BEA WebLogic Integration™

WebLogic Integration Internals

Version 8.1 Service Pack 6
Revised: June 2006

Copyright

Copyright © 2004-2006 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, BEA JRockit, BEA Liquid Data for WebLogic, BEA WebLogic Server, Built on BEA, Jolt, JoltBeans, SteelThread, Top End, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA AquaLogic, BEA AquaLogic Data Services Platform, BEA AquaLogic Enterprise Security, BEA AquaLogic Service Bus, BEA AquaLogic Service Registry, BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Manager, BEA MessageQ, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Enterprise Security, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Java Adapter for Mainframe, BEA WebLogic JDriver, BEA WebLogic JRockit, BEA WebLogic Log Central, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server Process Edition, BEA WebLogic WorkGroup Edition, BEA WebLogic Workshop, and Liquid Computing are trademarks of BEA Systems, Inc. BEA Mission Critical Support is a service mark of BEA Systems, Inc. All other company and product names may be the subject of intellectual property rights reserved by third parties.

All other trademarks are the property of their respective companies.

Contents

About This Document

Overview Documents for WebLogic Integration v
What You Need to Know vi
How to Print this Document. vi
Related Information vii
Contact Us! vii
Documentation Conventions viii

1. What Does WebLogic Integration Add to WebLogic Workshop?

2. An Example WebLogic Integration Component: A JPD File

HelloWorld.jpd 2-1
JPD Structure 2-2
JPD Annotation 2-2
JPD Conversation Lifetime 2-2

3. Component Compilation

WebLogic Integration Compilation Artifacts 3-1
Generated Process Class 3-1
Generated XML files 3-1
Component Compilation Products 3-2
Transport Objects 3-3
Dispatcher Objects 3-3

Dispatcher EJBs 3-3
Asynchronous Queues 3-4
Containers 3-4

4. Application Directory Structure

5. Component Invocation

Invocation Data Flow 5-1
Message Transport 5-2
Request Dispatch 5-3
Synchronous vs. Asynchronous Dispatch 5-3
Stateless vs. Stateful Methods 5-3
Use of JMS 5-3
Transactions 5-3
Implicit Timers 5-4

6. Application Customization

wli-config.properties File 6-1

Index

About This Document

This document is an addendum to the *WebLogic Workshop Internals* white paper available at the following location:

http://dev2dev.bea.com/products/wlworkshop81/articles/wlw_internals.jsp

Specifically, WebLogic Integration Internals describes the features that WebLogic Integration adds to the WebLogic Workshop platform, and compares the Java Web Service (JWS) files you can build with WebLogic Workshop to the Java Process Definition (JPD) files you can build when you add WebLogic Integration to your BEA development environment.

Overview Documents for WebLogic Integration

This document complements a series of documents that provide an overview of WebLogic Integration, and that explain how the functionality provided by WebLogic Integration is used at various stages in the design, development, and deployment of integrated solutions. Readers should start with these documents to gain a comprehensive understanding of the functionality provided by WebLogic Integration. The documents in the series are:

- *Introducing WebLogic Integration*—Provides an overview of WebLogic Integration. It describes the application integration, Trading Partner Integration, business process management, and data integration functionality provided by WebLogic Integration to solve e-business integration problems.
- *Managing WebLogic Integration Solutions*—Describes how to administer and manage applications built using WebLogic Integration.

These and other WebLogic Integration documents are available at the following URL:

http://edocs.bea.com/wli/docs81/index.html

Once you are familiar with the contents of the overview documents, you can proceed to the detailed documentation about the functionality provided by WebLogic Integration.

This document is organized as follows:

- Chapter 1, "What Does WebLogic Integration Add to WebLogic Workshop?," answers design-time and run-time questions regarding WebLogic Integration application performance.
- Chapter 2, "An Example WebLogic Integration Component: A JPD File," answers questions regarding how to WebLogic Integration application configuration.

