: *WebSphere Application Server Enterprise V5 and Programming Model Extensions*, SG24-6932.

- The following article in IBM developerWorks:

  Goetz, B., 2004: *Java Theory and Practice: Garbage Collection and Performance* at

  http://www-106.ibm.com/developerworks/java/library/j-jtp01274.html

20.2 Sample scenario

This sample application demonstrates how performance can be improved by using Object pools. A specified number of job threads are created and the application returns the total average execution time for objects managed with and without Object pooling.

This application takes two parameters:

- Number of threads: the number of job threads it will create.
- Java class name: which Java class is being used.

Each job thread runs two different units of work several times and calculates the execution time for both cases: the first unit of work is without Object pools while the second unit uses Object pooling. A unit of work in JobThread is represented by the doSomething() and doSomethingUseObjectPool() methods. These two methods allocate an object of the specified class type, then put the thread to randomly wait between 1 and 6 milliseconds. This represents doing some work with the objects.

Migration

The Object pools feature in WebSphere Business Integration Server Foundation is equivalent to the implementation in WebSphere Application Server Enterprise V5.0; there are no specific migration steps for Object pools.

Development

The development of this Object pool example involves creating a client and a JobThread class. The client will create multiple threads with or without pooling and the results will be printed to the Console. We will also need to create a reference to an Object Pool Manager which will be defined in “Unit test” on page 526 and “Assembly” on page 526. For further details, refer to 20.3, “Development” on page 526.

Note: The code for the sample application described here is available in the additional material. You can import the code from the objectpool.pi.zip file using the Project Interchange plug-in.

Unit test

To run the Object pool example in the WebSphere Test Environment, we need to create an Object Pool Manager. For further details, refer to 20.4, “Unit test” on page 532.

Assembly

No assembly tasks are required. They have already been performed during the development process, specifically when we created a reference to an Object Pool Manager.

Deployment

There are no specific deployment tasks for Object pools.

Testing and runtime

Information about the runtime environment (including choosing Object pool types, sharing Object pools and workload management and failover capabilities) are explained in 20.5, “Runtime environment” on page 535. Information regarding how to configure an Object Pool Manager and custom Object pools can be found in 20.5.1, “Configuration in runtime” on page 536.

20.3 Development

This section provides a short introduction to the Object Pools API and sample code to show how to use the API.

20.3.1 Object Pools API

This section briefly describes the Object Pools API contained in the com.ibm.websphere.asynchbeans.pool package.

Object Pool Manager

The Object Pool Manager interface describes the Object Pool Manager which is a factory for Object pools. The Object Pool Manager interface defines two methods:

- createFastPool(Class): this method returns an unsynchronized Object pool.
- getPool(Class): this method returns a synchronized Object pool.

Object pool

The Object pool is a pool of reusable objects of the same type. Using WebSphere's default Object pool implementation, any Java object with a default constructor can be pooled. The Object pool interface defines two methods:

- getObject(): retrieves an object from the Object pool.
- returnObject(Object): returns an object that is not needed to the Object pool.

Custom Object pool

You can use this interface to create your own Object pool implementation if WebSphere's Object pool implementation does not suit the needs of your application. For instance, you may require notification when objects are taken from and returned to the Object pool.

The custom ObjectPool interface extends the ObjectPool interface with these additional methods:

- flushPool(): called when the system needs memory and removes any idle objects from memory.
- setProperties(Map): called when the custom pool is constructed. Refer to "Custom Object pool configuration" on page 537 for details.

Poolable object

When using WebSphere's default Object pools implementation, you may require some initialization or cleanup code in the object for when the object to taken from or returned to the pool. The object needs to implement the PoolableObject interface with these two methods:

- init(): Called when the object is taken from the pool
- returned(): Called when the object is returned to the pool.

Note: As you may already determined from the Java API name, Object pools are related to Asynchronous Beans. Asynchronous Beans (and the Internationalization service) internally use Object pooling.

20.3.2 Coding with Object pools

To develop this example that uses Object pools, you will:

- Create a new Project
- Create the ObjectTest application client
- Add a resource environment reference
- Change the main class of the ObjectTest application client
- Create a JobThread class

Follow these steps to develop the code for the Object Pools example:

1. Start WebSphere Studio Application Developer Integration Edition.
   a. Make sure you enable server targeting support.
   b. Switch to the **J2EE Perspective**.
2. Create a new Project
   a. In WebSphere Studio Application Developer Integration Edition, select **File** → **New** → **Application Client Project**.
   b. Select **Create J2EE 1.3 Application Client** project and type `ObjectPool` in th Project field and `ObjectPoolEAR` in the EAR Project field, select **Integration Server v5.1** for the Target server. Click the **Finish** button.
3. Create the ObjectTest application client
   a. In the Project Navigator view, right-click the **ObjectPool** project and select **New** → **Class**. In the New Java Class window, enter `com.ibm.itso.sg246318.ObjectPool` in the Package field, enter `ObjectTest` in the Name field. Click the **Finish** button.
   b. Copy the code as shown in Example 21-1 into ObjectTest.java.

      In the code, you can see that the client finds the Object Pool Manager and then creates a specified number of job threads and starts them.

Example 20-1 ObjectTest.java

```
package com.ibm.itso.sg246318.ObjectPool;

import javax.naming.InitialContext;
import com.ibm.websphere.asynchbeans.pool.ObjectPoolManager;

public class ObjectTest {

   public static void main(String[] args) {
      if (args.length < 2) System.out.println("usage:ObjectTest numberOfThreads className");
      int numberOfThreads = Integer.parseInt(args[0]);
      JobThread[] threads = new JobThread[numberOfThreads];
      ObjectPoolManager opm = null;
      String className = args[1];
      InitialContext initalContext;
      //Lookup for object pool manager
      try {
         System.out.println("ObjectTest: looking for ObjectPoolManager");
         initalContext = new InitialContext();
         opm = (ObjectPoolManager) initalContext.lookup("java:comp/env/ObjectPool");
      } catch (Exception ex) {
         ex.printStackTrace();
      }
```

```
        for (int i = 0; i < numberOfThreads; i++) {
           //Create a new JobThread thread
           threads[i] = new JobThread("JobThread_" + i, opm, className);
           System.out.println("ObjectTest: starting thread: " + threads[i].getName());
           //Start the thread
           threads[i].start();
        }
     }
}
```

Important: Ignore any errors at this point since they will be resolved when JobThread Java class is created later in this chapter.

c. Save and close the file.

4. Add a resource environment reference.

   Since a local reference to Object pool is used in ObjectTest.java, we need to add a resource environment reference to the application:

   a. Open the **Client Deployment Descriptor** of the ObjectPool project. Switch to the **References** tab and click the **Add** button.

   b. Select **Resource environment reference** and click the **Next** button.

   c. Enter `ObjectPool` in the Name field and enter `com.ibm.WebSphere.asynchbeans.pool.ObjectPoolManager` in the Type field. Click the **Finish** button.

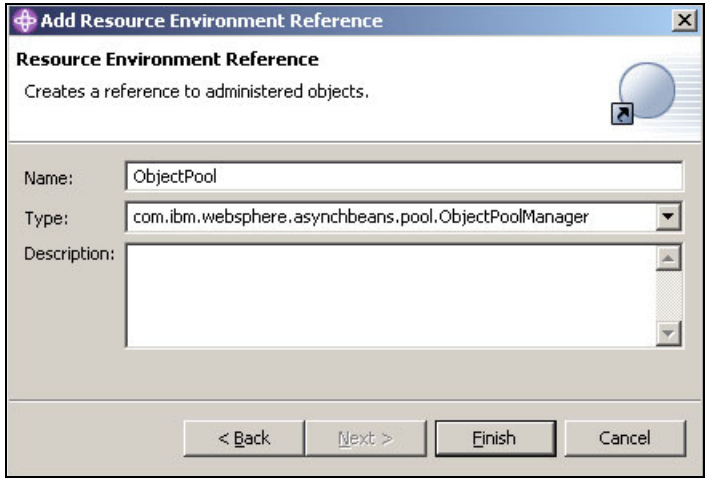


Figure 20-1 Add Resource Environment Reference

d. Select the created resource environment reference and enter `ObjectPoolMgr` in the JDNI name field.

e. Save and close the descriptor.

5. Change the main class of the application client.

   The next step is to change the specified main class of the client so the J2EE client container will know which class to run when starting the application.

   a. Open the MANIFEST.MF file in **appClientModule** → **META_INF** in the ObjectPool project.

   b. On the Dependencies tab, enter `com.ibm.itso.sg246318.ObjectPool.ObjectTest` in the Main Class field.

   c. Save and close the file.

6. Create JobThread class.

   The last step is to create a JobThread class.

   a. Right-click the **ObjectPool** project and select **New** → **Class**.

   b. In the New Java Class window, enter package `com.ibm.itso.sg246318.ObjectPool` for Package, enter `JobThread` in the Name field. Click the **Finish** button.

   c. Copy the code in Example 20-2 into JobThread.java.

Example 20-2 JobThread.java

```
package com.ibm.itso.sg246318.ObjectPool;

import java.util.*;
import com.ibm.websphere.asynchbeans.pool.*;

public class JobThread extends Thread {
   static Double averageExecutionTime = new Double(0);
   static Double averageExecutionTimeUsingPool = new Double(0);
   static Long totalNumberOfRepetitions = new Long(0);
   Random rand = new Random(System.currentTimeMillis());
   ObjectPool oPool = null;
   ObjectPoolManager opm;
   String className = new String();
   Class customClass;

//Initialize
   public JobThread(String name, ObjectPoolManager opm, String className) {
      super.setName(name);
      this.opm = opm;
      this.className = className;
}
```

```
   public void run() {
      // Wait between 1-100 milliseconds before start
      synchronized (this) {
         try {
            this.wait(1 + rand.nextInt(100));
         } catch (InterruptedException e) {};
      }
      try {
         customClass = Class.forName(className);
         //Get an object pool for specified class
         oPool = opm.getPool(customClass);
      } catch (Exception ex) {
         ex.printStackTrace();
      }
      int repetitions = rand.nextInt(500);
      synchronized (totalNumberOfRepetitions) {
         totalNumberOfRepetitions = new Long(totalNumberOfRepetitions.longValue() +
repetitions);
      }
      //Do something using objects without object pooling
      //Repeat the job 1-500 times
      //Read the time before starting the job
      long startTime = System.currentTimeMillis();
      for (int i = 0; i < repetitions; i++) {
         doSomething();
      }
      //Read the time after ending the job
      long endTime = System.currentTimeMillis();
      //Do something using objects WITH object pooling
      //Repeat the job 1-500 times
      long startTimeUsingObjectPool = System.currentTimeMillis();
      for (int i = 0; i < repetitions; i++) {
         doSomethingUseObjectPool();
      }
      long endTimeUsingObjectPool = System.currentTimeMillis();
      //Calculate doSomething execution time and add it to the total value
      synchronized (averageExecutionTime) {
         averageExecutionTime = new Double(averageExecutionTime.doubleValue() + endTime -
startTime);
      }
      //Calculate doSomethingUseObjectPool execution time and add it to the total value
      synchronized (averageExecutionTimeUsingPool) {
      averageExecutionTimeUsingPool = new
      Double(averageExecutionTimeUsingPool.doubleValue() + endTimeUsingObjectPool -
      startTimeUsingObjectPool);
   }
   System.out.println("Exiting thread" + this.getName());
   System.out.println("doSomething average execution time = " + (endTime - startTime) /
repetitions);
```

```
   System.out.println("doSomethingUseObjectPool average execution time = " +
(endTimeUsingObjectPool - startTimeUsingObjectPool) / repetitions);
   System.out.println("Total average execution time = " + averageExecutionTime.doubleValue() /
totalNumberOfRepetitions.doubleValue());
   System.out.println("Total average execution time using pool= " +
averageExecutionTimeUsingPool.doubleValue() / totalNumberOfRepetitions.doubleValue());
   }

   public void doSomething() {
      try {
         Object aL = customClass.newInstance(); //Create a new object
         randomWait(1 + rand.nextInt(5)); //Wait between 1 and 6 miliseconds
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }

   public void doSomethingUseObjectPool() {
      Object obj = oPool.getObject(); //Get an object from the object pool
      randomWait(1 + rand.nextInt(5));//Wait between 1 and 6 miliseconds
      oPool.returnObject(obj);//Return the object to the object pool
   }

   synchronized private void randomWait(int miliseconds) {
      try {
         this.wait(miliseconds);
      } catch (InterruptedException e) {}
   }
}
```

7. Save and close the file. All the errors from the Task view should disappear.

20.4 Unit test

To run the code in the WebSphere Test Environment, you need to add an Object Pool Manager to the Server Configuration.

1. Open the Server perspective and create a new Integration Test Server. If you need help with creating a test server, refer to , “Integration Server V5.1 test environment setup” on page 563.
2. Open the Server Configuration by double-clicking the test server you have created.
3. On the Object Pools tab, click the **Add** button under Server Settings, next to the Object Pool Managers. Enter `AObjectPoolManager` in the Name field and `ObjectPoolMgr` in the JNDI Name field. Click the **OK** button.

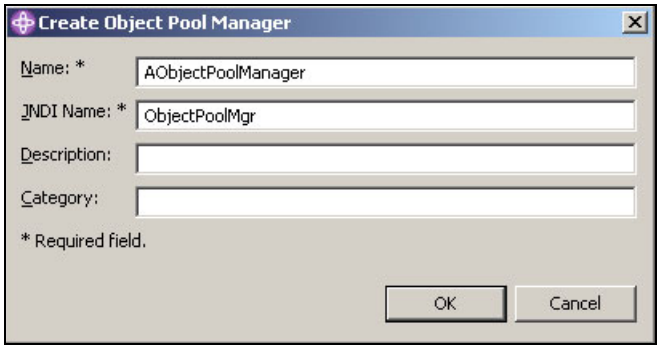


Figure 20-2 Creating an Object Pool Manager

4. Save the Server Configuration and right-click the Object Pool Server and select the **Start** button.
5. Now we can create a client to run the Object Pools example.
   a. Switch to the J2EE Perspective, **Run** → **Run...** from the menu.
   b. Select **WebSphere v5.1 Application Client** and click the **New** button.
   c. Enter `Object Pool Client` in the Name field.
   d. Select **WebSphere v5.1 EE** for Server Type.
   e. Select the **Arguments** tab and add the following parameter to the Program arguments:

      ```
      -CCverbose=true 500 java.util.ArrayList
      ```

      Using these parameters, logging is turned on and 500 job threads are going to be created that pool ArrayList Java objects.
   f. Change all the *base_v51* directory names to `ee_v51` under the VM arguments; there are eight occurences.
   g. Click **Apply**.
   h. Click **Run** to run the application client.
6. On the Console, you will see output similar to what is shown in Example 20-3.

