Developer's Guide

Borland AppServer™ 6.6

Borland®

Borland Software Corporation
20450 Stevens Creek Blvd., Suite 800
Cupertino, CA 95014 USA
www.borland.com

Refer to the file deploy.html for a complete list of files that you can distribute in accordance with the License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

For third-party conditions and disclaimers, see the Release Notes on your product CD.

BAS66DevGuide
April 2006
PDF

Contents

Chapter

1

Introduction to Borland AppServer

Borland AppServer (AppServer) is a set of services and tools that enable you to build, deploy, and manage distributed enterprise applications in your corporate environment.

The AppServer is a leading implementation of the J2EE 1.4 standard, and supports the latest industry standards such as EJB 2.1, JMS 1.1, Servlet 2.4, JSP 2.0, CORBA 2.6, XML, and SOAP. Borland provides two versions of AppServer, which include leading enterprise messaging solutions for Java Messaging Service (JMS) management (Tibco and OpenJMS). You can choose the degree of functionality and services you need in AppServer, and if your needs change, it is simple to upgrade your license. See Chapter 1, “Introduction to Borland AppServer” or Chapter 1, “Introduction to Borland AppServer” for details.

The AppServer allows you to securely deploy and manage all aspects of your distributed Java and CORBA applications that implement the J2EE 1.4 platform standard.

With AppServer, the number of server instances per installation is unlimited, so the maximum of concurrent users is unlimited.

AppServer includes:

- Implementation of J2EE 1.4.
- Apache Web Server version 2.2
- Borland Security, which provides a framework for securing AppServer.
- Single-point management of leading JMS management solutions included with AppServer (Tibco, and OpenJMS).
- Strong management tools for distributed components, including applications developed outside of AppServer.

AppServer features

AppServer offers the following features:

- Support for BAS platforms (please refer to `http://support.borland.com/kbcategory.jspa?categoryID=389` for a list of the platforms supported for AppServer).
- Full support for clustered topologies.
- Seamless integration with the VisiBroker ORB infrastructure.
- Integration with the Borland JBuilder integrated development environment.
- Enhanced integration with other Borland products including Borland Together ControlCenter, Borland Optimizeit Profiler and ServerTrace.
- AppServer allows existing applications to be exposed as Web Services and integrated with new applications or additional Web Services. Borland Web Services support is based on Apache Axis 1.2 technology, the next-generation Apache SOAP server that supports SOAP 1.2.

Borland AppServer Documentation

The AppServer documentation set includes the following:

- *Borland AppServer Installation Guide*—describes how to install AppServer on your network. It is written for system administrators who are familiar with Windows or UNIX operating systems.
- *Borland AppServer Developer's Guide*—provides detailed information about packaging, deployment, and management of distributed object-based applications in their operational environment.
- *Borland Management Console User's Guide*—provides information about using the Borland Management Console GUI.
- *Borland Security Guide*—describes Borland's framework for securing AppServer, including VisiSecure for VisiBroker for Java and VisiBroker for C++.
- Borland *VisiBroker for Java Developer's Guide*—describes how to develop VisiBroker applications in Java. It familiarizes you with configuration and management of the Visibroker ORB and how to use the programming tools. Also described is the IDL compiler, the Smart Agent, the Location, Naming and Event Services, the Object Activation Daemon (OAD), the Quality of Service (QoS), and the Interface Repository.
- Borland *VisiBroker VisiTransact Guide*—describes Borland's implementation of the OMG Object Transaction Service specification and the Borland Integrated Transaction Service components.

The documentation is typically accessed through the Help Viewer installed with your AppServer product. You can choose to view help from the standalone Help Viewer or from within a AppServer GUI tool. Both methods launch the Help Viewer in a separate window and give you access to the main Help Viewer toolbar for navigation and printing, as well as access to a navigation pane. The Help Viewer navigation pane includes a table of contents for all AppServer books and reference documentation, a thorough index, and a comprehensive search page.

The PDF books, Borland AppServer Developer's Guide and Borland Management Console User's Guide are available online at `http://info.borland.com/techpubs/appserver`.

Accessing AppServer online help topics in the standalone Help Viewer

To access the online help through the standalone Help Viewer on a machine where the product is installed, use one of the following methods:

Windows

- Choose Start|Programs|Borland Deployment Platform|Help Topics
- or, open the Command Prompt and go to the product installation `\bin` directory, then type the following command:

```
help
```

UNIX

Open a command shell and go to the product installation `/bin` directory, then enter the command:

```
help
```

Tip

During installation on UNIX systems, the default is to not include an entry for `bin` in your `PATH`. If you did not choose the custom install option and modify the default for `PATH` entry, and you do not have an entry for current directory in your `PATH`, use `./help` to start the help viewer.

Accessing AppServer online help topics from within a AppServer GUI tool

To access the online help from within a AppServer GUI tool, use one of the following methods:

- From within the Borland Management Console, choose Help|Help Topics
- From within the Borland Deployment Descriptor Editor (DDEditor), choose Help| Help Topics

The Help menu also contains shortcuts to specific documents within the online help. When you select one of these shortcuts, the Help Topics viewer is launched and the item selected from the Help menu is displayed.

Documentation conventions

The documentation for AppServer uses the typefaces and symbols described below to indicate special text:

Convention	Used for
italics	Used for new terms and book titles.
`computer`	Information that the user or application provides, sample command lines and code.
`bold computer`	In text, bold indicates information the user types in. In code samples, bold highlights important statements.
[ ]	Optional items.
…	Previous argument that can be repeated.
|	Two mutually exclusive choices.

Platform conventions

The AppServer documentation uses the following symbols to indicate platform-specific information:

Symbol	Indicates
Windows	All supported Windows platforms.
Win2003	Windows 2003 only
WinXP	Windows XP only
Win2000	Windows 2000 only
UNIX	UNIX platforms
Solaris	Solaris only

Contacting Borland support

Borland offers a variety of support options. These include free services on the Internet where you can search our extensive information base and connect with other users of Borland products. In addition, you can choose from several categories of telephone support, ranging from support on installation of Borland products to fee-based, consultant-level support and detailed assistance.

For more information about Borland's support services or contacting Borland Technical Support, please see our web site at `http://support.borland.com` and select your geographic region.

When contacting Borland's support, be prepared to provide the following information:

- Name
- Company and site ID
- Telephone number
- Your Access ID number (U.S.A. only)
- Operating system and version
- Borland product name and version
- Any patches or service packs applied
- Client language and version (if applicable)
- Database and version (if applicable)
- Detailed description and history of the problem
- Any log files which indicate the problem
- Details of any error messages or exceptions raised

Online resources

You can get information from any of these online sources:

World Wide Web: `http://www.borland.com`

Online Support: `http://support.borland.com` (access ID required)

World Wide Web

Check `http://www.borland.com` regularly. The AppServer Product Team posts white papers, competitive analyses, answers to FAQs, sample applications, updated software, updated documentation, and information about new and existing products.