What You Need to Know

This document is intended primarily for:

- Application developers who are creating WebLogic Integration applications.
- System administrators who set up, deploy, and administer WebLogic Integration in a production environment.
- Database administrators who set up, deploy, and administer database management systems for WebLogic Integration in a production environment.

This document assumes that you have read *WebLogic Workshop Internals*, which is available at the following location:

http://dev2dev.bea.com/products/wlworkshop81/articles/wlw_internals.jsp

For an overview of the WebLogic Integration architecture, see *Introducing WebLogic Integration*.

How to Print this Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—Print option on your Web browser.

A PDF version of this document is available on the WebLogic Integration documentation CD. You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format.

If you do not have the Adobe Acrobat Reader installed, you can download it for free from the Adobe Web site at http://www.adobe.com/.

Related Information

For information about installing WebLogic Integration, see Installing BEA WebLogic Platform which is available at the following URL:

http://edocs.bea.com/platform/docs81/index.html

WebLogic Integration documentation is available at the following URL:

http://edocs.bea.com/wli/docs81/index.html

WebLogic Server documentation is available at the following URL:

http://edocs.bea.com/wls/docs81/index.html

Contact Us!

Your feedback on the WebLogic Integration documentation is important to us. Send us e-mail at **docsupport@bea.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the WebLogic Integration documentation.

In your e-mail message, please indicate which version of the product and the documentation you are using.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and filenames and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Example*s: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

Convention	Item

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

. . .	Indicates one of the following in a command line:
	• That an argument can be repeated several times in a command line
	• That the statement omits additional optional arguments
	• That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER 1

What Does WebLogic Integration Add to WebLogic Workshop?

This paper describes the high-level components that WebLogic Integration adds to WebLogic Workshop to enable rapid business system integration. It is an addendum to *WebLogic Workshop Internals* which you can find at the following location:

http://dev2dev.bea.com/products/wlworkshop81/articles/wlw_internals.jsp

WebLogic Integration provides functionality for developing new applications, integrating them with existing systems, streamlining business processes, and extending e-business infrastructure through portal gateways. WebLogic Server, the industry-leading J2EE application server, provides the critical infrastructure needed to develop integrated solutions, including security, transaction management, fault tolerance, persistence, and clustering.

Leveraging WebLogic Server as the underlying deployment environment, WebLogic Integration uses web services to integrate distributed systems inside and outside an organization, and uses the WebLogic Workshop framework to simplify application development. WebLogic Integration supports the rapid development of integration applications by adding the following elements to WebLogic Workshop:

- Java Process Definitions (JPDs)
- WebLogic Integration Java Controls:
  - *Application View*, to access EIS (Enterprise Information System) applications
  - *ebXML*, to exchange messages with trading partners using the ebXML business protocol. The ebXML Message Service is sponsored by UN/CEFACT and OASIS. It provides security and reliability features that are not provided in the specifications for SOAP and SOAP Messages with Attachments.

For more information about these and other ebXML specifications, see the ebXML web site at the following URL:

http://www.ebxml.org/specs/index.htm

- *Email*, to allow WebLogic Integration business processes to send e-mail to a specific destination
- *File*, to read, write, or append to files in a file system
- *MB* (Message Broker) *Publish*, *MB Subscribe*, to provide publish and subscribe message-based communications for WebLogic Integration business processes
- *Process*, to allow WebLogic Integration business processes to invoke other business processes
- *RosettaNet*, to exchange messages with trading partners using the RosettaNet business protocol. The RosettaNet Implementation Framework (RNIF) specification is a guideline for applications that implement RosettaNet Partner Interface Processes (PIPs). These PIPs are standardized electronic business processes used between trading partners.