Example 20-3 Output from Object Pools example

```
IBM WebSphere Application Server, Release 5.1
J2EE Application Client Tool
Copyright IBM Corp., 1997-2003
WSCL0012I: Processing command line arguments.
WSCL0001I: Command line, property file, and system property arguments resolved to:
        File to launch          = C:/Documents and Settings/boardman/My
Documents/IBM/wsappdevie51/workspace/ObjectPoolEAR
```

```
        CC Property File        = null
        Client Jar File         = ObjectPool.jar
        Alternate DD            = null
        BootstrapHost           = kaOklfc
        BootstrapPort           = <default>
        Trace enabled           = false
        Tracefile               = null
        Init only               = false
        Classpath Parameter     = null
        Security Manager        = disable
        Security Manager Class  = Not used. -CCsecurityManager=disable
        Security Manager Policy = Not used. -CCsecurityManager=disable
        Exit VM                 = false
        Soap Connector Port     = null
        Application Parameters  = 500 java.util.ArrayList
        Provider URL            = null
        Dump Java Name Space    = null
        Admin Connector Host    = null
        Admin Connector Port    = null
        Admin Connector Type    = null
        Admin Connector User    = null
WSCL0013I: Initializing the J2EE Application Client Environment.
...
WSCL0035I: Initialization of the J2EE Application Client Environment has completed.
WSCL0014I: Invoking the Application Client class ObjectTest
ObjectTest: looking for ObjectPoolManager
ObjectTest: starting thread: JobThread_0
ObjectTest: starting thread: JobThread_1
ObjectTest: starting thread: JobThread_2
ObjectTest: starting thread: JobThread_3
ObjectTest: starting thread: JobThread_4
ObjectTest: starting thread: JobThread_5
...
Exiting threadJobThread_128
doSomething average execution time = 63
doSomethingUseObjectPool average execution time = 23
Total average execution time = 0.06144919781625436
Total average execution time using pool= 0.10548360936703147
...
Exiting threadJobThread_419
doSomething average execution time = 11
doSomethingUseObjectPool average execution time = 5
Total average execution time = 14.616670701202292
Total average execution time using pool= 8.448640361494393
```

Results

We received the following test results on our system:

```
Total average execution time           = 14.616670701202292
Total average execution time using pool= 8.448640361494393
```

The results can be interpreted as follows. When using 500 threads and the java.util.ArrayList class, it took approximately 14 milliseconds on average per job task when no Object pools were used. When Object pooling was used, it took approximately 8 milliseconds on average per job task.

Note: Obviously, test results depend on the capabilities of the test system. Even with considering this fact, the performance difference, between using Object pools or not using, is significant enough to accept the results as a good indicator of performance.

You can use the application to test the difference using your own classes. You can also check how the JVM memory settings affect behavior.

Important: Although this sample application shows the performance gain for Object pools, it is just a proof of concept. You cannot expect the same performance gains with a “real” application because there are many factors that have not been and cannot be calculated here.

20.5 Runtime environment

This section discusses Object pools in the runtime environment and configuration settings.

Choosing Object pool types

Do not forget to select the correct pool type, whether is synchronized or unsynchronized.

- A *synchronized pool* is useful where little contention exists for objects in the pool. A synchronized pool is thread-safe, meaning it can also be shared across threads. This way objects are used more efficiently as there are fewer idle objects in the pool.
- An *unsynchronized pool* is useful if your application uses an Object pool in one thread. An unsynchronized pool is not thread-safe meaning that applications can only use them within one thread. Unsynchronized pools are faster than synchronized pools.

Sharing Object pools

If an application server shares an Object pool between applications, class loader problems may occur between shared application objects. We recommend to not share pools between applications when *application objects* (and not JDK supplied objects such as java.util.HashMap) are being pooled.

Workload management and failover

The Object pool service does not have any special workload management or failover capabilities since it is running within the scope of a JVM. When you configure the Object pool configuration manager on the cell or on the node level, every application server that belongs to the cell or node will have one with the specified name registered, and they are not aware of each other. That is, if you work with Object pools on one server and you lose it, your application can still continue to run on another server. However, since it exists only in memory, Object pools on the crashing server will be lost.

Programmatically, you can still return the objects obtained from the crashed server pool to the new pool, but if the type of the pooled class is not the same, you will get a ClassCastException exception.

20.5.1 Configuration in runtime

This section describes how to configure your Object Pool Manager and custom Object pool using the Administrative Console. It also explains how to disable the Object pools service.

Object Pool Manager configuration

Perform the following steps to configure the Object Pool Manager:

1. To enable the Administrative Console, doubleclick the server and mark the Enable administrative console checkbox on the Configuration tab.
2. Start the server. When it has fully started up, rightclick the server and select Run administrative console.
3. Select **Resources** → **Object pools** and click the **New** button to create a Object Pool Manager.
4. In the Object Pool Manager configuration window, type `AObjectPoolManager` in the Name field and `ObjectPoolMgr` in JNDI Name field.

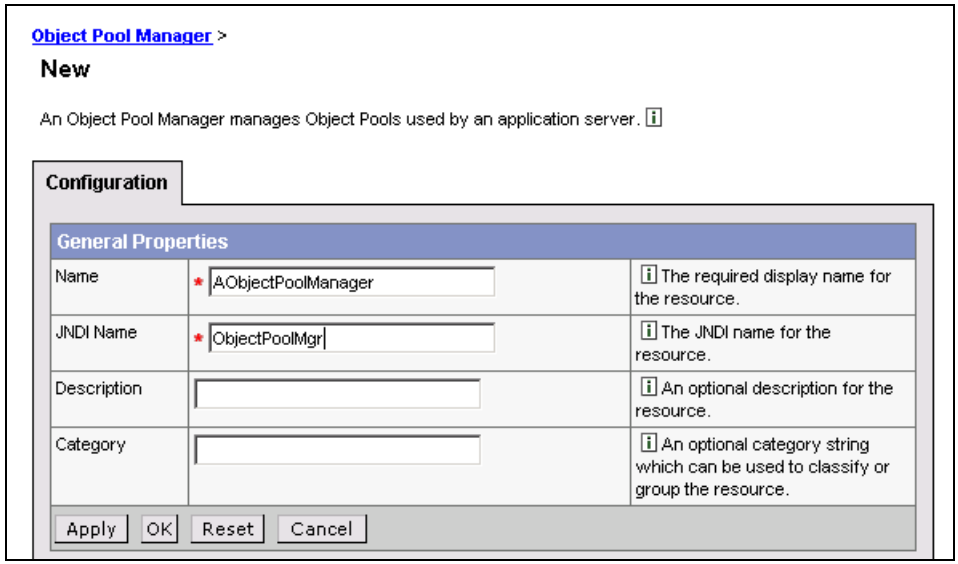


Figure 20-3 Creating an Object pool manager in the Administrative Console

5. Apply the changes and save the configuration for WebSphere.

Disabling the Object pool service

The Object pool service is enabled by default. It starts during the WebSphere startup. If you want to disable the service, take the following steps:

1. Launch the Administrative Console and log in.
2. Select **Servers** → **Application Servers**. Select the application server that has defined the Object pool service that you want to disable.
3. Under the Configuration tab, click **Object Pool Service**. In the Additional Properties table, unmark the **Startup** property.
4. Apply the changes and save the configuration for WebSphere.

Custom Object pool configuration

When WebSphere's Object pool implementation does not suit your application's needs, you can provide a customized Object pool implementation which implements the CustomObjectPool interface. Every custom Object pool has to be registered under an existing Object pool manager.

In order to define a custom Object pool on a given Object pool manager, click **Resources** → **Object Pools** → ***<your_Object_Pool_Manager>*** and click **Object Pools**.

You can create a new custom Object pool, by clicking the **New** button. Specify the type of class that is pooled in the Pool Class Name field and the class name of your Object pool implementation in the PoolImpl Class Name field. Also, by clicking the Custom Properties link, you can specify custom properties that will be given to your pool implementation in runtime when the customer Object pool is created.

20.6 Problem determination and troubleshooting

If you encounter problems using Object pool service and you suspect that the problem source is somewhere within the service then you can additionally use the following trace string to monitor what is happening in the execution time:

```
com.ibm.ws.asynchbeans.pool.*=all=enabled
```

21

Chapter 21

Internationalization (i18n)

The internationalization (i18n) service adds APIs and tooling that enable J2EE applications to manage the distribution of internationalization information via the internationalization context.

Note: The i18n name is derived from the first (i) and last (n) letter of the word internationalization and the number of characters in between (18). It is simply easier to write i18n than internationalization.

The service itself provides the mechanism to transparently propagate locale and time zone information from clients to servers and between server components. This information can be used by server application components to customize the results according to the client locale and time zone.

Why use the Internationalization service? In a distributed client/server environment, application processes may run on different machines configured to different locales corresponding to different culture conventions. They may also be located across geographical boundaries. With the advent of Internet-based business computational models, such as e-commerce, more and more clients and servers will operate in different locales and geographical regions.

Internationalization techniques have traditionally been expensive and difficult to implement, so they have been applied only to major development efforts. However, given the rise in distributed computing and in the use of the World Wide Web, application developers have been pressured to internationalize a

much wider variety of applications. This requires making internationalization techniques much more accessible to application developers.

21.1 Prerequisites

Learn more about the i18n service from the IBM redbook *WebSphere Application Server Enterprise V5 and Programming Model Extensions*, SG24-6932.

You can also find detailed information about the i18n API and other WebSphere Business Integration Server Foundation V5.1 related topic in the InfoCenter at:

`http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp`

by clicking **WebSphere Business Integration Server Foundation** → **Highlights and Features** and selecting **Internationalization service**.

You can learn about i18n basics at developerWorks using the following tutorial:

`http://www-106.ibm.com/developerworks/java/edu/j-dw-javai18n-i.html`

The article *International calendars in Java* at developerWorks at:

`http://www-106.ibm.com/developerworks/java/library/j-javacal/`

provides technical details about handling calendars in Java.

21.2 Sample scenario

In this chapter, we will not develop an application from scratch; instead, you will find details of the application provided as additional material.

The sample application is a very simple Web application sending greetings to an EJB application. The EJB application consists of two session EJBs, one to receive the greetings and another to log the greetings.

Migration

There are no migration-specific tasks related to i18n. The i18n service is the same as in WebSphere Enterprise V5.0.

Development

You can find details about the sample application in this section. In addition to the programming details of the i18n API, you will see how to configure i18n for application components.

There are two options for handling the i18n context in a component (for example, EJB, servlet):

- Container managed internationalization
- Application managed internationalization

In both cases, the i18n context is available for the programmers. When the container manages the context, you can set up the i18n features using the deployment descriptor. Application management means that the programmer is responsible for setting up the i18n attributes for the application programmatically.

Unit test

During the unit test, you can easily test your application using the i18n services.

Assembly

There are a few assembly tasks for enterprise applications and application components, depending on the application design.

You can configure i18n attributes for EJBs and servlets.

Deployment

There are no specific tasks for deployment when using i18n services in applications.

Runtime

There is only one task related to i18n in the runtime environment. Administrators can enable or disable the i18n service for an application server.

Important: The i18n service is not enabled by default; you have to enable it before you can use it with any of your applications.

Enabling or disabling the i18n service requires restarting the server.

21.3 Development

The application described in the previous section is distributed together with the book as additional material.

1. Launch WebSphere Studio Application Developer Integration Edition with an empty workspace. Make sure you enable the server target support.
2. Import the i18n.pi.zip archive using the Project Interchange plug-in.

You will find the i18nEAR enterprise application with a Web module (i18nWeb) and an EJB module (i18nEJB).

There are three scenarios covered in this sample:

1. Servlet using i18n with container-managed i18n.

2. EJB using i18n with application-managed i18n.
3. EJB using i18n with container-managed i18n. This one is very similar to the first (servlet) scenario, except that in this case we are configuring the EJB deployment descriptor, not the Web deployment descriptor.

i18n for servlets

This section shows how to set up i18n for a servlet and how to use the i18n API to access i18n information from the context.

1. Open the Web deployment descriptor for the i18nWeb module.
2. Switch to the Extended Services® tab.
3. Select the **SendGreeting** servlet.
4. Check the Internationalization section for the servlet; you will find it already configured.

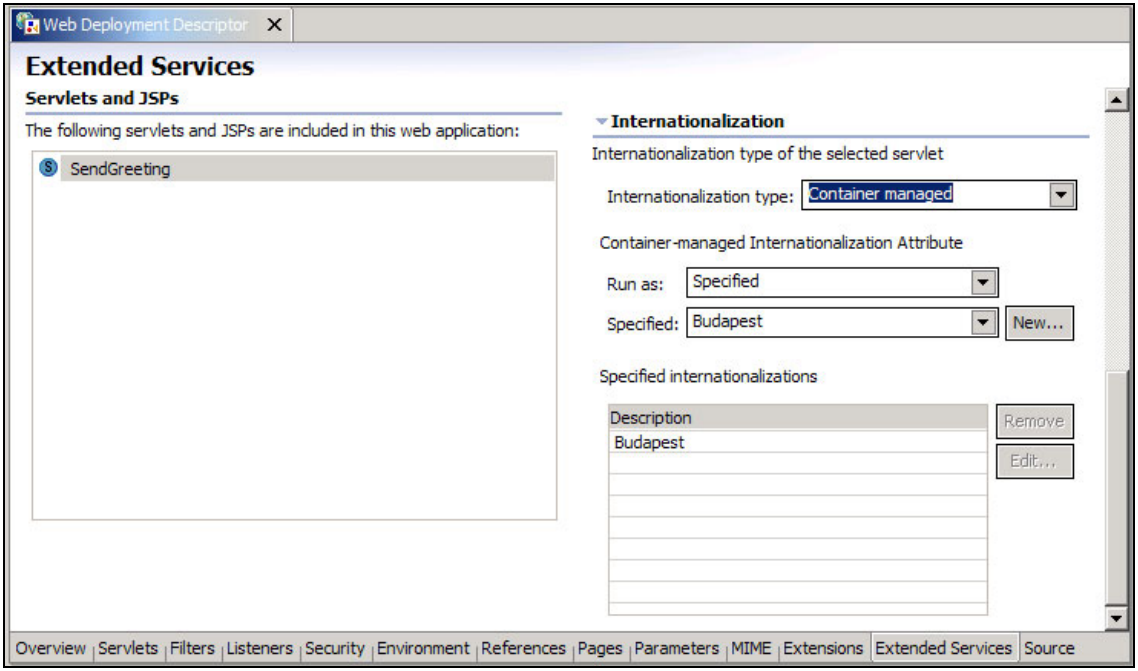


Figure 21-1 i18n configuration for servlet

When you configure i18n for a servlet, perform the following steps:

1. Set the Internationalization type, `Container managed` or `Application managed`.

2. Set the Run as `caller`, `server` or `specified`.
   a. If you select `caller`, then the i18n attributes are simply propagated from the caller for the next invocation.
   b. If `server` is selected then the server's i18n settings will be used for the servlet.
   c. When `specified` is selected, you will need to configure one or more attributes for i18n. Create a new specified item for the servlet, then click **New**. You need to set up two fields for i18n: time zone and locale, where locale normally consists of two fields: region (country) and language. Fill out the details, as in Figure 21-2.