You may want to check these URLs in particular:

- `http://www.borland.com/downloads/download_appserver.html` (AppServer software and other files)
- `http://support.borland.com` (AppServer FAQs)

Borland newsgroups

You can participate in many threaded discussion groups devoted to the AppServer. Visit `http://www.borland.com/newsgroups` for information about joining user-supported newsgroups for Enterprise Server and other Borland products.

Note These newsgroups are maintained by users and are not official Borland sites.

Chapter

2

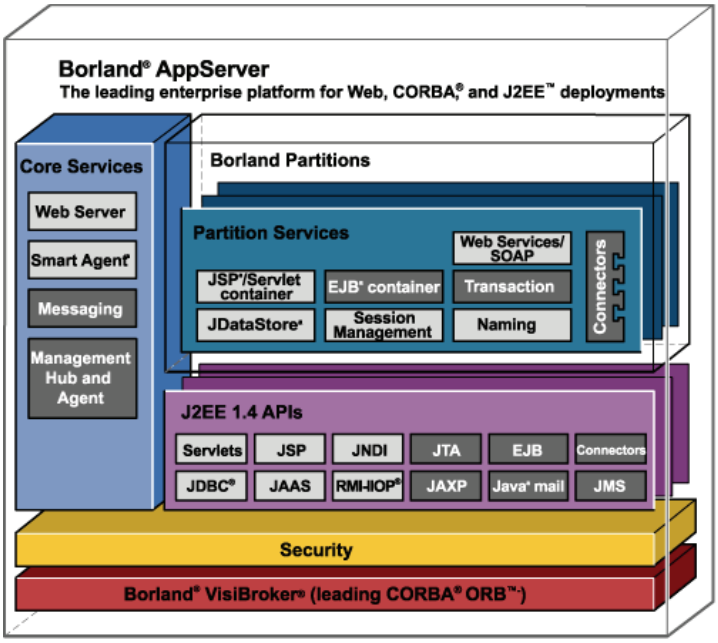
Borland AppServer overview and architecture

This section contains an overview of the Borland AppServer (AppServer).

AppServer architecture overview

The AppServer is a CORBA-based, J2EE server that utilizes distributed objects throughout its architecture. With the AppServer, you can establish connectivity to platforms from corporate mainframes to simpler systems with small-business applications and remote databases. The AppServer components process your enterprise application based on how it is packaged and how the deployment descriptors describe the application's modules.

In the following architectural diagram, your enterprise applications sit on top of the AppServer. An application server installation contains AppServer *core services* and *Partitions*.



AppServer services overview

AppServer services are those services available to all applications being hosted on the AppServer. They are:

- Web Server
- Java Messaging (JMS)
- Smart Agent
- 2PC Transaction Service

Web Server

The AppServer includes the Apache Web Server version 2.0. The Apache web server is a robust, commercial grade reference implementation of the HTTP. protocol. The Apache web server is highly configurable and extensible through the addition of third-party modules. Apache supports clients with varying degrees of sophistication and supports content negotiation to this end. Apache also provides unlimited URL aliasing.

Borland has added an IIOP Plug-in to the Apache web server. The IIOP Plug-in allows Apache and the Borland web container to communicate via Internet Inter-ORB Protocol (IIOP), allowing users to add the power of CORBA with their Web applications in new ways. In addition, IIOP is the protocol of the VisiBroker ORB, allowing your Web applications to fully leverage the services provided by the object-request-broker provided by Borland.

JMS

The AppServer provides support for standard JMS pluggability, and currently bundles the Tibco messaging service. See Chapter 23, “JMS provider pluggability” for vendor-specific information on JMS services.

Smart Agent

The Smart Agent is a distributed directory service provided by the VisiBroker ORB used in the AppServer. The Smart Agent provides facilities used by both client programs and object implementations, and must be started on at least one host within the local server network.

Note Users of the Web Edition do not have to use the Smart Agent if they expect their Web server and Web containers to communicate through HTTP or another Web protocol. To leverage the IIOP Plug-in (and, by extension, the ORB provided with the Web Edition), however, the Smart Agent must be turned on.

More than one Smart Agent can be configured to run on your network. When a Smart Agent is started on more than one host, each Smart Agent will recognize a subset of the objects available and communicate with the other Smart Agents to locate objects it cannot find. In addition, if one of the Smart Agent processes should terminate unexpectedly, all implementations registered with that Smart Agent discover this event and they will automatically re-register with another Smart Agent. It should be noted that If a heavy services lookup load is necessary, it is advisable to use the Naming Service (VisiNaming). VisiNaming provides persistent storage capability and cluster load balancing whereas the Smart Agent only provides a simple round robin on a per osagent basis.

2PC Transaction Service

The Two-Phase Commit (2PC) Transaction Service exists provides a complete recoverable solution for distributed transactional CORBA applications. Implemented on top of the VisiBroker ORB, the 2PC Transaction Service simplifies the complexity of distributed transactions by providing an essential set of services, including a transaction service, recovery and logging, integration with databases, and administration facilities within one, integrated architecture.

The Partition and its services

A Partition is an application's deployment target. The Partition provides the J2EE server-side runtime environment required to support a complete J2EE 1.3 application. While a Partition is implemented as a single native process, its core implementation is Java. When a Partition starts, it creates an embedded Java Virtual Machine (JVM) within itself to run the Partition implementation and the J2EE application code.

Partitions are present in each AppServer Edition and product but they host less diverse archives in the Web Services, Team and VisiBroker Editions. This section describes the full-featured functional Partitions offered in the full Borland AppServer. Each Partition instance provides:

- Connector Service
- EJB Container
- JDataStore Server
- Lifecycle Interceptor Manager
- Naming Service
- Session Storage Service
- Transaction Manager
- Web Container

Connector Service

The Connector Service, also known as VisiConnect, is the Borland implementation of the Connectors 1.0 standard, which provides a simplified environment for integrating various EISs with the AppServer. The Connectors provide a solution for integrating J2EE-platform application servers and EISs, leveraging the strengths of the J2EE platform—connection, transaction and security infrastructure—to address the challenges of EIS integration. For more information see Chapter 31, “VisiConnect overview.”

EJB Container

The AppServer provides integrated EJB container services. These services allow you to create and manage integrated EJB containers or EJB containers across multiple Partitions. Use this service to deploy, run, and monitor EJBs. Tools include a Deployment Descriptor Editor (DDEditor) and a set of task wizards for packaging and deploying EJBs and their related descriptor files. EJB containers can also make use of J2EE connector architecture, which enables J2EE applications to access Enterprise Information Systems (EISs).