  For complete information about the RNIF specification and a list of PIPs, see the RosettaNet web site at the following URL:

  http://www.rosettanet.org
- *Service Broker*, to allow WebLogic Integration business processes to interface with a single control that provides relays, based upon decision criteria, to any number of other services or business processes.
- *Task*, *Task Worker*, to allow interaction from end users—such as task creators, task workers, and task administrators—to business processes for handling process exceptions, approvals, status tracking, and so forth.
- *TPM* (Trading Partner Management), to provide WebLogic Integration business processes with query (read-only) access to trading partner and service information stored in the TPM repository
- *Transformation*, to transform XML data in business processes using either XQuery expressions or eXtensible Stylesheet Language Transformations (XSLTs).
- *WLI JMS* (WebLogic Integration Java Message Service), to allow WebLogic Integration business processes to easily interact with messaging systems that provide a JMS implementation

The Application View, ebXML, MB Publish, MB Subscribe, and RosettaNet controls have associated runtime components, with separate WebLogic Server deployed resources including EJBs, queues, and so on.

- Runtime components not directly associated with controls:
  - Process Monitoring and Tracking
  - Message Tracking (B2B business process, only)
  - Event Generators (JMS, email, file, timer)
  - Business Calendar

The difference between the runtime infrastructure for JPDs and JWS (Java Web Service) definitions is the central topic of this paper.

Discussion of the additional runtime components provided by Weblogic Integration will be covered in a future version of this paper.

CHAPTER 2

An Example WebLogic Integration Component: A JPD File

WebLogic Workshop Internals describes the structure of a JWS (Java Web Service) file. A JWS file is very similar to a JPD (Java Process Definition) file. This section describes the basic outline of a JPD file.

HelloWorld.jpd

Below is the code for a process version of `HelloSync.jpd`, a sample process definition that returns a string result to a web service request:

```
/**
* @jpd:process process::
* <process name="HelloSync">
*   <clientRequest name="Client Request with Return"
*                  method="clientRequest" returnMethod="clientReturn"/>
* </process>::
*/
public class HelloSync implements com.bea.jpd.ProcessDefinition
{
public static final java.lang.String mReturnVal = "Hello, World";

public void clientRequest()
{
}

public java.lang.String clientReturn()
```

```
{
return this.mReturnVal;
}
}
```

Note: The JPD comments that normally accompany an IDE-generated JPD file have been removed for brevity.

JPD Structure

Paralleling the structure of a JWS file, each JPD file contains a single top-level class that implements a web service. While the file has the special `.jpd` extension, it contains completely valid Java source code. The `.jpd` extension is merely used to identify the file as a business process web service. When a web service is accessed via its URL, the `.jpd` extension is used by the WebLogic Workshop runtime to route the request to the correct subsystem.

A JPD file contains custom Javadoc annotations that indicate various configuration attributes of the web service which are supported by the WebLogic Workshop runtime. A JPD file may reference WebLogic Workshop and WebLogic Integration Java controls.

JPD Annotation

In addition to the single top-level implementation class, a JPD has a mandatory process annotation, `jpd:process`. This annotation provides message orchestration capability, or a high-level program counter that can persist across Java calls.

Because the process definition describes the order in which messages are expected, there is no need to use the `jws:conversation` annotation. The `jpd:process` annotation already describes the start, continue, and finish properties of methods.

JPD Conversation Lifetime

There is one notable use of a JWS annotation in a JPD: `jws:conversation-lifetime`. This feature sets an implicit timer to time out the process after a specified idle time.

This has two side effects:

- A long running process may unexpectedly time out while waiting for a message.
- A short running process will generate many timer messages, which causes excessive memory consumption.

If your WebLogic Integration application doesn't require the `conversation-lifetime` feature, we recommend that you set it to `0` to disable the feature.

CHAPTER 3

Component Compilation

Before a WebLogic Integration application component can be invoked, it must be compiled. Compilation timing and products for JPDs are generally the same as for compilation of JWS files. The following sections describe the areas where compilation of JPDs differs from that of JWS files.

WebLogic Integration Compilation Artifacts

WebLogic Integration process projects generate a few additional artifacts during compilation.

Generated Process Class

When the JPD file is compiled, it generates two classes: the implementation class and the process class. The process class ends in `_wf`. The process class is generally invisible, but it will show up in stack traces (when running performance analyzers, for example).