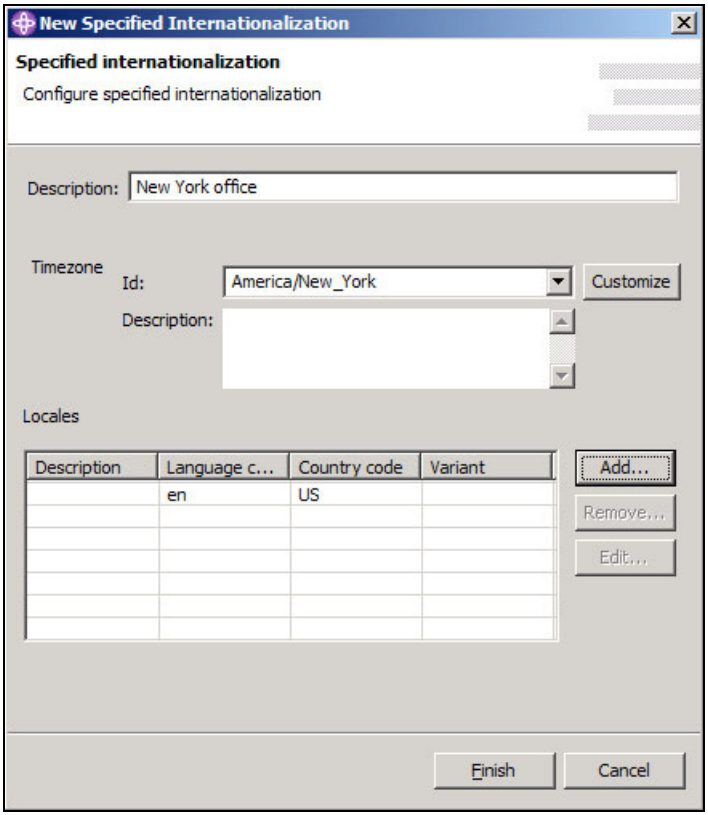


Figure 21-2 New specified settings for i18n

Add a new Locale to your settings, as shown in Figure 21-3 on page 545.

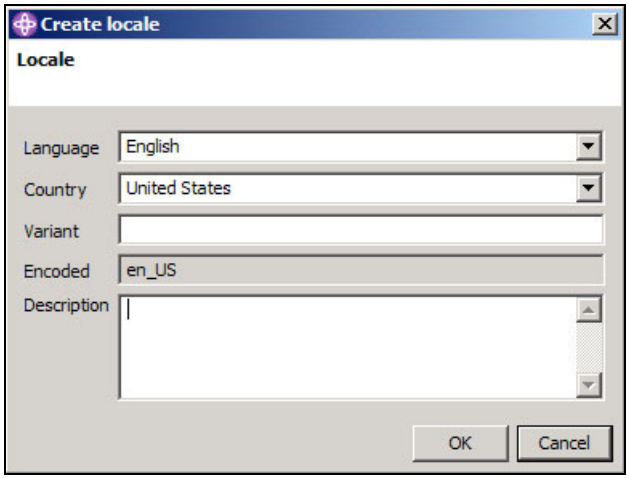


Figure 21-3 New locale for i18n

3. Save the deployment descriptor.

Let's take a look at the code details for the container-managed i18n. First, you have to look up the i18n service context in your servlet from the JNDI.

Example 21-1 i18n in a servlet

```
//...
//necessary import statements
import com.ibm.websphere.i18n.context.*;
import com.ibm.ws.i18n.context.*;
import javax.naming.*;
import java.util.Locale;
//...
// class attributes for i18n
protected UserInternationalization userI18n = null;
protected Internationalization callerI18n = null;
protected InvocationInternationalization invI18n = null;
public static final String UserI18nUrl =
   "java:comp/websphere/UserInternationalization";

// we put the i18n context lookup in the servlet's init method
public void init() throws ServletException {
   //getting the i18n context from the JNDI
   Context initialContext=null;
   try {
      initialContext = new InitialContext();
      userI18n = (UserInternationalization)initialContext.lookup(UserI18nUrl);
      callerI18n = userI18n.getCallerInternationalization();
```

```
      invI18n = userI18n.getInvocationInternationalization();
   } catch (NamingException ne) {
      throw new ServletException("Cannot resolve UserInternationalization" +
ne);
   } catch (IllegalStateException ise) {
      throw new ServletException ("Error: UserInternationalization is not
available: " + ise);
   }
   //...
}
// servlet's doPost method
public void doPost(HttpServletRequest req, HttpServletResponse resp)
   throws ServletException, IOException {
   //...
   try {
      // retrieving the caller locale
      callerLocale = callerI18n.getLocale();
      // retrieving the invocation locale
      invocationLocale = invI18n.getLocale();
   } catch (IllegalStateException ise) {
      errorMsg="An anomaly occurred accessing I18N Invocation context: ";
   }
   // dump out the results
   System.out.println("Sending greeting: "+greeting);
   System.out.println("  caller locale: "+callerLocale);
   System.out.println("  invocation locale: "+invocationLocale);
   //...
}
```

Note that the JNDI name for the i18n context is *java:comp/websphere/UserInternationalization*, and it has not been set up for the application as a resource reference. This object is available once the i18n service is enabled for the application server.

The i18n context lookup is coded in the servlet's init() method. It is a process you do not want to go through very often since it is costly from a performance point of view.

Later in the doPost() method, you can see how the caller and invoker locales are retrieved from the context. Once you have the locale, you can extract region and language codes as well. Besides the locale, you can also retrieve the time zone from the context. See the API documentation for details.

Note: i18n in servlets uses the accept-language HTTP header from the browser's request to determine the language for the locale.

i18n for EJBs using container-managed internationalization

This section shows how to configure i18n for a session bean using application-managed i18n; we then take a look at i18n fields programmatically.

1. Open the EJB deployment descriptor.
2. Switch to the Internationalization tab.
3. Check the *Internationalization type, Beans configured for application-managed internationalization* section; the ReceiveGreeting is configured here.

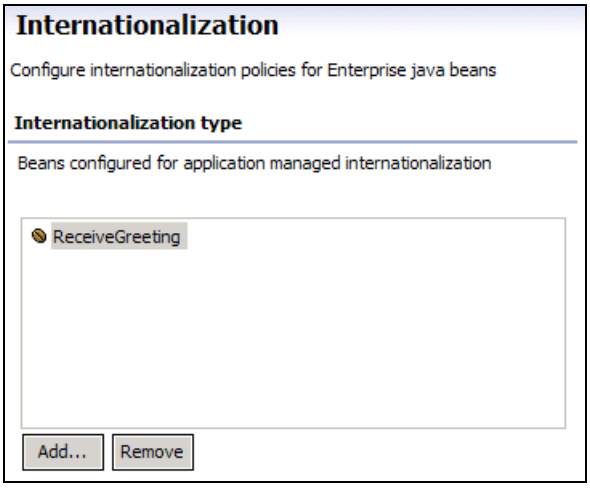


Figure 21-4 Application managed i18n

When you configure an EJB to use application-managed i18n, perform the following steps:

1. Click **Add...** under *Beans configured for application-managed internationalization*.
2. Select the EJB that you want to add to the application-managed i18n list.

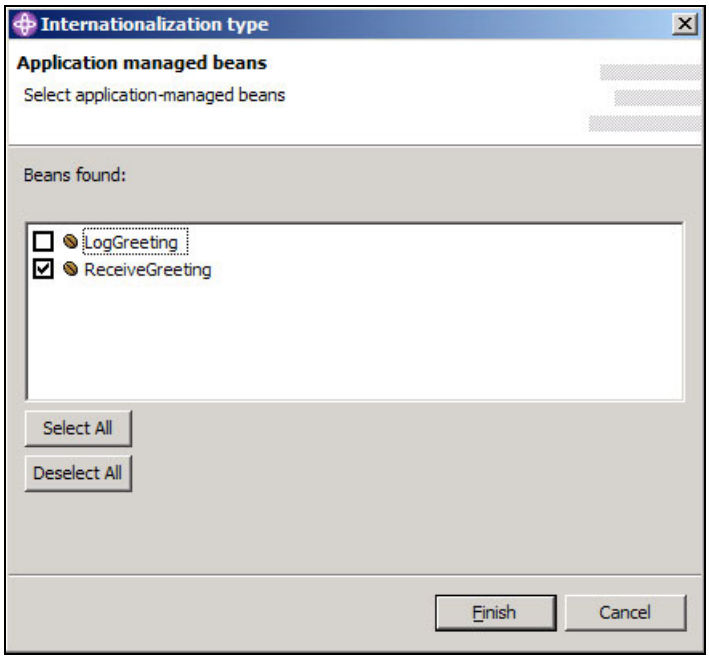


Figure 21-5 Adding EJB to application-managed i18n

3. Save the deployment descriptor.

When an EJB is enabled for application-managed i18n, the application server manages the i18n context for the bean, but no attributes are set for the context by default, nor from the deployment descriptor.

The following example shows how to use the i18n API in an EJB.

Example 21-2 i18n in an EJB

```
//...
//necessary import statements
import com.ibm.websphere.i18n.context.*;
import javax.naming.*;
import java.util.Locale;
//...

// attributes for i18n
protected UserInternationalization        userI18n = null;
protected InvocationInternationalization  invI18n  = null;
public static final String UserI18NUrl =
"java:comp/websphere/UserInternationalization";
```

```
public void ejbCreate() throws javax.ejb.CreateException {
   //getting the i18n context from the JNDI
   try {
     Context initialContext = new InitialContext();
     userI18n = (UserInternationalization)initialContext.lookup(UserI18NUrl);
     invI18n = userI18n.getInvocationInternationalization();
   } catch (NamingException ne) {
      ne.printStackTrace();
   } catch (IllegalStateException ise) {
      ise.printStackTrace();
   }
   //...
}

public void receiveGreeting(String greeting) {
   Locale callLocale = null;
   Locale invLocale = null;
   try {
      // retrieving the caller locale
      callLocale = userI18n.getCallerInternationalization().getLocale();
      // setting the invocation locale
      invI18n.setLocale( new Locale("pt_BR"));
      invLocale = invI18n.getLocale();
   } catch (IllegalStateException ise) {
      ise.printStackTrace();
   }
   // dump out the results
   System.out.println("Received greeting: "+greeting);
   System.out.println("  caller locale: "+callLocale);
   System.out.println("  invocation locale: "+invLocale);
   //...
}
```

Using i18n in the EJB is equivalent to the servlet version. There is no need to configure any resource reference for the i18n context.

The context lookup from the JNDI is taken care of in the ejbCreate() method.

In the receiveGreeting() method, you can see how the i18n attributes are retrieved and set. The caller attributes are available since they are propagated from the servlet, but there is no configuration for the invocation attributes. The invocation is set using the i18n API. You can set one or more locales in the i18n context; this is also true of the time zone. See the API documentation for more details.

i18n for EJBs using application-managed internationalization

This section provides details about container-managed i18n for EJBs.

1. Open the EJB deployment descriptor.
2. Switch to the Internationalization tab.
3. Check the *Internationalization attributes, Method-scoped policies configured for container-managed internationalization beans*; the logGreeting() method is configured here.

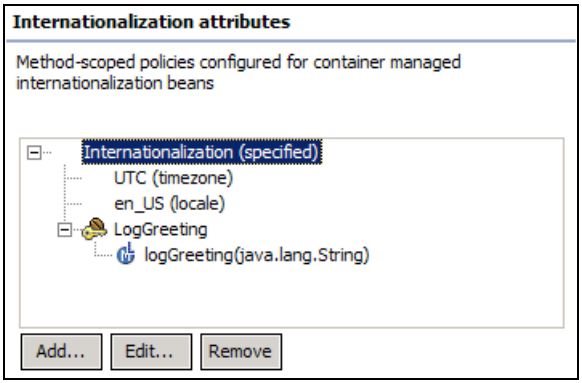


Figure 21-6 Container-managed i18n for EJB methods

To create a new entry for container-managed i18n, follow the steps below:

1. Click **Add...** under *Method-scoped policies configured for container-managed internationalization beans*.
2. Set Run as, for example: `Specified`. Click **Next**.

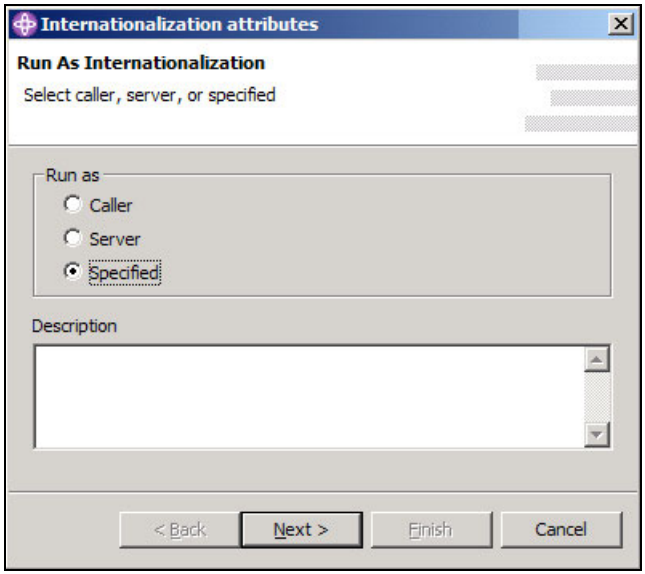


Figure 21-7 Run as for i18n

3. On the next panel, set the time zone ID, for example: `UTC`. You can use the drop-down list to select the value.

   Provide a description if you like.

   Click **Add** next to the locales.

4. In the Locale window, set the language and the country values, for example: `English, United States`. You can use the drop-down list to select the values. Click **OK**.