JDataStore Server

Borland's JDataStore is a relational database service written entirely in Java. You can create and manage as many JDataStores as desired. For more information on JDataStore, see the JDatastore online documentation at `www.borland.com/techpubs/jdatastore/`.

Lifecycle Interceptor Manager

You can use Lifecycle Interceptors to further customize your implementation. Partition Lifecycle Interceptors allow you to perform operations at certain points in a Partition's lifecycle. For more information see Chapter 30, “Implementing Partition Interceptors,”

Naming Service

The Naming Service is provided by the VisiBroker ORB. It allows developers, assemblers, and/or deployers to associate one or more logical names with an object reference and store those names in a VisiBroker namespace. It also allows application clients to obtain an object reference by using the logical name assigned to that object. Object implementations can bind a name to one of their objects within a namespace which client applications can then use to resolve a name using the `resolve()` method. The method returns an object reference to a naming context or an object.

Session Storage Service

The Java Session Service (JSS) is a service that stores information pertaining to a specific user session. The JSS provides a mechanism to easily store session information into a database. For example, in a shopping cart scenario, information about your session (your login name, the number of items in the shopping cart, and such) is polled and stored by the JSS. So when a session is interrupted by a Borland web container unexpectedly going down, the session information is recoverable by another Tomcat instance through the JSS. The JSS must be running on the local network. Any web container instance (in the cluster configuration) will find the JSS, connect to it, and continue session management. For more information, see “Java Session Service (JSS) configuration” on page 57.

Transaction Manager

A Partition Transaction Manager exists in each AppServer Partition. It is a Java implementation of the CORBA Transaction Service Specification. The Partition Transaction Manager supports transaction timeouts, one-phase commit protocol, and can be used in a two-phase commit protocol under special circumstances. For more information, see Chapter 18, “Transaction management.”

Web Container

The Web Container is designed to support deployment of web applications or web components of other applications (for example, servlets and JSP files). The AppServer provides the Borland Web Container, which is based on Tomcat 4.1. Tomcat is a sophisticated and flexible open-source tool that provides support for servlets, JavaServer Pages, and HTTP. Borland has also provided an IIOP plug-in with its Web Container, enabling communication with application components and the web server over IIOP rather than strict HTTP. Other features of the Web Container are:

- EJB referencing
- DataSource referencing
- Environment referencing
- Integration into industry-standard web servers

For more information, see “Web components” on page 33.

Borland AppServer and J2EE APIs

Since the AppServer is fully J2EE 1.4 compliant, it supports the use of the following J2EE 1.4 APIs:

- JNDI: the Java Naming and Directory interface
- RMI-IIOP: remote method invocation (RMI) carried out via internet inter-ORB protocol (IIOP)
- JDBC: for getting connections to and modeling data from databases
- EJB 2.1: the Enterprise JavaBeans 2.1 APIs
- Servlets 1.0: the Sun Microsystems servlets APIs
- JSP: JavaServer Pages APIs
- JMS: Java Messaging Service
- JTA: the Java transactional APIs
- Java Mail: a Java email service
- Connectors 1.5: the J2EE Connector Architecture
- JAAS: the Java Authentication and Authorization Service
- JAXP: the Java API for XML parsing

JDBC

Borland implements the Java DataBase Connection APIs from Sun Microsystems. JDBC provides APIs for writing database drivers and a full Service Provider Interface (SPI) for those looking to develop their own drivers. JDBC also supports connection pooling and distributed transaction features. For more information, go to the Transaction management and JDBC, JDBC API Modifications section.

Java Mail

Java Mail is an implementation of Sun's Java Mail API. It is a set of abstract APIs that model a mail system. The API provides a platform independent and protocol independent framework to build Java-technology-based email client applications.

JTA

The Java Transactional API (JTA) defines the UserTransaction interface required by application components to start, stop, rollback, or commit transactions. EJBs establish transaction participation through the `getUserTransaction` method, while other components do so using JNDI lookups. JTA also specifies the interfaces needed by Connectors and resource managers to communicate with an application server's transaction manager.

JAXP

The Java APIs for XML Parsing (JAXP) enable the processing of XML documents using the DOM, SAX, and XSLT parsing implementations. Developers can easily use the parser provided with the reference implementation of the API to XML-enable their Java applications.

JNDI

The Java Naming and Directory Interface is used to allow developers to customize their application components at assembly and deployment without changes to the component's source code. The container implements the runtime environment for the components and provides the environment to the component as a JNDI naming context. The components' methods access the environment through JNDI interfaces. The JNDI naming context itself stores the application environment information and makes it available to all application components at runtime.

RMI-IIOP

The VisiBroker ORB supports RMI-over-IIOP protocol. When used in conjunction with the IIOP Connector Module for Apache and the Borland web container, it allows distributed web applications built on CORBA foundations. For more information, see “Using RMI over IIOP” in the *VisiBroker for Java Developer's Guide*.

Other Technologies

It is also possible to wrap other technologies, provide them as services, and run them in the AppServer.

Optimizeit Profiler and Optimizeit ServerTrace

Borland's Optimizeit Profiler (purchased separately) helps you track memory and CPU usage issues during the development of Java applications. Optimizeit ServerTrace provides a comprehensive, high-level application performance analysis and root-cause diagnostics that accelerate time-to-resolution of performance issues across complex, distributed J2EE and SOA-enabled systems. The AppServer runs Optimizeit Profiler and Optimizeit ServerTrace at the Partition level.

See the Sun Java Center for more information on these APIs.

Chapter

3

Partitions

This section explains what Partitions are and how they work. It explores the Partition's footprint, facilities, configuration, and how to run a Partition.

Partitions Overview

Partitions are the runtime hosting environment for J2EE and web service application components. A Partition is a process that can be tuned to suit the application it is hosting. You can create any number of Partitions to isolate, scale, or cluster your application deployment to meet your own requirements. Extensive tooling enables you to simply create, configure, and distribute Partitions to your needs.

A Partition provides containers and services needed for your applications:

- Web Container
- EJB Container
- Naming Service
- Session Service
- Transaction Service
- Connector Service
- JDataStore Database Server
- Partition Lifecycle Interceptor Service

Additional applications and application components are also provided that can be used in your applications:

- UDDI Server
- Apache Struts
- Apache Cocoon
- Petstore J2EE blueprint application
- SmarTicket J2EE blueprint application

By enabling and disabling the various Partition containers and services, and configuring the Partition's environment, you can “right-size” the Partition to its specific task. Typical use cases for a Partition include:

- Providing a complete isolated J2EE server platform for an application with all relevant J2EE container and services enabled.
- Providing a platform for a component of a distributed application such as its Web Tier with just the Web Container and Session Service enabled.

- Providing a central service such as a platform for the Borland AppServer (AppServer) UDDI server with just its Web Container enabled.
- Providing a diagnostic platform for an application such as running under Optimizeit.