Generated XML files

In production (`non-iterativedev`) mode, a WebLogic Integration application listener loads specific XML files into in-memory tables when the application is deployed.

The `com.bea.wlw.runtime.core.servlet.WebappContextListener` is defined in the WEB—the listener `INF/web.xml` file of a process project. The `INF/web.xml` file is generated when creating a process project, and it contains the listener. No user action is necessary to configure it.

The WebLogic Integration application listener loads the following XML files:

- Version file (`META-INF/wli-process.xml`)

  During compilation, WebLogic Workshop creates a version file describing all versioned JPDs.

- Channel file (`META-INF/wli-channels.xml`)

  During EAR (Enterprise Application Archive) generation, WebLogic Workshop creates an aggregated channel file combining all `.channel` files in schema directories for the application.

- Static subscribers file (`META-INF/wli-subscriptions.xml`)

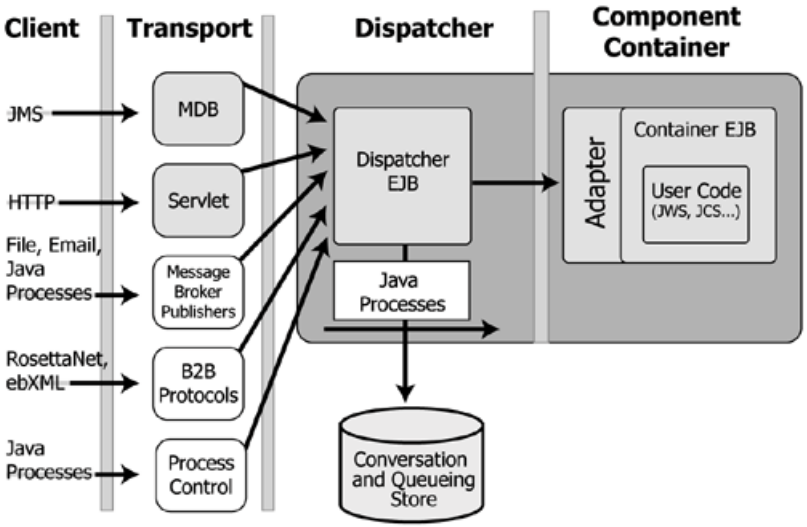
  During EAR generation, WebLogic Workshop creates an aggregated subscriptions file combining all the `jpd:mb-static-subscription` annotations and all `mbsubscription` control annotations of the application.

Component Compilation Products

The compiled components of a JPD project are generally the same as those of a JWS. There's another dispatcher type described in Dispatcher EJBs, and different client types are allowed to access the Dispatcher—Message Broker publishers, B2B protocols (ebXML and RosettaNet), and Java processes using the process control.

The following figure shows the components of a process project. The shaded areas show overlap with web service projects; the unshaded areas are specific to process projects. You can see from this figure that the basic compilation components are the same, but process projects allow the additional, aforementioned transport clients.

Figure 3-1 Invocation Dispatch Detail



Transport Objects

JWS supports two transport protocols by which a client may invoke web services: HTTP and JMS. As mentioned earlier, JPDs support the same two transport protocols and add support for:

- Message Broker publishers (event generators and Java processes)
- B2B protocols (RosettaNet and ebXML)
- JPDs using the process control
- Messages from JPD proxies that come in through a WLI Process Proxy Dispatcher, which then forwards to the project dispatcher. (Not shown in Figure 3-1.)

Dispatcher Objects

Dispatcher EJBs

WebLogic Integration uses the same `SyncDispatcher` and `AsyncDispatcher` as JWS. In addition, it adds a third Dispatcher EJB: `AsyncDispatcherErrorBean`.

`AsyncDispatcherErrorBean` is a Message Driven Bean that handles invocations of buffered component methods that have exceeded their JMS retry limits. This condition triggers an exception handler in the JPD.