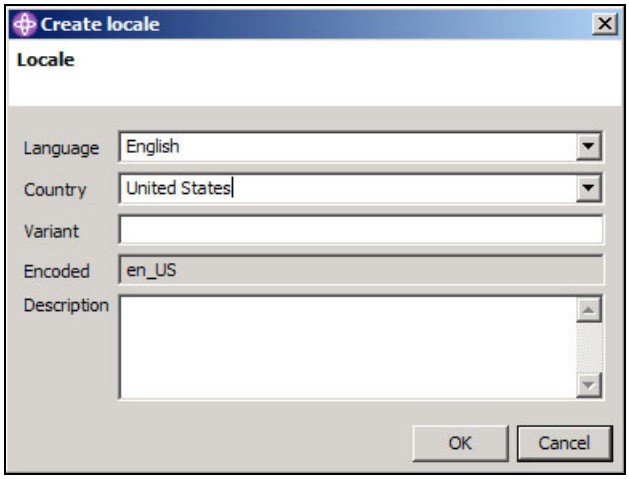


Figure 21-8 Locale settings for i18n

`en_US` should appear as a new locale as in Figure 21-9 on page 553.

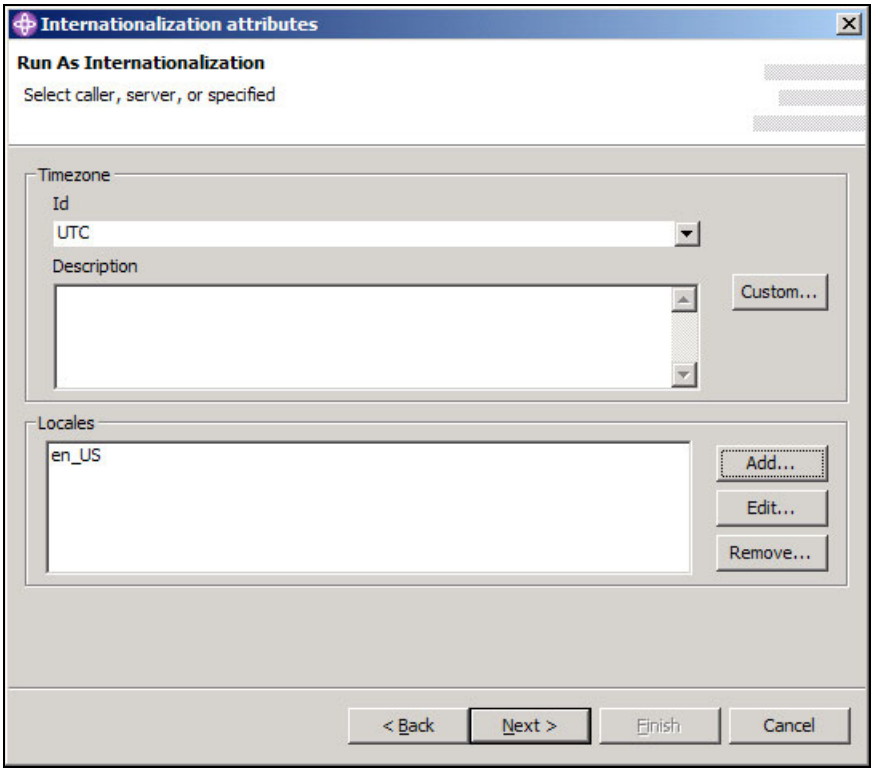


Figure 21-9 Application managed i18n settings for EJBs

Click **Next**.

5. Select the EJB on the Enterprise Beans panel, for example: **LogGreeting**. Click **Next**.
6. Select the method on the Method elements panel, for example: **logGreeting()** method.

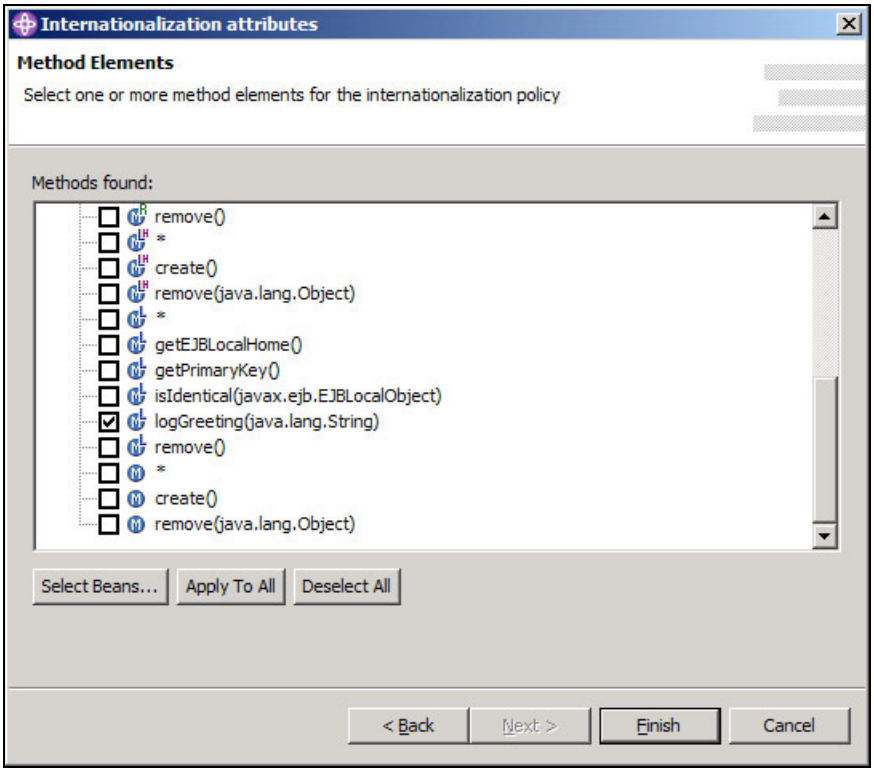


Figure 21-10

7. Click **Finish**.

21.4 Unit test

Unit testing internationalization is as simple as any other unit testing with WebSphere. You need to have an Integration Test server for your application and the application must be deployed.

What is difficult is that internationalization deals with different regions, languages and time zones. If you want to run proper testing, you have to change the time zone and locale settings for your client and for your server(s).

Note: You can change the locale for the operating system; that locale is propagated to the browser and then to the HTTP request. In this way, you can mime another location.

For the application server, you need to set up two custom properties for the process definition:

- *user.language*, for example: `en`
- *user.region*, for example: `US`

For a J2EE application client or a standalone Java application, you need to use two Java directives in the command line, for example:

```
java -Duser.language=en -Duser.region=US ...
```

The Internationalization service is not enabled by default. There are two ways to enable the i18n service in WebSphere Studio Application Developer Integration Edition.

- Enable the Administrative Console for the test server and follow the steps in 21.6, “Runtime environment” on page 556 to enable the service. In this case, you have to start and stop the application server.
- The other option is to “hack” the server configuration file when the server is stopped. This has the same effect as starting the server, enabling the i18n service, then starting the server.

  a. Open the server-pme.xml file for your test server. It is located at: **<*your_server_folder*>** → **<*your_server_name*.wsc>** → **cells** → **localhost** → **nodes** → **localhost** → **servers** → **server1**.

  b. Edit the XML file with an XML editor and change the *enable* attribute to `true` under: **pmeserver:PMEServerExtension** → **i18nService**.

  c. Save and close the file.

  d. Start the application server.

Once the server is running with i18n enabled, launch a Web browser and open the location: http://localhost:9080/i18nWeb/GreetingInput.html. You can change the greeting test if you like, then click **Send** on the page.

After a successful test, you should see results similar to the ones shown in Example 21-3 on page 556.

Example 21-3 Console output

```
SystemOut     O Sending greeting: Hello
SystemOut     O   caller locale: fi
SystemOut     O   invocation locale: hu_HU
SystemOut     O Received greeting: Hello
SystemOut     O   caller locale: hu_HU
SystemOut     O   invocation locale: pt_br
SystemOut     O Greeing is logged: Hello
SystemOut     O   caller locale: pt_br
SystemOut     O   invocation locale: en_US
```

The previous example output contains the following information. The browser was running with the region set to `Finland` when it called the servlet. The servlet made the call to the ReceiveGreeting EJB using the region `Hungary` and the language `Hungarian`.

When the ReceiveGreeting EJB receives the call, the caller was using the region `Hungary`, and the language `Hungarian`. The bean is going to use the region `Brazil`, and the language `Portuguese` to call another bean.

The LogGreeting EJB receives the call using region `Brazil` and language `Portuguese`. Even though the EJB is not going to call other components, the invocation is set to region `US`, language: `English`.

21.5 Assembly

Refer to 21.3, “Development” on page 542 for assembly details for WebSphere Studio Application Developer Integration Edition.

Assembly settings are available for the J2EE application components: servlets and EJBs. In both cases, you can define either application-managed i18n or container-managed i18n services.

21.6 Runtime environment

The Internationalization service is represented by a single flag in the runtime environment, which is either enabled or disabled. No further configuration is required.

The i18n service is not enabled by default; you have to enable it and restart the server with i18n enabled before you can run your application using the i18n service.

1. Make sure the application server is running, then launch the Administrative Console.
2. Navigate to **Servers** → **server1** → **Internationalization Service**.
3. Change the Startup flag as needed; the i18n service is enabled if the checkbox is selected.
4. Save the configuration for WebSphere.
5. Restart the application server.

Part 4

Appendixes

A

Appendix A.

Additional sample application configurations

This appendix provides additional help for the sample applications used in other chapters of the book.

You will find additional setup, configuration and development steps for the samples.

Project Interchange archive import/export

Throughout this redbook, you will find examples where we have provided sample code or solutions that you can use to follow the steps. In order to share this code, it is practical to make use of the Project Interchange plug-in, which enables quick and simple sharing of WebSphere Studio projects. The plug-in allows us to provide projects in a format that you can simply import into your own workspace.

Project Interchange is described in an IBM developerWorks WebSphere article located at:

http://www-106.ibm.com/developerworks/websphere/library/techarticles/0309_bergschacher/bergschacher.html#IDARTRDB

You can find the plug-in download and installation instructions there.

Tip: Note that for WebSphere Studio Application Developer Integration Edition V5.1, the installation directory defaults to:

C:\Program Files\IBM\WebSphere Studio\Application Developer IE\v5.1

Once this plug-in is added to your WebSphere Studio Application Developer Integration Edition, you will be able to make use of it through the **File** → **Import/Export** → **Project Interchange** menu system.

We also use the convention of naming zip files that contain project interchanges with the suffix .ic.zip.

HelloWorld process application

HelloWorld is a very simple business process (BPEL); it is only a supporting component for sample scenarios. The HelloWorld sample process is used in the Startup beans and Scheduler sample applications.

1. Open WebSphere Studio Application Developer Integration Edition with a new workspace, for example: C:\HelloWorld.Proj.
2. Open the Business Integration perspective.
3. Import the entire HelloWorldProcess.zip file using the Project Interchange plug-in.
4. Right-click the **HelloWorld.bpel** file under **Service Projects** → **HelloWorldProcess** → **com.helloworld.process**, then select **Enterprise Services** → **Generate Deploy Code...**.

5. In the Generate BPEL Deploy Code window, click **OK**; it will generate the service with EJB binding.
6. At the end of the code generation, you should only see three warnings listed in the Task view.

Integration Server V5.1 test environment setup

This short section shows how to set up the WebSphere Business Integration Server Foundation test environment in WebSphere Studio Application Developer Integration Edition.

1. Open the Server perspective.
2. Select **File** → **New** → **Server and Server Configuration...** from the menu.
3. Set the Server name: `HelloWorld` server, select **Integration Test Environment**, then click **Finish**.

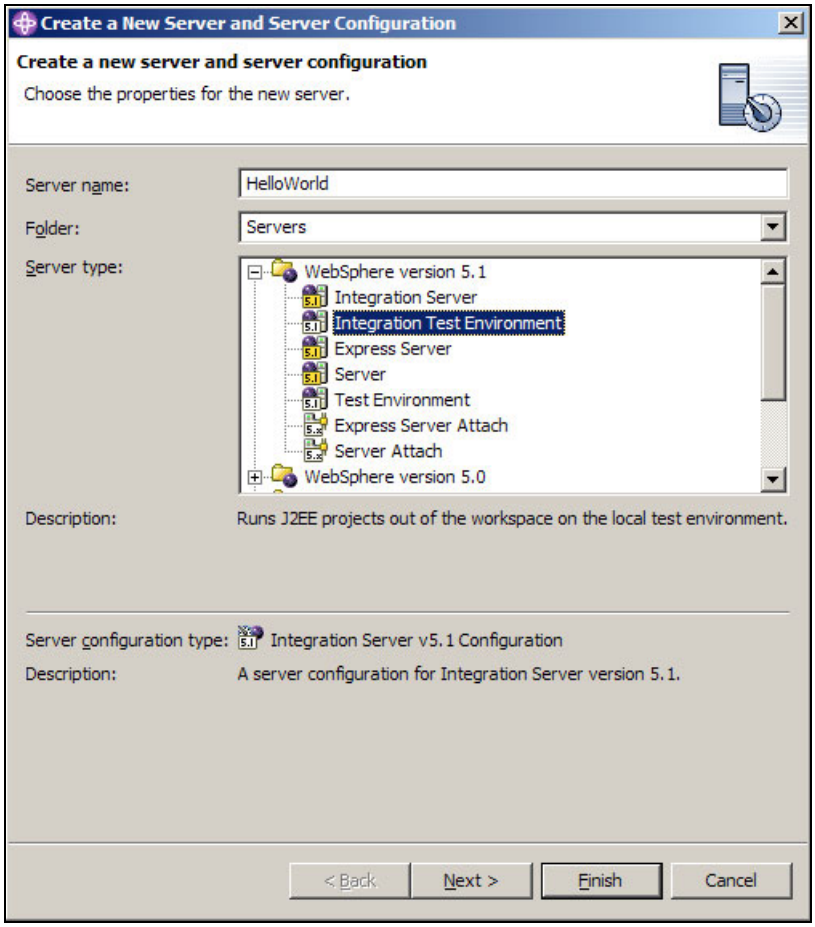


Figure A-1 Test Server configuration

The server is ready to start.

Adding the process to the test server

Once you have an Integration server, you can add a project to the server.

We are going to use the HelloWorld process application to add a process application to the test server.

1. Open the Server perspective.
2. Right-click the test server, for example **HelloWorld server**, then select **Add and remove projects...**.

3. Select the project that you want to add to the server on the left side, for example: **HelloWorldProcessEAR**, then click **Add >**.
4. Click **Finish**.
5. The server is ready to start; select the **Server** view, right-click the server item that you want to start, for example: **HelloWorld server**, then select **Start**.
6. Wait until you see the `... server is ready for e-business` message.

External service for the simple process

The NiceJourney application requires an external service to invoke. This section provides information about how to set up the external process in the test environment.