Avoiding monolithic J2EE server Partitions hosting many applications also allows you to fine tune the Java environment the application needs. The version and type of JDK together with such configuration as heap space available ensures a satisfactory environment in which to run, while not over-allocating resources. Limits on pooled resources such as threads and connections may similarly be configured for optimal total performance. Partitions also have their own individual security settings for authentication mechanisms, authorization tables, and so on. A user who has authority to access all resources in a development Partition may be granted much more limited authority in a production Partition.

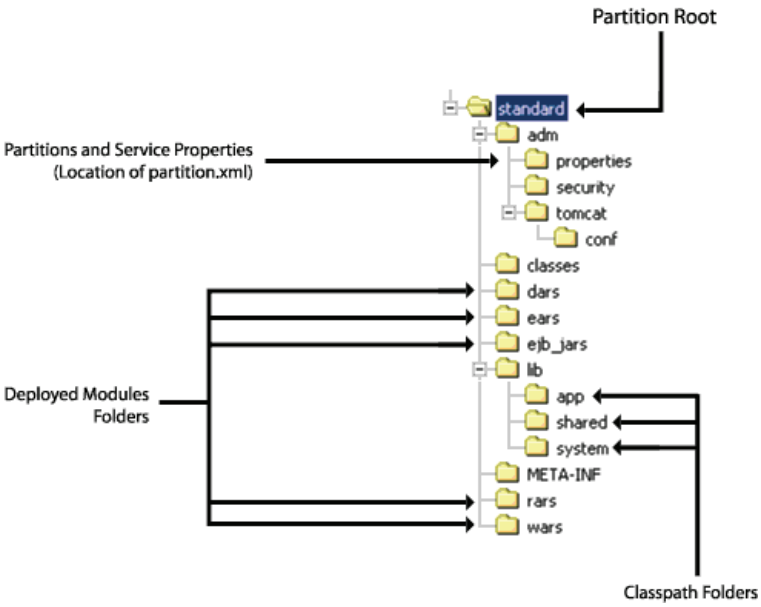
Creating Partitions

Partitions are created as Managed Objects in a “configuration” from templates provided in the Borland Management Console. Typically the Partition disk footprint is created in:

```
<install-dir>/var/domains/<domain-name>/configurations/<configuration-name>/
```

You can specify another location for the Partition and add a pre-existing Partition to a configuration. The Management Console provides a rich configuration experience for a Partition and is discussed in “Using Partitions” in the *Management Console User's Guide*. Most configuration data for the Partition and its services is captured in its partition.xml file described in Chapter 35, “Partition XML reference.”

Figure 3.1 Partition Footprint



Running Partitions

Partitions are typically run under the control of a management agent within a configuration, but they can also be run directly from the command line as unmanaged Partitions. In both cases the Partition requires that a Smart Agent (osagent) be running in the same sub-net on the same Smart Agent port.

See “Using Partitions” in the *Management Console User's Guide* for information about managing Partitions within a configuration.

Running unmanaged Partitions

To run an unmanaged Partition (not managed by SCU), use the following command:

```
partition [-path <my_partition_path>]
```

If no `-path` is specified, then the current directory is used.

The full list of Partition arguments is available in the following tables. Many of these arguments are for use by the management agents and not by a user.

```
partition [<-options>] [-path <partitionpath>] [-management_agent <true|false>
[-management_agent_id <id>]] [-no_user_services] [-unique_cookie <cookie>]
```

`<-options>` are the usual Java options and VM system properties recognized by the Partition.

Note Options that are typically static, and pertinent to both managed and unmanaged Partitions, are best encapsulated in the Partition's configuration files.

Table 3.1 Partition command options

Option	Description
`-Dlog4j.configuration`	Path to the Partition's log4j configuration file. Default is `<partitionpath>`/adm/properties/logConfiguration.xml
`-Dlog4j.configuration.update.delay`	Specifies the period, in milliseconds, between checks for updates to the log4j configuration file. Default is 60000 milliseconds (1 minute).
`-Dpartition.ignore_shutdown_on_signal=<true|false>`	Use this property to decide whether to ignore shutdown signals and wait for a shutdown request via the Partition's management interface(s). **Note:** UNIX sends a *Ctrl+C* signal to all processes in a process group. A Partition in control of its own life cycle would not set this. When the Partition is invoked by some parent controlling process, such as the SCU, then this would be set to `true` to ensure that the Partition does not immediately exit when the parent is issued a shutdown signal.
`-Dpartition.default.smartagent.port`	Overrides the User ORB Smart Agent port and overrides all Partition configuration. This property is only overridden by `-Dvbroker.agent.port`. Typically used by a parent controlling process, such as the SCU.
`-Dpartition.default.smartagent.addr`	Overrides User ORB Smart Agent `addr` property and overrides all Partition configuration. Is only overridden by `-Dvbroker.agent.addr`. Typically used by a parent controlling process, such as the SCU.
`-Dvbroker.agent.port`	Ultimate override for the User ORB Smart Agent port. This is typically never used by a parent controlling process, but it may be used by a command-line user.

Table 3.1 Partition command options (continued)

Option	Description
-Dvbroker.agent.addr	Ultimate override for the User ORB Smart Agent addr. Typically never used by a parent controlling process, but it may be used by a command-line user.
-Dpartition.management_domain.port	Sets the Management ORB Smart Agent port. Default 42424. Typically used by a parent controlling process, such as the SCU.
-DTomcatLoaderDebug	Sets the Web Container debug level. Default 0 (zero).

Table 3.2 Partition command available arguments

Arguments	Description
-path <partitionpath>	Partition footprint path.
-management_agent <true|false>	The default is false which disables the Partition management agent and runs a standalone Partition. To enable the Partition management agent, set to true.
-management_agent_id <id>	Sets the identity to be used for the Partition's management interface object name.
-unique_cookie <cookie>	Sets the cookie to be used to construct unique identities in the Partition. In particular, used to construct default external interface names. The default is: <host><partitionpath>.
-no_autostart_user_services <true|false>	If set to true, disables the autostart of user domain Partition services that are configured to be started.

Running managed Partitions

Managed Partitions are started when the configuration to which they belong starts. Typically the Partition starts according to a default mechanism, but you can configure additional command-line options to be passed at creation-time. Or, you can edit configuration.xml. Open the file, search for <partition-process>, and find the <arguments> data block. Insert new command-line arguments within <argument> tags.