Asynchronous Queues

JPD projects use the same type of asynchronous queue as that used by JWS projects. In addition, JPDs also have an error queue defined. This queue is configured to be an error destination of the asynchronous queue used for the project. The name of the queue is the name of the project queue suffixed by *_error*. For example, the JMS queue configured to handle asynchronous requests to a JPD named `WebServices/async/Buffer.jpd` would be named `WebServices.queue.AsyncDispatcher`. The error queue would be named `WebServices.queue.AsyncDispatcher_error`.

Containers

JPDs use the same internal container mechanism as JWS. Subclassing of internal containers takes care of minor differences. No user action is required.

CHAPTER 4

Application Directory Structure

The application directory structure for a process application is generally the same as that for a web service application. There are a few additional files produced during compilation of a JPD file. These files are described in "WebLogic Integration Compilation Artifacts" on page 3-1.

CHAPTER 5

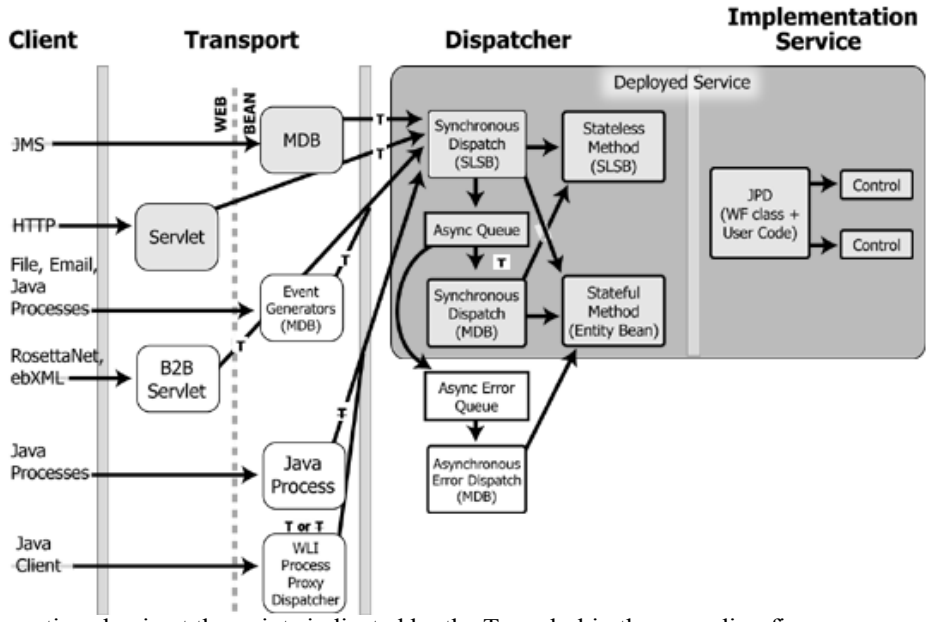
Component Invocation

This section describes what occurs when an XML or SOAP message representing a web service or business process method invocation arrives at the server.

Invocation Data Flow

The following figure illustrates the data flow for an invocation of a JPD method. The shaded areas indicate overlap with JWS method calls. The unshaded portions indicate additions to JPD invocation paths, compared to JWS method calls.

Figure 5-1 Invocation Dispatch Detail



Transactions begin at the points indicated by the T symbol in the preceding figure.

Message Transport

Requests arriving through JMS and HTTP protocols to a JPD follow the same runtime path as those going to a JWS.

For the other clients (Message Broker publishers, B2B protocols, Java processes using the process control, and Java clients using a JPD proxy), slight modifications to the runtime path occur:

- If the message is defined to be XML, then the message is parsed into a token stream. If the document store threshold (`weblogic.wli.DocumentMaxInlineSize`) is reached, then the document is stored in the document store and passed by reference. Otherwise, it is passed by value.
- If the message is defined to be `RawData` and if the document store threshold is reached (`weblogic.wli.DocumentMaxInlineSize`), then the document is stored in the document store and passed by reference. Otherwise, it is passed by value.

To learn more about configuring the document store threshold, see http://edocs/wli/docs81/deploy/wliconfig_appx.html.