1. Open the J2EE perspective.
2. Select **File** → **Import** from the menu.
3. Select the **EAR file**, then click **Next**.
4. Select the **FlightBookingSystem.ear** file from the C:\Dev\ directory. Set the Project name to `bookyourflight.com`. Click **Next**.
5. Open the Server perspective.
6. Select **File** → **New** → **Server and Server Configuration** from the menu.
7. Set the server name to `bookyourflight.com` . Select the Server type by clicking **WebSphere version V5.1** → **Server**. We are going to use a simple WebSphere Application Server V5.1 test server. Click **Next**.

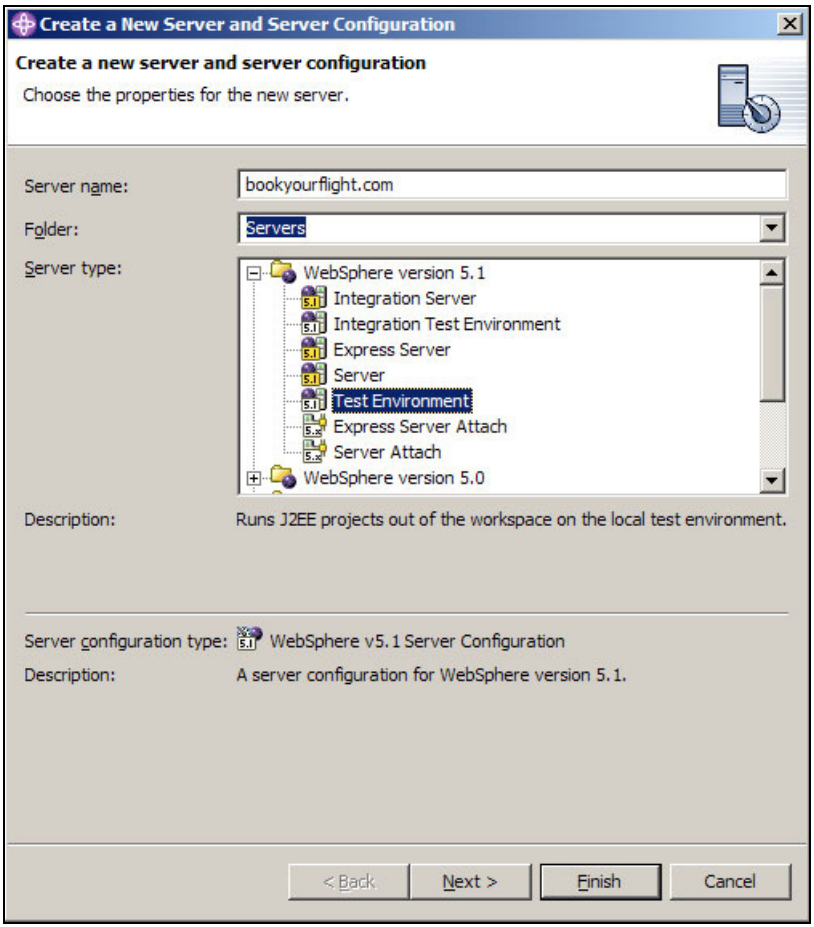


Figure A-2 Server wizard

8. On the WebSphere Server Configuration Settings panel, select **Use consecutive port numbers**, then set the First port number to 9080. Click **Finish**.

Note: By selecting the consecutive port numbers with the starting number 9080, the following port numbers (some of the most important for us) are dedicated to the application server:

- SOAP connector: 9080
- RMI: 9084
- HTTP port: 9085
- HTTPS port: 9086

You can find out the port numbers by opening the server configuration view for the bookyourflight.com server.

9. On the Server configuration view, right-click the **bookyourflight.com** item, then select **Add and remove projects...**.
10. On the new panel, select the **bookyourflight.com** project and click **Add** to add it to the server.

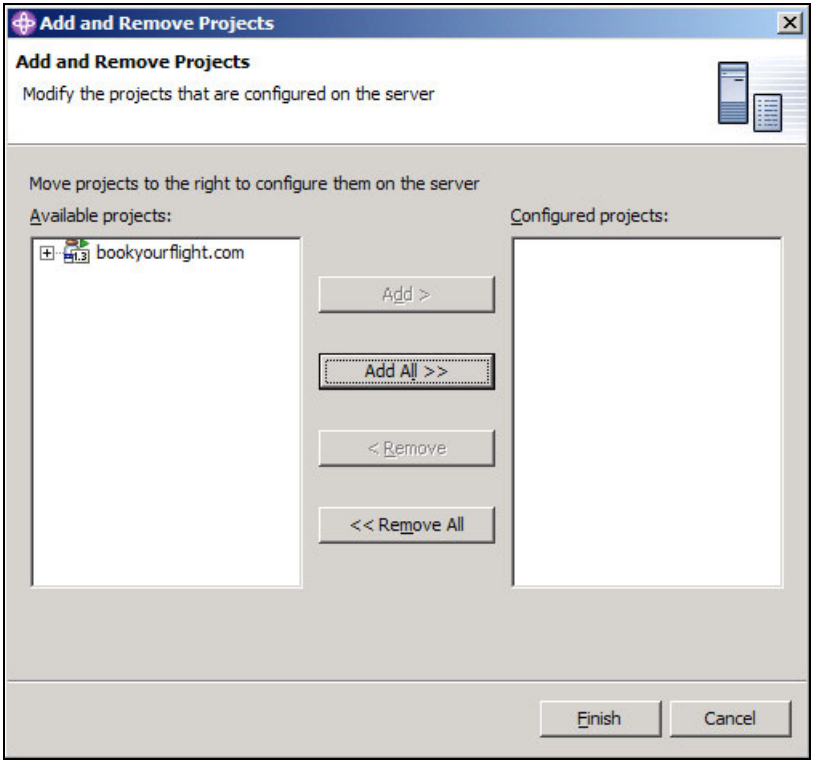


Figure A-3 Add project to the server

Click **Finish**.

11. Start the server by selecting the **bookyourflight.com** item in the Server view, then click the start icon (running man figure).
12. The Server view will change to the Console view automatically; wait until the server is started and the `... open for e-business` message appears on the Console view.

Building a stored procedure in DB2 for the CMP over Anything sample

The CMP over Anything sample in Chapter 16, “Container Managed Persistence over Anything” on page 435 requires a stored procedure in a DB2 database and some sample data in the tables. This section provides instructions about how to set up the database and create the stored procedure for the sample.

Software requirements for the development are as follows:

- DB2 application development client on Database server
- DB2 supported C/C++ compiler

Important: DB2 supports VisualAge for Java C++ compiler and Microsoft Visual C++ compiler. Since IBM stopped the development of the Visual Age product line, the only option is to get the Microsoft Visual C++ compiler. The Microsoft Visual C++ Toolkit 2003 is available as a free download from Microsoft at the following URL:

`http://msdn.microsoft.com/visualc/vctoolkit2003/`

Steps for building the stored procedure

The two basic steps for building SQL stored procedure are as follows:

1. Configure DB2 to identify installed C/C++ compiler
2. Configure DB2 to use the **`compile`** command in the installed compiler.

The supported C/C++ compilers for IBM DB2 on different platforms are as follows:

- AIX
  - IBM C for AIX Version 3.6.6 (Version 3.6.6.3, 4.0, 5.0 for 64-bit)
  - IBM C Set++ for AIX Version 3.6.6 (Version 3.6.6.3 for 64-bit)
  - IBM VisualAge C++ Version 4.0, 5.0 (32-bit and 64-bit)

- HP-UX
  - HP C Compiler Sersion A.11.00.03
  - HP aC++ Version A.03.25
- Linux
  - GNU/Linux gcc version egcs-2.91.66 (egcs-1.1.2 release)
  - GNU/Linux g++ version egcs-2.91.66 (egcs-1.1.2 release)
- Solaris
  - Forte/Workshop Compiler C Versions 4.2 (for 32-bit) and 5.0, 6.0 and 6.1 (for 32-bit and 64-bit)
  - Forte/Workshop Compiler C++ Version 4.2 (for 32-bit) and 5.0, 6.0 and 6.1 (for 32-bit and 64-bit)
- Windows
  - Microsoft Visual C++ Version 5.0 and 6.0 and newer
  - IBM VisualAge C++ for Windows Version 3.6.5, 4.2 and 5.0 7-8

Important: For the complete and latest list, refer to the IBM DB2 Web site at:

http://www-306.ibm.com/software/data/db2/

Since we do not have IBM VisualAge® C++ for Windows, for this redbook we used the Microsoft Visual C++ Toolkit 2003 (which is a freely available for developers).

Microsoft Visual C++ Toolkit 2003 was installed under the C:\Program Files\Microsoft Visual C++ Toolkit 2003 directory.

Note: You can also install MS Visual C++ under a different directory to avoid future problems with the space in the directory name.

Configuring DB2 to detect the MS Visual C++ compiler

MS Visual C++ comes with environment file vcvars32.bat under the installed root directory. When we tried to set up this file as “customized executable file for installed compiler” due to the space in the directory name, we were not able to set the compiler environment. So we copied *vcvars32.bat* to a different location (C:\MSVC) and were then able to set the environment with the following command in the DB2 command window:

```
db2set DB2_SQLROUTINE_COMPILER_PATH="c:\MSVC\vcvars32.bat"
```

Configuring DB2 to use the compile command in the installed compiler

The following is the default compile command for MS Visual C++.

```
db2set DB2_SQLROUTINE_COMPILE_COMMAND= "cl -Od -W2 /TC -D_X86_=1
-I%DB2PATH%\include SQLROUTINE_FILENAME.c /link -dll
-def:SQLROUTINE_FILENAME.def /out:SQLROUTINE_FILENAME.dll
%DB2PATH%\lib\db2api.lib"
```

Important: The above configuration works with a Windows platform with Microsoft Visual C++ compiler only. For other platforms and other C/C++ compilers, please refer to the IBM DB2 Web site.

Once the C/C++ compiler has been installed and configured, you are ready to create the SQL stored procedure for the application.

Creating an SQL stored procedure

Important: We need to create the database and the tables before creating the SQL stored procedure. The database and tables can be created in many ways. The simpliest way is to get the .ddl from the EJB project. Create a database called ITSO_CMP, then run the command: **`db2 -tvf table.ddl`**.

1. Click **Start** → **IBM DB2** → **Development Tools** → **Development Center**; this will open the launch pad.
2. Click **Create a Project**.
3. Select the **New** tab and provide the Project Name: `CMPAproject` and Project path: `C:\Projects\CMPAproject`. Click **OK**.

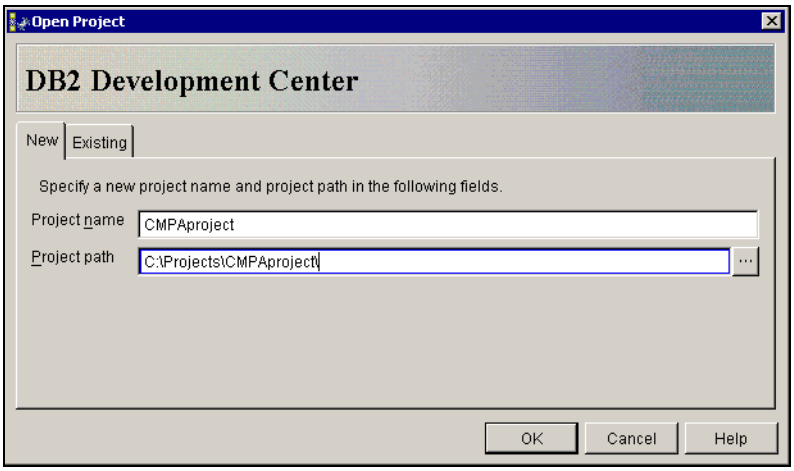


Figure A-4 Create a new DB2 project

4. Click **Add Connection**, then provide the details for the online connection type, as shown in Figure A-8 on page 573.

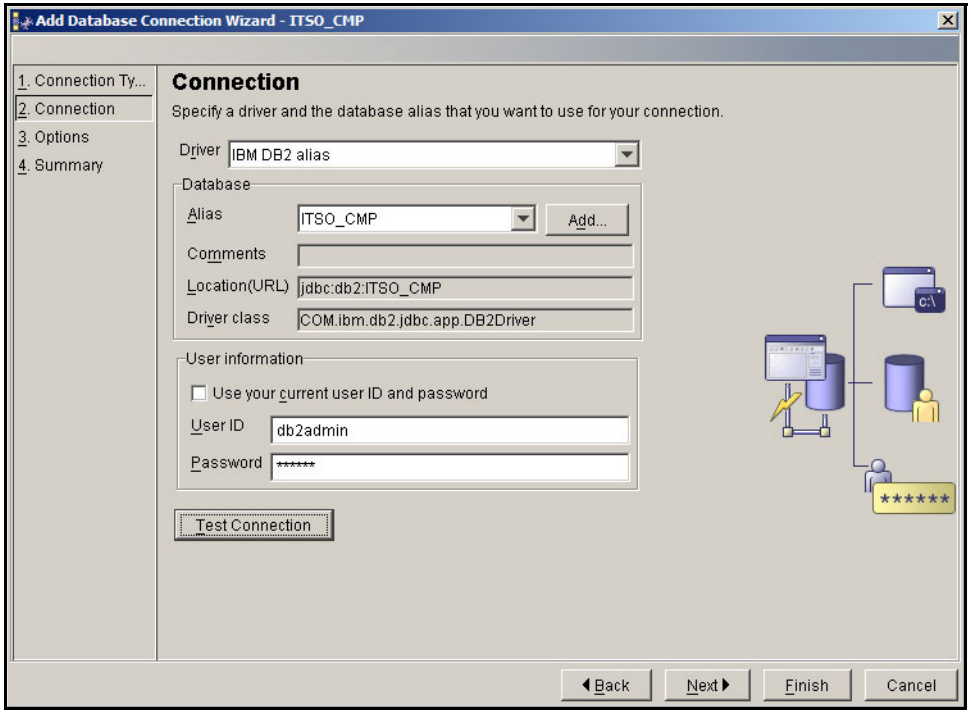


Figure A-5 Connection details

The connection can be validated by clicking the **Test Connection** button.

You can specify the SQL schema name, the package owner and build owner. These can also be changed or set later.

When you are finished, click **Finish**.

5. Click **Create Object** to create a new stored procedure.

   Select **Stored Procedure** and **SQL** for the object to be created.