Running Partitions with Optimizeit Profiler or ServerTrace

You can configure the Profiler and ServerTrace using the Management console. If you choose not to use the Management Console you must set the following environment variables for running Partitions with either Optimizeit Profiler or ServerTrace:

```
OPTIT_HOME=<server_trace_root when running server trace|optimize_root when running profiler>
```

Windows

```
PATH=<server_trace_root when running server trace|optimize_root when running profiler>;%PATH%
```

Solaris

```
LD_LIBRARY_PATH=<server_trace_root when running server trace|optimize_root when running profiler>/lib
```

To run a Partition with ServerTrace standalone:

Partition configured to use JDK 1.5:

```
partition -classpath <Server Trace Home>\lib\optit.jar -path <path to
partition> -management_agent true -management_agent_id <config_name>/
<hub_name>/<partition_name>  -no_autostart_user_services false -optimizeithome
<server_trace_home> -Xrunoii:filter=>server_trace_root</filters/
BES.oif,port=1473 -Xbootclasspath/p:<Server Trace Home>\lib\oibcp\
oibcp_sun_150_06.jar
```

Partition configured to use JDK 1.4:

```
partition -classpath <Server Trace Home>\lib\optit.jar -path <path to
partition> -management_agent true -management_agent_id <config_name>/
<hub_name>/<partition_name>  -no_autostart_user_services false -optimizeithome
<server_trace_home> -Xrunoii:filter=>server_trace_root</filters/
BES.oif,port=1473 -Xbootclasspath/p:<Server Trace Home>\lib\oibcp\
oibcp_sun_142_05.jar
```

To run a Partition with Optimizeit Profiler standalone:

```
partition -path <path_to_partition> -include_cfg partition_optimizeit.config -
optimizeithome <optimizeit_root> -Xrunpri:filter=<optimizeit_root>/filters/
BES.oif,port=1470
```

Solaris:

To run a Partition with Optimizeit Profiler or ServerTrace standalone, you must edit the `partition_trace.config` or `partition_optimize.config` respectively, adding

```
vmparam -Xboundthreads
```

Note When re-starting BAS's Partition with ServerTrace enabled, the Partition may not stop cleanly and appear to hang. This happens because the SNMP thread in the ServerTrace subsystem does not shut down. Use the kill operation to stop the partition and then use the start operation to restart it.

Partition logging

The Partition uses log4j for its logging mechanism. It is configured using a DOMConfigurator from the file `<partitionpath>/adm/properties/logConfiguration.xml`. The default configuration is to log in a text layout to rolling log files in `<partitionpath>/adm/logs`. The Partition `logConfiguration.xml` file is monitored for updates with a default check interval of 1 minute. See previous table of Partition options for information about configuring the configuration file and monitor check interval.

Any output sent to `System.out` or `System.err` is redirected as log4j events to the logs. `System.out` is logged at the INFO level and `System.err` is logged at the ERROR level.

If your application uses log4j then to configure application logging you should edit the Partition's `<partitionpath>/adm/properties/logConfiguration.xml` file.

Configuring Partitions

Partitions offer a variety of fully-configurable services. This section discusses how to work with Partition services, including archives, security, application services, and statistics.

Application archives

Application components are hosted in the Partition itself. You can dynamically deploy application archives to Partitions prior to running them or when they are running. If the application archive is already hosted by the Partition, then it is unloaded and the new archive loaded. To deploy modules to a Partition, simply right-click its icon in the Management Console's Navigation Pane, and select Deploy Modules. The deployed modules appear in the Partition footprint, as shown in the Partition Footprint figure in “Creating Partitions” on page 14.

You can also host modules at locations outside the Partition footprint. To do so, open the `partition.xml` file for the Partition whose module paths you want to configure. Search for the <archives> node. Within this node, you can configure archive repositories for all your archives by type, or provide the location of a specific archive that you want hosted outside the Partition repositories. See “<archives> element” on page 345 for syntax.

In the Management Console, archives hosted within the Partition's footprint are called “Deployed Modules”. Archives that are hosted outside the Partition's footprint are called “Hosted Modules”.

Working with Partition services

The Partition allows you to specify which services will run within it and how they will behave in the context of the Partition instance. You can configure the Partition to automatically start some or all of its services at Partition startup. You can specify the order in which Partition services start and shut down. Additionally, you can configure which Partition services are configurable through the Management Console. Again, the `partition.xml` file captures this information as attributes of its `<services>` element.

Partition handling of services

The `<services>` element has four attributes, which are:

autostart	The services to be started with the Partition.
startorder	The startup order imposed on the Partition services included in the `autostart`.
shutdownorder	The shutdown order imposed on the Partition services running at shutdown.
administer	The Partition services that will appear in the Management Console as configurable.

To set any one of these attributes, use either the Management Console or search the Partition's `partition.xml` file for the `<services>` node. The valid value for each attribute is a space-separated list of Partition service names, which are read left to right. For example, if you wanted to shutdown a Partition service named `ejb_container` before a service named `transaction_service`, you would set the value of `shutdownorder` to:

```
ejb_container transaction_service
```

Configuring individual services

Each Partition service is configurable within the context of its Partition parent. The `partition.xml` file captures information about individual services in the `<service>` node, the child node of `<services>`. In addition, you can use the `<properties>` sub-element within `<service>` to set service-specific properties that do not come under the auspices of the Partition's runtime executable.

If your services are to be included in the `service` node lists, you must define them with a `service` data block and give them a unique name using the `name` attribute. For a full description of the attributes that are configurable for Partition services, see Chapter 35, “Partition XML reference.”

Configuring VisiNaming Service Clusters for AppServer

To configure a VisiNaming cluster for AppServer, in addition to the steps outlined in Configuring the VisiNaming Service Cluster in the VisiBroker for Java Developer's Guide, and in Configuring the VisiNaming Service Cluster section in the VisiBroker for C++ Developer's Guide, you must add the name of the factory to the property `jns.name` in `partition.xml`.

Gathering Statistics

Each Partition has a Statistics Agent that can be enabled for the short-term gathering of statistics data. The data is stored onto disk, and is viewable using the Management Console. Statistics are collected in *snapshots* performed at a specified interval, and are cleaned up (removed from disk) at discreet intervals and after the collection period. This function is called *reaping*.

You can enable, disable, and configure statistics gathering using the Management Console or by setting attributes in the `<statistics.agent>` attribute of `partition.xml`. For more information, see Chapter 35, “Partition XML reference.”

Security management and policies

Each Partition can have its own security settings. You can specify the security manager to use for each Partition by specifying a valid security class. You can also set the Policy to use for that manager (generally using a `.policy` file). You can configure security using either the Management Console or by setting the attributes of the `<security>` node of `partition.xml`. For more information, see Chapter 35, “Partition XML reference.”

Important The default security profile shipped is not SSL-enabled. If your Partition requires secure transport, change the security profile for your Partition to the `ssl_enabled` profile as follows:

1. Open the Management Console and expand the Configurations node in the left pane.
2. Right-click on Partition and choose Properties.
3. Click on the Security tab to bring it forward.
4. Select ssl-enabled from the Security Profile dropdown menu.

Classloading policies

You can configure the Partition's classloading policies, including the prefixes to load, the classloader policy, and whether or not to verify JARs as they are being loaded. You can configure classloading either using the Management Console or by setting attributes of the `<container>` node of `partition.xml`.