Request Dispatch

Synchronous vs. Asynchronous Dispatch

JPDs handle synchronous and asynchronous calls using the same runtime paths, respectively, as for JWS. The only difference is in the annotation. Instead of using `@jws:message-buffer enable="true"` annotations, JPD methods are buffered by default. A synchronous method can be enabled using the `returnMethod` attribute of a `clientRequest` element of a process definition. Only the start method of a JPD can be synchronous.

Stateless vs. Stateful Methods

In a JWS, a process is made stateful by the presence of the `jws:conversation` attribute. JPDs use the process definition attribute to deduce whether a process is stateful or not.

A process is stateless if it has:

- no explicit transactions
- no implicit timers
- no control receives
- no `returnMethod` methods with a process node following the associated `clientRequest` node

Otherwise, a process is stateful.

Use of JMS

A JPD uses the same internal project queues as a JWS.

Transactions

A JPD generally uses the same implicit transaction model as a JWS, as shown in Figure 5-1.

Note: There is one important difference between transaction boundaries for the JPD and JWS transaction models. When a Java process calls another Java process (or when an EJB

container calls using a JPD proxy), the current transaction is propagated to the Dispatcher. A new transaction is not started.

In a JWS, a transaction ends on a method boundary. In a JPD, a transaction ends when:

- A client receive or control callback node is reached

  In this case, the process code returns from the container EJB, and waits for a client receive or control callback to start a new transaction.

- A parallel node is started or ended

  The JPD sends itself a JMS message for each branch in the parallel node. It then returns from the container EJB and waits for the Dispatcher to restart the JPD from the JMS message.

  Note: You can disable this behavior by using the `continueTransaction` attribute of the parallel node. To learn more about configuring this attribute, see http://edocs.bea.com/wli/docs81/relnotes/relnotesLimit.html.

- An explicit transaction is started

  The JPD sends itself a JMS message to start the explicit transaction. It then returns from the container EJB and waits for the Dispatcher to restart the JPD from the JMS message.

- A `returnMethod` method is executed, and a process node follows the associated `clientRequest` node

  The JPD sends itself a JMS message to start the next node after the client request node. It returns from the container EJB with the result of the `returnMethod`, and waits for the Dispatcher to restart the JPD from the JMS message.

If any uncaught exception occurs during the invocation of the component method, the encapsulating transaction is rolled back. If the JPD was invoked from the `AsyncDispatcher` (either through a buffered message, buffered callback or one of the transaction restarts described above), then the request is retried according to the redelivery delay and the redelivery delay of the project asynchronous dispatcher queue. When retries are exhausted, the message is sent to the JMS error destination associated with that queue. The JPD `AsyncErrorDispatcher` will then invoke the JPD exception handler, or terminate the process if an exception handler doesn't exist.

Implicit Timers

JPDs make use of JMS *timer* messages. These are messages sent to the JMS project asynchronous dispatcher queue with a delivery time value set to a point in the future.

The following constructs send JMS timer messages:

- the `jws:conversation-lifetime` attribute

  For more information about `jws:conversation-lifetime`, see "JPD Conversation Lifetime" on page 2-2.

- the timer branch of an event choice
- the timer control, which sends these messages explicitly

There is some memory overhead for timer messages, as JMS keeps the indexing in memory. Messages are never cancelled; "cancelled" messages (where the timer callback is no longer expecting the message) are simply ignored when they are delivered.

JMS timer messages appear in the messages pending column in the WebLogic Server console.

CHAPTER 6

Application Customization

In addition to the base configuration files, JPD applications use the `wli-config.properties` file to enable additional application customization.

wli-config.properties File

The `wli-config.properties` file contains infrequently used parameters which are not configurable through the WebLogic Integration console.