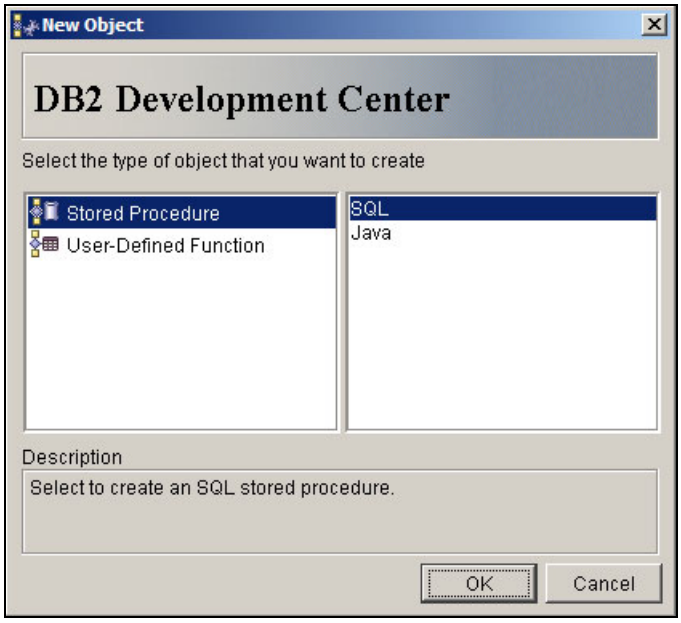


Figure A-6 Object type settings

6. A new dialog opens. Provide the stored procedure name, which wil be called from the push-down method later. The stored procedure name is `ITSO.ACCOUNT_SP`.

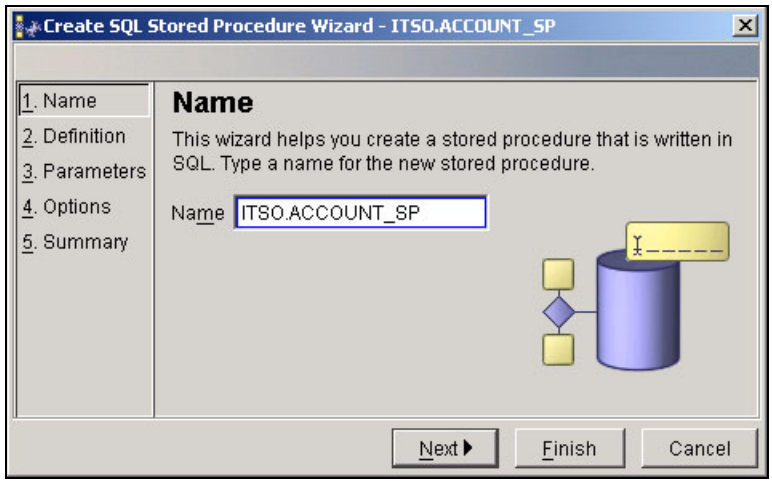


Figure A-7 Stored procedure name

Click **Next**. This will take you to the panel where you can define the SQL statement.

7. Change the default SQL statement by clicking the **...** button under the value.

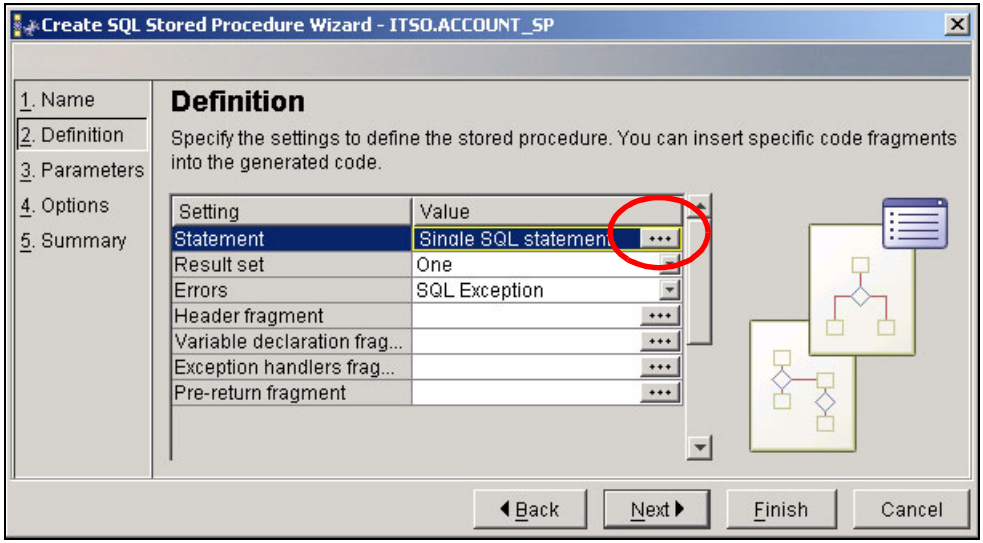


Figure A-8 SQL definition

8. Click **SQL Assist**.

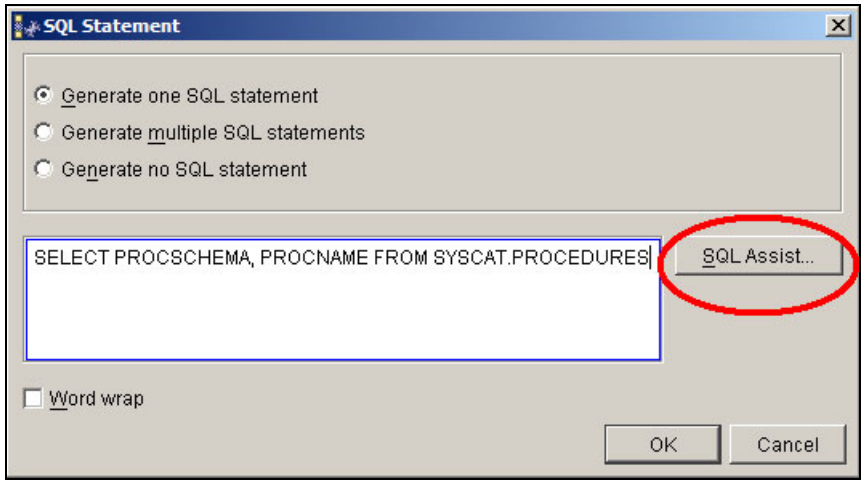


Figure A-9 SQL statement editor

This will open the SQL Assist window where you can create an SQL stored procedure. First, remove the default SQL statement by clicking **Clear** under the SQL code panel.

In the outline section:

a. Select **FROM (Source tables)** under the `SELECT` statement. This will show the tables available under the Details pane. Expand the **ITSO_CMP** table and select **EJB.ACCOUNT_CMP** table. click the **>** button, so that the selected source table will be added to the SQL statement.

b. Select **SELECT (Result columns)** and expand the **EJB.ACCOUNT_CMP** table under the Details pane. Select **BALANCE** and click **>**. Click **...** under the Result columns next to the `BALANCE` item.

   A new window appears; click **Clear** at the bottom. Double-click the **SUM** function, then select **ACCOUNT_CMP.BALANCE** in the new panel. Click **OK**. Click **OK** again to close the Expression builder window.

c. Select the **WHERE (Row filter)** item. Under the Details pane, select **EJB.ACCOUNT_CMP.CUSTNO**; in the column, leave the operator as = and select the value **Host variable...**. This will open the variable window. Enter the variable name as `CustNo`. Click **OK** and add it to the SQL statement by clicking **>**.

   The SQL code window should have the code shown below:

```
SELECT SUM(ACCOUNT_CMP.BALANCE) FROM EJB.ACCOUNT_CMP AS ACCOUNT_CMP
WHERE ACCOUNT_CMP.CUSTNO = ACCOUNT_SP.CustNo
```

d. Click **OK** twice to get back to the Definition panel.

e. Click **Next**.

f. Add an out parameter by clicking **Add**.

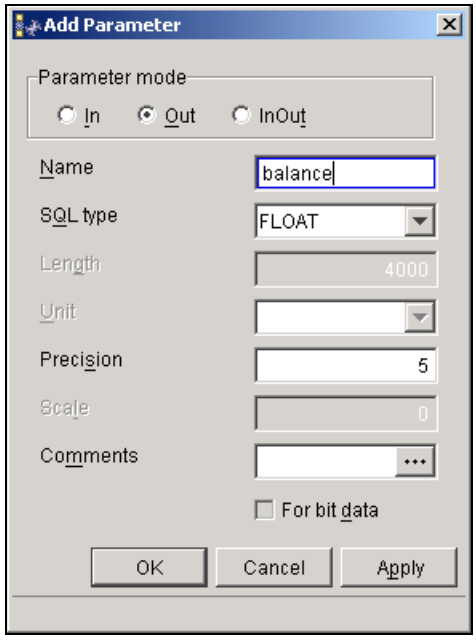


Figure A-10 Add an out parameter to the stored procedure

g. Click **Finish** to finish creating the SQL stored procedure.

9. The stored procedure is built and deployed automatically when you use the wizard to generate the code for you. At this point in time, the code is not ready yet, however.

10. Edit the stored procedure by right-clicking the **ITSO.ACCOUNT_SP** stored procedure in the DB2 Development Center from the hierarchy then selecting **Edit**. Insert the following line at the end of the source code after the `SET balance = balance_TMP;` line.

```
FETCH cursor1 INTO balance;
```

The code will put the result value into the out variable.

11. Save the source under the Editor View and close the Editor View window.

12. Remove the previously deployed code from the database by right-clicking the **ITSO.ACCOUNT_SP** stored procedure in the DB2 Development Center from the hierarchy then selecting **Drop From Database**. Click **Yes** to drop the procedure.

13. Build the code again by right-clicking the **ITSO.ACCOUNT_SP** stored procedure in the DB2 Development Center from the hierarchy then selecting **Build**. The build also deploys the stored procedure automatically.

14. Test the stored procedure by right-clicking the **ITSO.ACCOUNT_SP** stored procedure in the DB2 Development Center from the hierarchy then selecting **Run...**.

    Set the *CustNo* value to 1, then click **OK**.

    Check the output view for results; it should return 0 for an empty database.

    Note: You can fill up the account table with some data for testing purposes using the following commands in a DB2 command window:

    ```
    db2 CONNECT TO ITSO_CMP
    db2 INSERT INTO EJB.ACCOUNT_CMP (ACCOUNT, CUSTNO, NAME, BALANCE)
    VALUES (11, 1, 'John Doe', 10)
    db2 INSERT INTO EJB.ACCOUNT_CMP (ACCOUNT, CUSTNO, NAME, BALANCE)
    VALUES (12, 1, 'John Doe', 20)
    db2 INSERT INTO EJB.ACCOUNT_CMP (ACCOUNT, CUSTNO, NAME, BALANCE)
    VALUES (13, 1, 'John Doe', 30)
    db2 INSERT INTO EJB.ACCOUNT_CMP (ACCOUNT, CUSTNO, NAME, BALANCE)
    VALUES (21, 2, 'Mary Jane', 40)
    db2 DISCONNECT CURRENT
    ```

    If you run the test on the stored procedure again, it should show the result 60.

15. You can close the DB2 Development Center; save the project when the application asks for confirmation.

B

Appendix B

Additional configuration help

This appendix provides additional information for Chapter 4, “Runtime environment” on page 31. While the runtime environment chapter only focuses on WebSphere Business Integration Server Foundation installation and configuration, this appendix provides additional information to set up and configure the following additional components.

- “WebSphere MQ setup instructions” on page 578
- “DB2 Enterprise Server Edition V8.1 installation” on page 579
- “Tivoli Directory Server V5.2 installation” on page 580
- “IBM HTTP Server, IBM HTTP Web server plug-in, and Tivoli Performance Viewer installation” on page 582
- “Creating a WebSphere cluster” on page 583

WebSphere MQ setup instructions

The following section outlines instructions for installing WebSphere MQ on a single node.

Installation

1. Run the Setup.exe under the MQ Server installation image directory.
2. In the menu that comes up, click **Supported Java Runtime environment 1.3 or later**.
3. Install the JDK provided on WebSphere MQ CD by clicking **WebSphere MQ CD**.
   - We chose to install JDK in C:\WebSphere\Java14.
   - Choose **custom installation** and deselect Sources.
   - Do not install this JDK as a System JVM.
4. Verify that all prerequisites have been met in the setup window. Pay special attention to the Network Prerequisites. You may want to install MQ under a user on the local machine that has administrator authority.
5. Under the WebSphere MQ Setup menu, click the **Launch MQ Installer** button to start the installation process.
6. Install under C:\WebSphere\WMQ\.
7. Under the Features window, make sure all components are selected.
8. Review your selections and click the **Install** button to begin installation.

Configuration

1. When the configuration starts, under the Domain Controller window, select **No domain controller**. You should verify that there is no domain controller in your environment.
2. Next, skip Default Configuration to set up. This basically creates a queue manager and makes it part of a cluster. We will create our own queue manager and queues later on.
3. In the last page, deselect everything and close the configuration.

Fixpack installation

1. Download WMQ CSD5, file name U200202A.exe, from the WebSphere MQ Support site.
2. Stop WebSphere MQ from the Windows Services. This ensures that all queue managers are stopped.
3. Run the U200202A.exe, and specify `C:\WebSphere\WMQCSD06`.

4. Change the default back-up folder to C:\WebSphere\WMQ_FPBK, and click the **Install** button to proceed with Fixpack installation.

If you get an error window stating that WebSphere MQ files are in use, make sure to stop all WebSphere MQ processes from the Windows services. Make sure that you are not viewing the WebSphere MQ Help. Also verify that the back-up directory is not under the WebSphere MQ installation folder.

DB2 Enterprise Server Edition V8.1 installation

DB2 must be set up prior to installing TDS because it requires/assumes the existence of DB2 FP2. It is possible to install DB2 FP2 as part of installing TDS but the decision was made to install DB2 and FP5 prior to installing TDS.NC

Installation

1. Download the DB2 Fixpack 5 binary from DB2 support site. The name of the executable is FP5_WR21334_ESE.exe. This FP includes a full version of DB2 that can be installed as part of the FP 5 installation. This is typical for Windows platforms only. For a UNIX® installation, we will install DB2 8.1.
2. Run the executable and select **DB2 8.1 Enterprise Edition**.
3. Select **Custom**.
4. Under Feature, select all features. Change the directory location to C:\WebSphere\SQLLIB.
5. Follow on-screen instructions to install DB2.
6. Only select the **English** language.
7. In the User window, leave `db2admin` as the user. Make sure no Domain is specified. Choose `db2admin` as the password.
8. Under Administrative contact, select the defaults.
9. Make sure **Create the DB2 instance** is checked in the next window.
10. Under Configure DB2 instances, select the defaults.
11. Select **Do not prepare any metadata**.
12. Defer contact for health monitoring.
13. Request satellite information; select **defer to later time**.
14. Review the summary and click the **Install** button to proceed with DB2 installation.

Fixpack installation

Fixpack 5 will be automatically installed as part of the setup instructions above. This is unique to the Windows environment. For a UNIX setup, a separate installation of FP5 is needed.

Tivoli Directory Server V5.2 installation

The following section discusses Tivoli Directory Server V5.2 setup instructions.