The `system.classload.prefixes` attribute takes a comma-separated list of resource prefixes as its value. These prefixes are delegated from the custom classloader to the system classloader prior to attempting its own load. The `classloader.classpath` attribute contains a semicolon-separated list of JARs to be loaded by each instance of the application classloader. To verify the JARs as they load, set the `verify.on.load` attribute to `true`, the default.

The classloader policy is set in the `classloader.policy` element. There are two acceptable values:

per_module	Creates a separate application classloader for each deployed module. This policy is required for hot deployments (deployments while the Partition is running).
container	Loads all deployed modules in the shared classloader. This policy prevents the ability to hot deploy.

Partition Lifecycle Interceptors

You can use Partition Lifecycle Interceptors to further customize your implementation. Partition Lifecycle Interceptors allow you to perform operations at certain points in a Partition's lifecycle. You deploy a Java class that implements:

```
com.borland.enterprise.server.Partition.service.PartitionInterceptor
```

and contains code to perform operations at one or more of the following interception points:

- At Partition initialization before any Partition services (Tomcat, for example) are created and initialized.
- At Partition initialization after any services are started but prior to the loading of any modules.
- At Partition startup after all Partition services have loaded their respective modules.
- At Partition shutdown before Partition services have unloaded their respective modules but prior to the services themselves shutting down.
- At Partition termination after Partition services have been shut down.

Partition Interceptors have a variety of uses, including pre-loading JARs prior to startup, inserting debugging operations during module loading, or even simple messaging upon the completion of certain events.

For information about how to implement a Partition Lifecycle Interceptor, see "Implementing Partition Interceptors" on page 257.

JMX support in Partitions

The Java Management Extensions (JMX) defines an architecture, the design patterns, the APIs, and the services for application management in the Java programming language. JMX is used to get and set configuration and performance information, and send and receive alerts for Java programs. The JMX architecture is composed of Management Beans (MBeans), a JMX agent, and JMX adaptors.

Each AppServer Partition hosts a fully functional JMX agent. AppServer uses the MX4J HTTP adaptor without modification, but the MX4J RMI adaptor implementation has been modified to suit the AppServer Partition in the following ways:

- Uses the VisiBroker for Java ORB for RMI-IIOP remoting.
- RMI Connector is modified to use RMI-IIOP as the underlying transport. This connector works both in JDK 1.4 and JDK 1.5 environments.
- The `RMIConnectorServer` is made available by both the Smart Agent component of AppServer and the JMX Service URL.

The goal of the AppServer JMX implementation is to expose key runtime aspects of an AppServer Partition as MBeans. J2EE Enterprise Management (JSR-77) support is available in AppServer Partitions and has two fundamental aspects:

1. Instrumentation of the J2EE Server (Partition) following the JSR-77 model.
2. JSR-77 EJB interface to those JMX MBeans.

See "Partition MBeans" on page 24 for a list of Mbeans provided by Borland for the Partition.

For more details about MX4J see `http://mx4j.sourceforge.net`, and for the JMX JSR-3 specification see `http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html`.

Configuring the JMX Agent

The JMX Agent is configured in the `jmx` section of the `partition.xml` file. The MBean Server is enabled by default and starts when the Partition is started. By default, the RMI-IIOP adaptor is enabled and the HTML adaptor is disabled. The HTML adaptor is supported only for JDK 1.4. See "<jmx> element" on page 340 for more details about the `jmx` element.

JMX agent configuration can also be accomplished using the Borland Management Console. See "JMX Agent properties" in the *Management Console user's Guide* for more information.

Partition monitoring

A JMX agent is embedded in each Partition, along with the HTTP and RMI-IIOP adaptors, to enable Partition monitoring with a JMX client. This means it is possible for a JMX-enabled client (such as the MC4J Management Console, provided with your AppServer installation) to automatically detect these MBeans and plot graphs for the changing values. For more information about using the MC4J Management Console see "Using the JMX console" in the *Management Console User's Guide*.

Note When you right-click on a partition name, the Launch JMX Console... menu option will be enabled only if you have enabled the JMX agent in the partition properties.

If you enable the `http.adaptor` and `xslt.processor` elements in `partition.xml`, you can also use a Web browser to monitor the Partition by way of the HTTP adaptor. The default location of the HTTP adaptor is `http://localhost:8082`. This adaptor is however not supported for JDK 1.5's JMX Agent. See "<jmx> element" on page 340 for information about configuring the HTTP adaptor.

Note In this release the emphasis is on monitoring rather than actively managing the Partition using JMX. It is recommended that you continue using the Borland Management Console to manage (stop, start) Partitions. See "Using Partitions" in the *Management Console User's Guide* for more information.

Using the RMI-IIOP connector in MC4J console

The pluggable JMX Connector in BAS implements JSR-160, and is based on RMI/IIOP protocol. The connector works both in JDK 1.4, and JDK 1.5 environments. Run the MC4J console. See "Launching the JMX console standalone" in the *Management Console User's Guide* for information on how to run the console. The JMXConnector Mbean is displayed along with the corresponding attributes, operations and notifications.

You can also use a standalone client to connect to the portable JMX connector running in BAS partition. The following code snippet is taken from a sample JMX connector client:

```
HashMap props = new HashMap();
  props.put("jmx.remote.credentials",new String[] { "admin", "admin" });
  org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
  props.put( JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES,
             "com.borland.jmx.remote.provider" );
  props.put( "jmx.remote.orb", orb );
  ClassLoader cl = Thread.currentThread().getContextClassLoader();
    JMXConnector m_connector = null;
    MBeanServerConnection m_connection = null;
    try
  {
```

```
Thread.currentThread().setContextClassLoader(JMXConnectorFactory.class.getClass
Loader());
      JMXServiceURL m_jmxurl;
      m_jmxurl = new JMXServiceURL("service:jmx:iiop://null/
corbaloc::xyz:42222/j2eeSample_WelcomePartition");
      m_connector = JMXConnectorFactory.connect(m_jmxurl,props);
      m_connection = m_connector.getMBeanServerConnection();
      Set set = null;
      set = m_connection.queryNames(new ObjectName("*:*"), null);
      System.out.println("Set.isEmpty : " +  set.isEmpty()); //Just to see if
connection is OK
 }
  catch( IOException io )
  {
      io.printStackTrace();
  }
 catch (MalformedObjectNameException ex) {
   ex.printStackTrace();
  }
  finally
  {
      Thread.currentThread().setContextClassLoader(cl);
  }
```

In the above code, `xyz` represents the hostname and `42222` represents the port on which the connector is started.

To run the JMX Connector client:

1. Build the client code as an AppClient container.
2. Start the sample partition (For example, WelcomePartition in the j2eeSample configuration) with the RMI-IIOP connector enabled.
3. Run the AppClient container. See the secure cart example in the `<bas_install>/AppServer/examples/security/securecart` directory.