You can find documentation for the `wli-config.properties` file at:

http://edocs/wli/docs81/deploy/wliconfig_appx.html

Index

A

annotations
  jpd:mb-static-subscription 3-2
  jpd:process 2-2
  jws:conversation 2-2, 5-3
  jws:conversation-lifetime 2-2, 2-3, 5-5
  jws:message-buffer 5-3
  mbsubscription 3-2
application listener 3-1
Application View control 1-3
AsyncDispatcher 3-3, 5-4
AsyncDispatcherErrorBean 3-3, 3-4
AsyncErrorDispatcher 5-4

B

business calendar 1-3

C

clientRequest 5-3
com.bea.wlw.runtime.core.servlet.WebappCont extListener 3-1
compilation artifacts
  generated process class 3-1
  generated XML files 3-1
containers 3-4
continueTransaction 5-4
controls
  Application View 1-1, 1-3
  ebXML 1-1, 1-3
  email 1-2
  file 1-2
  MB Publish 1-2, 1-3
  MB Subscribe 1-2, 1-3
  process 1-2, 3-2, 3-3
  RosettaNet 1-2, 1-3
  service broker 1-2
  task 1-2
  task worker 1-2
  timer 5-5
  TPM (Trading Partner Management) 1-2
  transformation 1-2
  with runtime components 1-3
  without runtime components 1-3
  WLI JMS (WebLogic Integration Java Message Service) 1-2

D

data transformation 1-2
dispatch
  asynchronous 5-3
  request 5-3
  synchronous 5-3
Dispatcher
  AsyncDispatcher 3-3
  AsyncDispatcherErrorBean 3-3, 3-4
  EJBs 3-2, 3-3
  invocation details 3-3
  objects 3-3
  restarting JPD 5-4
  SyncDispatcher 3-3
  transactions propagated to 5-4
  WLI Process Proxy Dispatcher 3-3
document store threshold

and RawData messages 5-2
and XML messages 5-2
weblogic.wli.DocumentMaxInlineSize 5-2

E

ebXML
accessing Dispatcher 3-2
as transport object 3-3
control 1-1, 1-3
message runtime path 5-2
specifications 1-2
EJBs
calls using a JPD proxy 5-3
Dispatcher 3-2
email control 1-2
event generator
as transport object 3-3
runtime component 1-3
exceptions
and AsyncDispatcherErrorBean 3-4
uncaught 5-4
excessive memory consumption 2-2
eXtensible Stylesheet Language Transformations (XSLTs) 1-2

F

file
channel 3-2
compilation artifacts 3-1
control 1-2
generated XML 3-1
INF/ web.xml 3-1
META-INF/ wli-channels.xml 3-2
META-INF/ wli-process.xml 3-2
META-INF/ wli-subscriptions.xml 3-2
static subscribers file 3-2
version 3-2
wli-config.properties 6-1
See also JPD (Java Process Definitions) file

H

HelloWorld.jpd 2-1
HTTP 5-2

J

JMS (Java Message Services)
internal project queues 5-3
requests arriving through 5-2
retry limits 3-4
JPD (Java Process Definition) file
and WLI Process Proxy Dispatcher 3-3
application directory structure compared to JWS (Java Web Services) file 4-1
as transport object 3-3
channel file 3-2
compilation compared to JWS (Java Web Services) file 3-1
compilation products 3-2
containers 3-4
conversation lifetime 2-2
custom Javadoc annotations 2-2
data flow compared to JWS (Java Web Services) method 5-1
data flow for method compared to JWS (Java Web Services) method 5-2
data runtime paths 5-2
HelloWorld.jpd 2-1
mandatory process annotation 2-2
message transport 5-2
method buffering 5-3
proxy 5-3
queues 3-4
structure compared to JWS (Java Web Service) file 2-2
subscriptions file 3-2
transaction boundaries 5-3
transaction model 5-3
transaction model compared with that of JWS (Java Web Services) 5-4
transport objects 3-3

version file 3-2
jpd:mb-static-subscription 3-2
jpd:process 2-2
JWS (Java Web Services)
container mechanism 3-4
data flow compared to JPD 5-1
Dispatcher EJBs 3-3
file compilation compared to JPD file 3-1
file structure compared to that of JPD file 2-2
jws:conversation 5-3
queues 3-4
request dispatch 5-3
runtime paths 5-2
transaction model 5-3
transport protocols 3-3
jws:conversation 2-2, 5-3
jws:conversation-lifetime 2-2, 2-3, 5-5
jws:message-buffer 5-3