Installation

1. Make sure to check the PATH and CLASSPATH variables so as not to include any references to a system-wide JDK.
2. Check the prerequisites and verify all the setup requirements. This information is given in the ldapinst.html which should be located under the Tivoli Directory Server installation folder.
3. Run ismp\setup.exe to start the installation.
4. Select the **English** language; the Tivoli Directory Server Welcome window should appear.
5. Select **English** as the language.
6. Also select **WebSphere Application Embedded server**.
7. Review the summary and start the installation.
8. Wait for the machine to reboot.

Post-installation configurations

Tivoli Directory Server requires a supporting application server (WebSphere Application Server V5 Express) to administer the LDAP server.

There might be cases when a second application server introduces problems in the system configuration. One example is a single machine configuration for WebSphere Business Integration Server Foundation; the LDAP application server collides with the WebSphere Business Integration Server Foundation application server.

This section provides additional information about how to configure the LDAP application server manually, so there will be no port collision with the WebSphere Business Integration Server Foundation application server.

We assume that Tivoli Directory Server has been installed with WebSphere Application Server V5 Express and the Web management application.

WebSphere Application Server V5 Express does not have the GUI version Administrative Console, so we cannot use that to reconfigure the ports used by the application server.

1. Open the serverindex.xml file in a text editor, from the <TDS_root>\appsrv\config\cells\DefaultNode\nodes\DefaultNode directory.

   Edit the file and replace the port numbers as listed below:

   – BOOTSTRAP_ADDRESS: port=2810
   – SOAP_CONNECTOR_ADDRESS: port=8881
   – DRS_CLIENT_ADDRESS: port=0
   – JMSSERVER_QUEUED_ADDRESS: port=0
   – JMSSERVER_DIRECT_ADDRESS: port=0

2. Open the **server.xml** file in a text editor, from the <TDS_root>\appsrv\config\cells\DefaultNode\nodes\DefaultNode\servers\server1 directory.

   Edit the file and replace the port numbers as listed below:

   – HTTPTransport_1: port=9081 (original value: 9080)
   – HTTPTransport_2: port=9444 (original value: 9443)
   – HTTPTransport_3: port=9091 (original value: 9090)
   – HTTPTransport_4: port=9044 (original value: 9043)

3. Open the virtualhosts.xml file in a text editor, from the <TDS_root>\appsrv\config\cells\DefaultNode directory.

   Edit the file and replace the port numbers as listed below:

   – HostAlias_1: port=9081
   – HostAlias_3: port=9444
   – HostAlias_4: port=9091
   – HostAlias_5: port=9044

4. Open a command prompt (or terminal) window and go to the directory <TDS_root>\appsrv\bin.

5. Issue the following command: **`startserver server1`** or **`./startServer.sh server1`**

6. After a couple of minutes, depending on your system's speed, you should see a message stating that the server was started successfully.

7. If you have another application server installed, you may want to try to start it also to make sure that it is running without any problems.

Configuration

1. The IBM TDS Configuration Tool will appear when the computer restarts.
2. In the Tools window, on the right-hand menu, select an administrator DN, for example: `cn=root`, and set the password, for example: `tdsadmin`.
3. Click **Configure database** on the right-hand menu, and select **Create a new database**.
4. For the password, enter the administrator user ID and password; we used `db2admin` for the user name and `db2admin` for the password.
5. Enter the name of the database as `TDSDB`.
6. Select the first option in the next window, **Create a universal DB2 database (UTF-8/UCS-2)**.
7. Select **C:** as the database location.
8. Review the summary and click **Finish** to create the database.
9. Once the creation of the database is complete, close the window.

IBM HTTP Server, IBM HTTP Web server plug-in, and Tivoli Performance Viewer installation

Follow the steps outlined below to install IBM HTTP Server, the Web server plug-in and the Tivoli Performance Viewer.

1. Change to the WebSphere Application Server directory under the WebSphere Business Integration Server Foundation install image.
2. Run the install utility, setup.exe, and deselect the application server.
3. After accepting the license agreement, select **Add features to the existing copy** in the next window. Make sure that the PATH points to an existing WebSphere Business Integration Server Foundation installation.
4. Select only the following components:
   - IBM HTTP Server V1.3
   - Web server plug-ins for IBM HTTP Server V1.3
   - Tivoli Performance Viewer
   - Dynamic Cache Monitor
   - Performance Servlet

   Note that some components may already be installed and these components will be grayed out.
5. Change the location of the IBM HTTP Server installation directory.

6. Specify the Administrator password to run IBM HTTP Server as a background Windows service.
7. Review the summary and click **Finish** to install the additional components.

Creating a WebSphere cluster

These instructions provide details of configuration steps required to create a clustered WebSphere Application Server or WebSphere Business Integration Server Foundation environment.

Federating nodes

This task entails adding the WebSphere Business Integration Server Foundation V5.1 nodes to the WebSphere Application Server Network Deployment server. Perform the following steps on each WebSphere Business Integration Server Foundation V5.1 node:

1. Change to the /usr/WebSphere/AppServer/bin directory.
2. Issue the following command to federate WebSphere Business Integration Server Foundation node to Deployment Manager.

   ```
   ./addNode.sh <dmgr_hostname>
   ```

   The output from this command should look like the following example.

Example: B-1 addNode.sh output

```
# ./addNode.sh m10df56f
ADMU0116I: Tool information is being logged in file
           /usr/WebSphere/AppServer/logs/addNode.log
ADMU0001I: Begin federation of node m10df4ff with Deployment Manager at
           m10df56f:8879.
ADMU0009I: Successfully connected to Deployment Manager Server: m10df56f:8879
ADMU0505I: Servers found in configuration:
ADMU0506I: Server name: server1
ADMU2010I: Stopping all server processes for node m10df4ff
ADMU0512I: Server server1 cannot be reached. It appears to be stopped.
ADMU0024I: Deleting the old backup directory.
ADMU0015I: Backing up the original cell repository.
ADMU0012I: Creating Node Agent configuration for node: m10df4ff
ADMU0014I: Adding node m10df4ff configuration to cell: m10df56fNetwork
ADMU0016I: Synchronizing configuration between node and cell.
ADMU0018I: Launching Node Agent process for node: m10df4ff
ADMU0020I: Reading configuration for Node Agent process: nodeagent
ADMU0022I: Node Agent launched. Waiting for initialization status.
ADMU0030I: Node Agent initialization completed successfully. Process id is:
           11426
```

```
ADMU0524I: WebSphere Embedded Messaging support not installed; Queue Manager
           not created
ADMU9990I:
ADMU0300I: Congratulations! Your node m10df4ff has been successfully
           incorporated into the m10df56fNetwork cell.
ADMU9990I:
ADMU0306I: Be aware:
ADMU0302I: Any cell-level documents from the standalone m10df4ff configuration
           have not been migrated to the new cell.
ADMU0307I: You might want to:
ADMU0303I: Update the configuration on the m10df56fNetwork Deployment Manager
           with values from the old cell-level documents.
ADMU9990I:
ADMU0306I: Be aware:
ADMU0304I: Because -includeapps was not specified, applications installed on
           the standalone node were not installed on the new cell.
ADMU0307I: You might want to:
ADMU0305I: Install applications onto the m10df56fNetwork cell using wsadmin
           $AdminApp or the Administrative Console.
ADMU9990I:
ADMU0003I: Node m10df4ff has been successfully federated.
```

Creating clusters and cluster members

The following guidelines demonstrate how to configure a business process container in a cluster:

1. Log in to the WebSphere Application Server Administrative Console on the Network Deployment machine.
2. Verify that all nodes are running and are in a synchronized state. Also verify that all node agents are up and running. This is illustrated in Figure B-1 on page 585 and Figure B-2 on page 585.

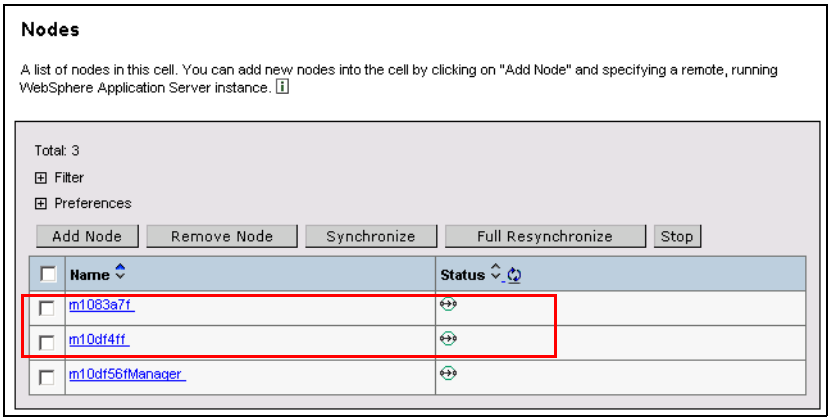


Figure B-1 List of nodes and statuses

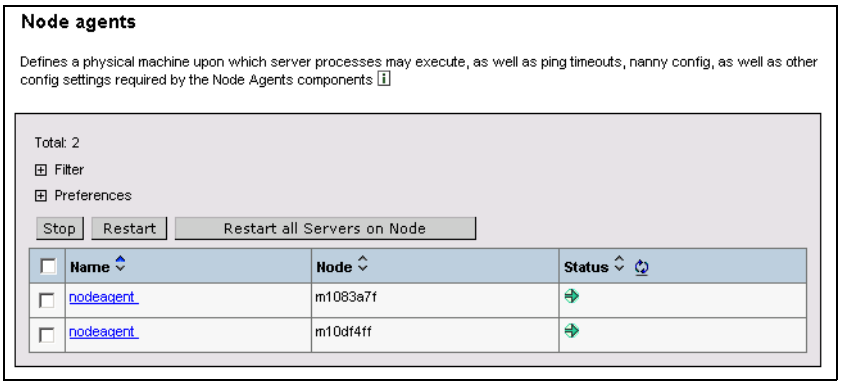


Figure B-2 List of node agents and statuses

Note: No applications servers exist prior to creating the cluster. Application servers will be created and deployed on individual nodes as part of cluster creation.

3. Click **Cluster** and **New** to create a new cluster.
4. Specify the name of the cluster and add existing application servers to this cluster. Application servers (or cluster members) will be created in the next step. Verify that all the replication options are checked as indicated in Figure B-3 on page 586. Click **Next**.

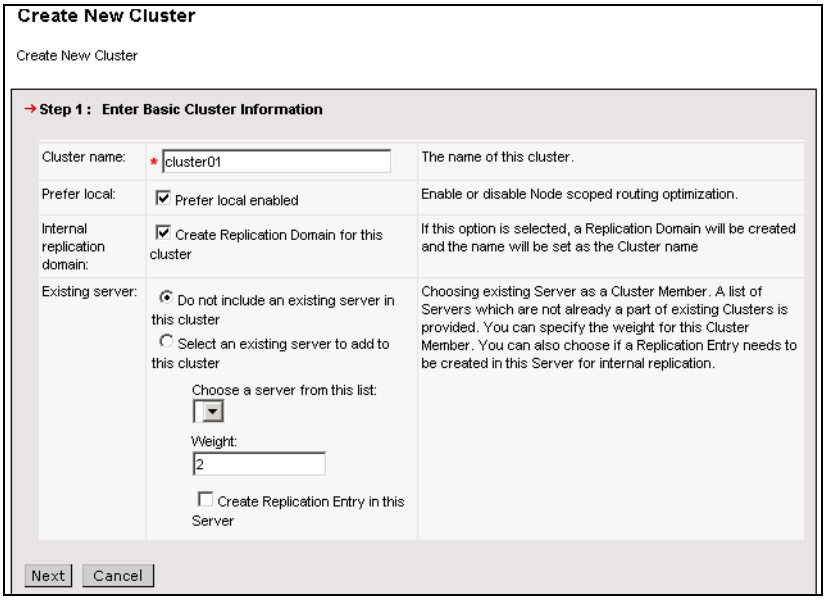


Figure B-3 Create a new cluster

5. In the next window (step 2), choose a name for the cluster member (WebSphere Business Integration Server Foundation application server). Also select a node where this server will be deployed as part of the cluster. Accept defaults for the remaining options. Make sure to select **Create Replication Entry** in this Server option. Select the default application **server1** template. Click **Apply** to create the cluster member. Do not click **Next** yet, since we are going to add more cluster members.

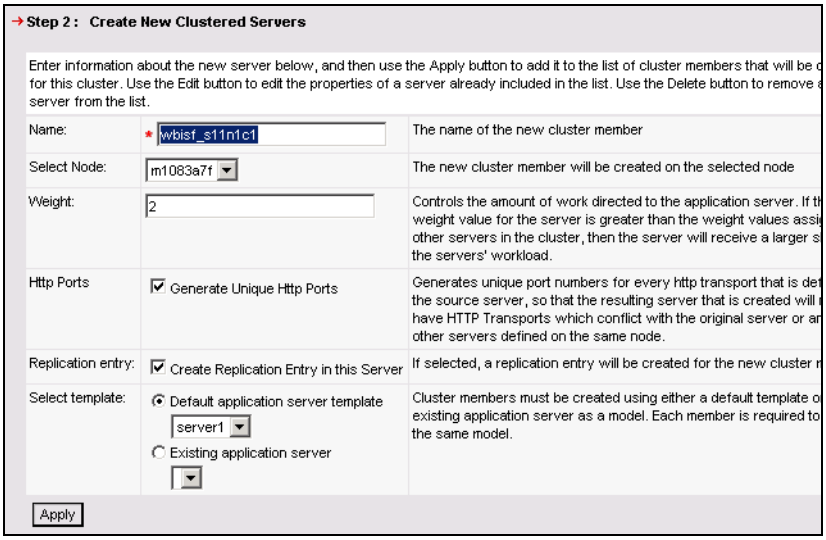


Figure B-4 Create New Clustered Servers

6. Create another cluster member by filling out appropriate entries as indicated. Click **Apply** to create the new cluster member. The process is shown in Figure B-5 on page 588.