Configuring the RMI-IIOP connector

You can configure the RMI-IIOP connector using the BAS Management Console. To do so:

1. Open the Management Console.
2. Right-click on the partition name in the left pane and select Properties from the resulting menu.
3. Click on the JMX Agent tab to bring it forward.
4. Check the Enable JMX checkbox if it is not already checked.

Note If you have JDK 1.5 or later version, you have the option of using the built in JMX implementation. Check the Use JDK JMX checkbox in order to use this implementation of JMX.

You can specify a port number for the pluggable RMI-IIOP Adaptor in the UI or configure it in the `partition.xml` file available in the `<bas-install>/var/domains/configurations/<configuration-name>/mos/<partition-name>/adm/properties` folder. Use the following tag format for specifying RMI-IIOP Adaptor port number:

```
<jmx>
   <mbean.server enable="true"/>
   <mlet.service enable="false"/>
   <http.adaptor port="8082" enable="true" host="localhost" processor.name=""
socket.factory.name="" authentication.method="none">
   <xslt.processor File="" PathInJar="" enable="true" UseCache="true"
LocaleString="en"/>
   </http.adaptor>
   <rmi-iiop.adaptor enable="true" port="42222"/>
</jmx>
```

If you specify the port number as 0 or an invalid string, or you do not specify a port number, the pluggable RMI-IIOP Connector will be started at a random port number.

`mlet` is a JMX service which allows you to download and register MBeans from a remote MBean server. You can enable it by checking the Enable mlet services checkbox.

Note If you try to register an MBean via the MEJB interface, the MBean class should be present both on the client and the server side.

Creating a secure JMX client

JMX inherits its security from the partition's Management domain security settings. In order for a JMX client to communicate with a JMX server in a secure partition, the client will need to be secure as well in the management domain.

If you are using MC4J console, you can use it as is whether the console is communicating with a secure or non-secure server. When you launch the client, it automatically creates a secure client side ORB if the server is running in a secure partition. To use an MEJB client you must make the client secure. Refer to the secure cart example in the `<bas_install>/AppServer/examples/security/securecart` directory for information on how to create a secure MEJB client.

Remoting mechanism for the Server side JMX components in BAS uses the management domain VisiBroker ORB of the partition. The security of management domain hence JMX Server is on by default. The remote JMX clients that need to communicate with the JMX server which is hosted in a BAS partition must have appropriately created secure ORB.

In BAS, the JMX clients talk to JMX Server over standards based JMX Connector that uses RMI-IIOP. See the code sample in “Using the RMI-IIOP connector in MC4J console” on page 21 for details on how a JMX client can establish a connection to a JMX Server in the BAS partition, and also how to pass an ORB.

Switching between JDK 1.5 and MX4J JMX agents

When you run the partition using JDK 1.5, the partition uses the MX4J Agent by default. In order to use the JDK 1.5's JMX agent, check the Use JDK JMX checkbox in the Partition Properties dialog or make sure that the following line in `<bas-install>/var/domains/base/configurations/<configuration-name>/mos/<partition-name>/adm/properties/partition-server.config` file is commented:

```
# MX4J as the default MBean Server
vmprop javax.management.builder.initial=mx4j.server.MX4JMBeanServerBuilder
```

When you check the Use JDK JMX checkbox in the Management Console indicating that you are using JDK 1.5, the above lines get automatically commented out. To make MX4J as the default JMX agent for all partitions, add the above line in `<bas-install>/bin/partition.config` file.

Partition level properties

The following extract from the `partition.xml` file shows the property for the pluggable JMX RMI-IIOP connector:

```
<partition version="6.6" name="WelcomePartition" description="A partition
hosting a number of pre-deployed BAS example applications. More examples are
available in the archive repository">
  <jmx>
  <mbean.server enable="true" />
  <mlet.service enable="false" />
  <http.adaptor port="8082" enable="true" host="localhost" processor.name=""
socket.factory.name="" authentication.method="none">
  <xslt.processor File="" PathInJar="" enable="true" UseCache="true"
LocaleString="en" />
  </http.adaptor>
  <rmi-iiop.adaptor enable="true" port="42222"/>
  </jmx>
```

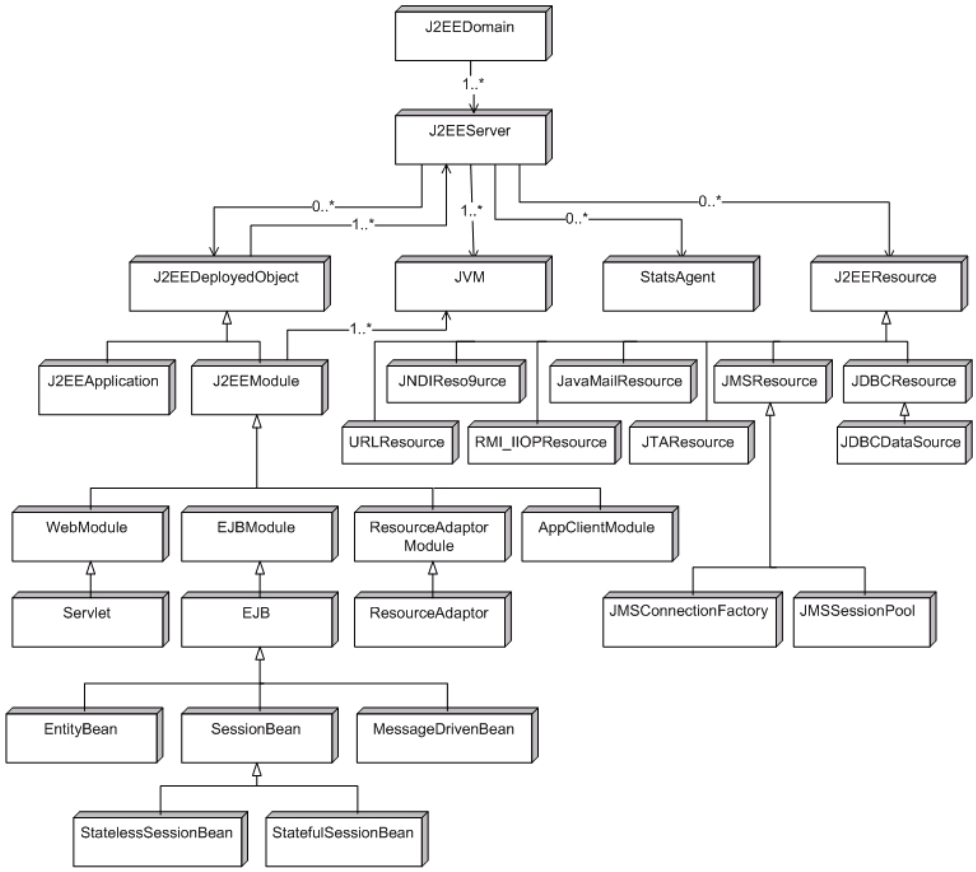
Partition MBeans

This section describes the MBeans provided by Borland for the Partition. If you want to deploy your own MBeans see "Deploying custom MBeans" on page 26.