M

MB Publish control 1-3
MB Subscribe control 1-3
mbsubscription 3-2
message
and document store threshold 5-2
as transport object 3-3
cancelled 5-5
causing timeout 2-2
from JPD proxies 3-3
in WebLogic Server console 5-5
JMS 5-4
orchestration 2-2
runtime paths 5-2
starting transaction 5-4
timer 2-2
tracking 1-3
transport 5-2
Message Broker
accessing Dispatcher 3-2
as transport object 3-3
jpd:mb-static-subscription annotation 3-2
MB Publish control 1-2, 1-3
MB Subscribe control 1-2, 1-3
mbsubscription annotation 3-2
message runtime path 5-2
META-INF/ wli-subscriptions.xml 3-2
META-INF/ wli-channels.xml 3-2
META-INF/ wli-process.xml 3-2
META-INF/ wli-subscriptions.xml 3-2
methods
boundary 5-4
returnMethod 5-4
start 5-3
stateful 5-3
stateless 5-3
synchronous 5-3

N

nodes
callback 5-4
clientRequest 5-4
parallel 5-4

O

OASIS 1-1

P

process
clientRequest 5-3
control 1-2
control as transport object 3-3
generated class 3-1
generating many timer messages 2-2
HelloWorld.jpd example 2-1
mandatory JPD (Java Process Definition) file annotation 2-2
monitoring and tracking 1-3
project, components of 3-2, 3-3

returnMethod 5-3
start method 5-3
stateful 5-3
stateless 5-3
termination 5-4
timeout condition 2-2
production mode 3-1

Q

queues
asynchronous dispatcher 5-4
error 3-4
naming conventions for 3-4

R

RawData 5-2
repository 1-2
retry limits 3-4
returnMethod 5-3
RosettaNet
accessing Dispatcher 3-2
as transport object 3-3
control 1-2, 1-3
message runtime path 5-2
Partner Interface Processes (PIPs) 1-2
specification 1-2
runtime components
business calendar 1-3
controls having 1-3
event generators 1-3
message tracking 1-3
not associated with controls 1-3
process monitoring and tracking 1-3

S

service broker control 1-2
SOAP 1-1
start method 5-3
stateful method 5-3
stateful processs 5-3
stateless method 5-3
stateless process 5-3
SyncDispatcher 3-3

T

task control 1-2
task worker control 1-2
timer
control 5-5
JMS messages 5-4
jws:conversation-lifetime annotation 2-2
Trading Partner Management (TPM)
control 1-2
repository 1-2
transactions
and Dispatcher 5-4
boundaries of 5-3
end conditions 5-4
explicit 5-4
model of 5-3
start conditions 5-4
transformation control 1-2
transport objects
ebXML 3-3
event generators 3-3
JPD (Java Process Definition) file 3-3
Message Broker publishers 3-3
messages from JPD proxies 3-3
RosettaNet 3-3
transport protocols
HTTP 3-3
JMS (Java Message Services) 3-3

U

UN/CEFACT 1-1

W

WebLogic Server 1-1, 5-5

weblogic.wli.DocumentMaxInlineSize 5-2
WLI JMS (WebLogic Integration Java Message Service) control 1-2
WLI Process Proxy Dispatcher 3-3
wli-config.properties 6-1

X

XML
and document store threshold 5-2
com.bea.wlw.runtime.core.servlet.WebappContextListener 3-1
generated files 3-1
INF/ web.xml file 3-1
META-INF/ wli-channels.xml 3-2
META-INF/ wli-process.xml 3-2
META-INF/ wli-subscriptions.xml 3-2
transformation 1-2
XQuery 1-2