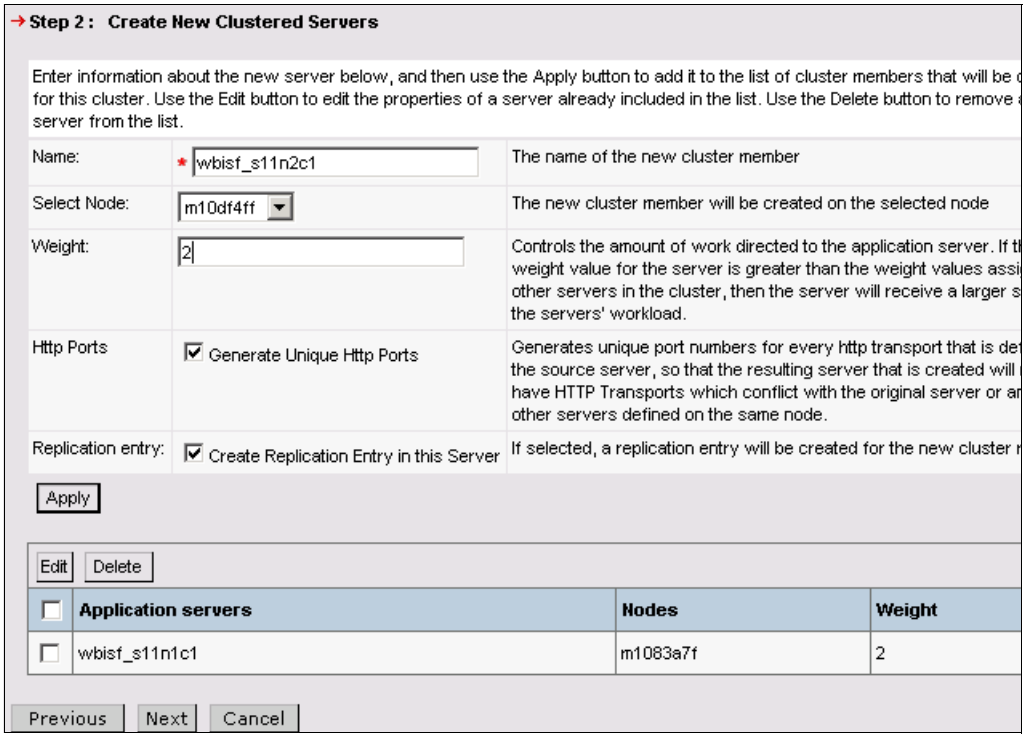


Figure B-5 Create New Clustered Serves

7. Once each node has a cluster member (application server), as indicated in the figure below, click **Next** to go to Step 3.

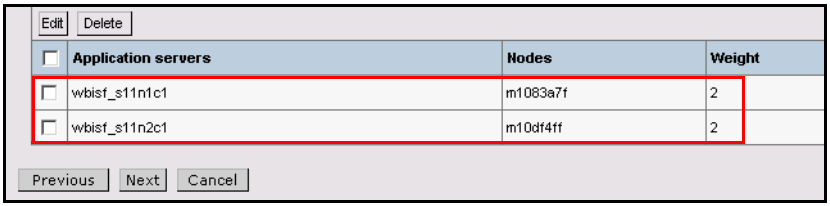


Figure B-6 Cluster members

8. Review the summary information and click **Finish** to create the cluster and the cluster members.

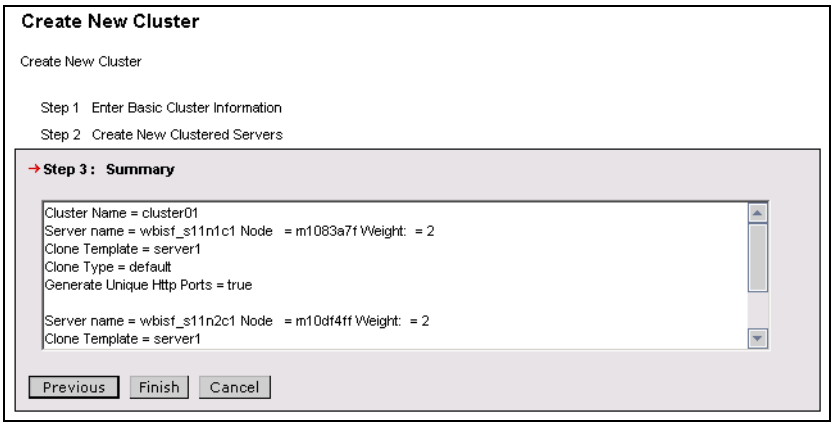


Figure B-7 Create New Cluster summary

9. The cluster will be created and the status will be set to an unknown state, as indicated in the figure below. Make sure to save the changes by clicking the **Save** link at the top of the screen.

10. During the save operation, make sure to select the **Synchronize changes with Nodes** option, as indicated in Figure B-8.

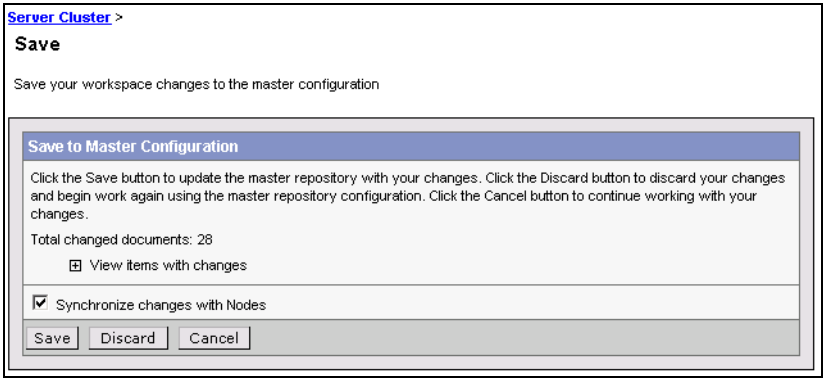


Figure B-8 Save WebSphere configuration with Synchronization enabled

11. Once the save operation finishes, the main Administrative Console window will be displayed. Verify that all nodes are in the synchronized state by navigating to **System Administration** → **Nodes**.

12. Verify that the cluster state is currently set to `stop` by navigating to **Servers** → **Clusters**.

This concludes the tasks necessary to create a WebSpher cluster and WebSphere Business Integration Server Foundation cluster members.

C

Appendix C.

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG246318.zip

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246318.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

File name	*Description*
SG246318.zip	Zipped code samples

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	20MB minimum
Operating System:	Windows 2000 Server
Processor:	Pentium® 4, 1GHz minimum
Memory:	512MB minimum

How to use the Web material

Create a directory on your workstation, for example: C:\redbooks\SG246318, and unzip the contents of the Web material zip file into this folder.

Abbreviations and acronyms

1PC	One-phase commit
2PC	Two-phase commit
AAT	Application Assembly Tool
ACID	Atomicity, Consistency, Isolation, Durability
AMC	Application / Module / Component
AMI	Application Managed Internationalization
API	Application Programming Interface
BMP	Bean Managed Persistency
BPE	Business Process Engine
BPEL4WS	Business Process Execution Language for Web Services
CEI	Common Event Infrastructure
CMI	Container Managed Internationalization
CMP	Container Managed Persistency
CMR	Container Managed Relationship
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CSS	Cascading Style Sheet
DD	Deployment Descriptor
DMZ	Demilitarized Zone
EAR	Enterprise Archive
EIS	Enterprise Information Systems
EJB	Enterprise JavaBeans
EJB QL	EJB Query Language
EM	Extended Messaging
FAR	Flow Archive
FDML	Flow Definition Markup Language
FP	FixPack
GMT	Greenwich Mean Time
GUI	Graphical User Interface
HA	High Availability
HACMP™	High Availability Cluster Multi-Processing
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines Corporation
IE	Integration Edition (WebSphere Studio)
ITSO	International Technical Support Organization
J2EE	Java 2 Enterprise Edition
JAR	Java Archive
JAX-RPC	Java API for XML-based RPC
JCA	Java Connector Architecture
JDBC	Java Database Connection
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JSP	JavaServer Pages
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
MDB	Message-Driven Bean
MVC	Model-View-Controller
ND	Network Deployment
NLS	National Language Support

OASIS	Organization for the Advancement of Structured Information Standards
ORB	Object Request Broker
OS	Operating System
PME	Programming Model Extension
PMI	Performance Monitoring Interface
QL	Query Language
QoS	Quality of Service
RDB	Relational Database
RMI/IIOP	Remote Method Invocation / Internet Inter-ORB Protocol
RRA	Relational Resource Adapter
SDK	Software Development Kit
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSS	Staff Support Service
UDDI	Universal Description, Discovery and Integration
WAR	Web Archive
WIM	Work Item Manager
WSDL	Web Services Description Language
WSFL	Web Services Flow Language
WSIF	Web Services Invocation Framework
XML	Extended Markup Language
XSLT	Extensible Stylesheet Language Transformations

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 598. Note that some of the documents referenced here may be available in softcopy only.

- *Patterns: Serial and Parallel Processes for Process Choreography and Workflow*, SG24-6306
- *IBM WebSphere Application Server V5.1 System Management and Configuration WebSphere Handbook Series*, SG24-6195
- *IBM WebSphere V5.1 Performance, Scalability, and High Availability WebSphere Handbook Series*, SG24-6198
- *WebSphere Application Server and WebSphere MQ Family Integration*, SG24-6878
- *Exploring WebSphere Studio Application Developer Integration Edition V5*, SG24-6200
- *A Practical Guide to the IBM Autonomic Computing Toolkit, IBM Redbook*, SG24-6635
- *WebSphere Application Server Enterprise V5 and Programming Model Extensions*, SG24-6932

Online resources

These Web sites and URLs are also relevant as further information sources:

- WebSphere Business Integration Web site

  `http://www.ibm.com/websphere/integration`

- WebSphere Business Integration Adapters Web site

  `http://www.ibm.com/websphere/integration/wbiadapters/`

- WebSphere Business Integration Server Foundation Web site

  `http://www-306.ibm.com/software/integration/wbisf/requirements/`

- WebSphere Studio Application Developer Integration Edition requirements

  http://www-306.ibm.com/software/integration/wsadie/requirements/
- WebSphere Business Integration Server Foundation requirements

  http://www.ibm.com/software/integration/wbisf/reqirements/
- WebSphere Business Integration Server Foundation support

  http://www.ibm.com/software/integration/wbisf/support/
- IBM AIX Fix central

  https://techsupport.services.ibm.com/server/aix.fdc
- WebSphere MQ supported platforms

  http://www.ibm.com/software/integration/mqfamily/platforms/supported/wsmq_for_aix_5_3.html
- WebSphere Business Integration products - platform support

  http://www.ibm.com/software/integration/websphere/mqplatforms/supported.html
- Tivoli Directory Server supported platforms

  http://www.ibm.com/software/tivoli/products/directory-server/platforms.html
- IBM DB2 system requirements

  http://www-306.ibm.com/software/data/db2/udb/sysreqs.html
- WebSphere MQ Books and Manuals

  http://www.ibm.com/software/ts/mqseries/library/manualsa/index.htm
- IBM DB2 Information center

  http://publib.boulder.ibm.com/infocenter/db2help/index.jsp
- WebSphere MQ Queue Manager Clusters documentation

  http://publibfp.boulder.ibm.com/epubs/html/csqzah06/csqzah06tfrm.htm
- WebSphere Application Server V5: *Using WebSphere and WebSphere MQ clustering*

  http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/tips0224.html?Open
- WebSphere Application Server Network Deployment Information Center

  http://publib.boulder.ibm.com/infocenter/ws51help/topic/com.ibm.websphere.nd.doc/info/ae/ae/tins_install.html
- WebSphere Application Server support

  http://www.ibm.com/software/webservers/appserv/was/support/

- WebSphere Business Integration Server Foundation Process Choreographer library

  http://www.ibm.com/developerworks/websphere/zones/was/wpc.html

- WebSphere Application Server V5.1 Information Center

  http://publib.boulder.ibm.com/infocenter/ws51help/index.jsp

- IBM developerWorks: Web services programming tips and tricks: Roundtrip issues, an introduction

  http://www-106.ibm.com/developerworks/webservices/library/ws-tip-roundtrip1.html

- IBM developerWorks: *Customizing WebSphere Application Server Enterprise Process Choreographer Web Client,* Part 1: “Adapting the look and feel”

  http://www-106.ibm.com/developerworks/websphere/library/techarticles/wasid/WPC_Client1/WPC_Client1.html

- Specification: Common Base Event

  http://www.ibm.com/developerworks/webservices/library/ws-cbe/

- IBM developerWorks: Autonomic computing

  http://www.ibm.com/developerworks/autonomic/

- IBM developerWorks: *On the road to self-healing computing systems: Standardizing product logs*

  http://www.ibm.com/developerworks/rational/library/2084.html

- IBM WebSphere Developer Technical Journal: *Creating Extended Messaging Applications for WebSphere Application Server Enterprise, Version 5*

  http://www-106.ibm.com/developerworks/websphere/techjournal/0304_klinger/klinger.html

- IBM developerWorks: WebSphere V5 Extended Messaging Support

  http://www-106.ibm.com/developerworks/websphere/library/techarticles/0302_wadley/wadley.html

- IBM developerWorks: *IBM WebSphere Developer Technical Journal: Simplify Applications by Using WebSphere Extended Messaging*

  http://www-106.ibm.com/developerworks/websphere/techjournal/0303_green/green.html

- IBM developerWorks: *Introduction to container-managed persistence and relationships, Part 3*

  http://www-106.ibm.com/developerworks/edu/ws-dw-wscomp3-i.html

- IBM developerWorks: Goetz, B. 2004: *Java Theory and Practice: Garbage Collection and Performance*

  http://www-106.ibm.com/developerworks/java/library/j-jtp01274.html

- IBM developerWorks: *Java internationalization basics*

  http://www-106.ibm.com/developerworks/java/edu/j-dw-javai18n-i.html

- IBM developerWorks: *International calendars in Java*

  http://www-106.ibm.com/developerworks/java/library/j-javacal/

- IBM developerWorks: *Share and Share Alike - A New Project Interchange Feature for Eclipse and WebSphere Studio*

  http://www-106.ibm.com/developerworks/websphere/library/techarticles/0309_bergschacher/bergschacher.html

- Microsoft Visual C Toolkit Web site

  http://msdn.microsoft.com/visualc/vctoolkit2003/

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

WebSphere Business Integration Server Foundation V5.1 Handbook

Process Choreography with WebSphere Studio Integration Edition V5.1

Runtime and development of BPEL4WS with sample code

J2EE Programming Model Extensions

This IBM Redbook describes the technical details of WebSphere Business Integration Server Foundation and discusses using WebSphere Studio Application Developer Integration Edition for application development. It provides valuable information for system administrators, developers and architects about the products covered. This redbook specifically focuses on WebSphere Process Choreographer and on solutions implementing it.

Part 1, “Architecting a WebSphere Enterprise solution” includes high-level details about WebSphere solutions using WebSphere Business Integration Server Foundation.

Part 2, “Setting up the environment” provides step-by-step details about installing the runtime and development environments.

Part 3, “Implementing WebSphere Enterprise solutions” provides details about the J2EE Programming Model Extensions and functions in WebSphere Business Integration Server Foundation. You can learn how to design, develop, assemble, deploy and administer applications in the WebSphere Business Integration Server Foundation environment.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-6318-00 ISBN 0738490857