The naming convention for the MBeans is from JSR-77. Please refer to JSR-77 section 3.1.1 for a complete description of this naming convention.

The following illustration shows you a high level overview of the MBeans implemented for the AppServer Partition.

Figure 3.2 Partition MBeans



The MBeans are available only if you enable the JMX feature. The level of information that is provided in the MBeans depends on the statistics level you set in the Performance Tuning Wizard. Complete JSR-77 statistics will be available only if you select the statistics level as “maximum”. You will get only a subset of the JSR-77 statistics if you select “medium” or “minimum”.

Note Certain MBeans have operations that are exposed specifically for invocation by other related MBeans. These operations should not be invoked from the JMX (MC4J) Console. MC4J Console does not distinguish between these operations and operations that are relevant to the user. If the operation requires complex input parameter then you may run through the method invocation wizard and experience a NullPointerException. Even with simple input parameters, you have to explicitly select the argument in the wizard. If not selected, the argument defaults to null and MC4J cannot deal with this.

Deploying custom MBeans

To deploy custom MBeans:

1. Decide where to initialize your MBeans (possible places include a servlet or Startup class).
2. Add the JNDI lookup code to locate the MBeanServer.

   There is currently no standard way for the application to locate the JMX agent (server) in an application server infrastructure. So, what you should do is follow the model that is similar to the J2EE approach of looking up the ORB, UserTransaction, etc, that is shown in the code snippet below.

   Note that JMX agent is a server side feature, and the lookup operation in the client VM will not be able to find a pointer to the server instance as the server is only initialized in the server VM. A client will need to use the RMI-IIOP connection to connect to a remote JMX agent.
3. Register the MBeans.

   Once the MBean server is located, the registration step is just a normal process. Note that your MBeans have to be in the classpath when you are doing these steps. Also note that depending upon where you end up having your JMX code, you may want to clean up your MBeans (unregister them) when your module is unloaded. For example if your MBeans were registered in the Startup class, you may want to unregister your MBeans in a Shutdown class.

The following code snippet shows how to deploy the custom MBean.

```
javax.naming.Context  context = new javax.naming.InitialContext();
server = (MBeanServer)context.lookup("java:comp/env/jmx/MBeanServer");

// Create an ObjectName for the MBean
ObjectName name =
          new ObjectName("qa:mbeanName=helloworld,mbeanType=Standard");
com.borland.enterprise.qa.mbeans.helloworld.HelloWorld hello =
          new com.borland.enterprise.qa.mbeans.helloworld.HelloWorld();
server.registerMBean(hello,name);

// Invoke a method on it
server.invoke(name, "reloadConfiguration", new Object[0], new String[0]);

// Get an attribute value
Integer times = (Integer)server.getAttribute(name, "HowManyTimes");
```

Using Management EJB (MEJB)

The Management EJB (MEJB) allows you to get access to the managed objects defined by the JSR-77 management model. It is an interface to the JMX MBeans. MEJB is a stateless session bean which can be located using JNDI. It allows you to query objects, get metadata for an object and set and get attributes of an object.

Deploying the MEJB

In the BAS footprint, the MEJB is located as a pre-built jar (`mjeb_beans.jar`) in the `<bas_install>/etc/prebuilt-ejbs/mejb` directory. Deploy this MEJB to the desired partition before running the client. To deploy the MEJB:

1. Open the Management Console.
2. Right-click on the partition name and select Deploy modules... from the menu. The Borland AppServer Deployment Wizard will open.
3. Click on the Add... button and navigate to the mejb_beans.jar file and click OK.
4. Check the Generate stubs checkbox if not already checked.
5. Click on the Next button.
6. Click on Finish.

Writing an MEJB client

See the secure cart example provided in the `<bas_install>/AppServer/examples/security/securecart` directory for details on how to write an MEJB client.

Before you use the client, you must generate the stubs for it. To create the stubs:

1. Create an EAR file with the client code and the mejb_beans.jar in it.
2. Run the following command from iastool:

   ```
   iastool -gendeployable -src <EAR_filename>
   ```

   The stubs will be created in the EAR file.

To deploy and run the MEJB client do the following:

1. Build the appclient code using the following deployment descriptors:
   - Add the following in the `application-client-borland.xml` file:

     ```
     <application-client>
       <ejb-ref>
           <ejb-ref-name>ejb/mgmt/MEJB</ejb-ref-name>
           <jndi-name>j2ee/management/MEJB</jndi-name>
       </ejb-ref>
     </application-client>
     ```

   - Add the following in the `application-client.xml` file:

     ```
     <application-client>
       <display-name>mejb_client</display-name>
       <ejb-ref>
           <ejb-ref-name>ejb/mgmt/MEJB</ejb-ref-name>
           <ejb-ref-type>Session</ejb-ref-type>
           <home>com.borland.management.mejb.BesManagementHome</home>
           <remote> com.borland.management.mejb.BesManagement</remote>
       </ejb-ref>
     </application-client>
     ```

2 Start the partition in which you have deployed the `mejb_beans.jar`.

3 Run the appclient container with the following command:

```
appclient <EAR_filename> -uri <client_jarfile>
```

Alternately, you can generate the stubs and make sure to include it in the CLASSPATH of the client application.

Event notification with MEJB

The MEJB allows clients to receive notifications whenever the state of specified managed objects change. The example below shows you how to add a notification listener to the MEJB:

```
Context context = new InitialContext();
//look up jndi name
Object ref = context.lookup("j2ee/management/MEJB");
//look up jndi name and cast to Home interface
besManagementHome = (ManagementHome) PortableRemoteObject.narrow(ref,
     BesManagementHome.class);
Management mejb = besManagementHome.create();
ObjectName objectname = new
ObjectName("JMImplementation:type=MBeanServerDelegate");
ListenerRegistration lr = mejb.getListenerRegistry();
lr.addNotificationListener(objectname, new MyNotificationListener(),
     new MyNotificationFilter(), new HandBackObject(4040))
```

In the above example, `MbeanServerDelegate`, a system MBean, is used to add a notification listener. The `MyNotificationListener` passed as the notification listener to be triggered in case of an event basically implements the `NotificationListener` interface and provides the action to be taken whenever a notification is received.

The `MyNotificationFilter` is a custom filter which is an implementation of `NotificationFilter` interface. The notification filter is used to specify conditions under which the notifications should be sent. The client can also choose some pre-defined notification filters like `AttributeChangeNotificationFilter` and `NotificationFilterSupport` in `javax.management` package and `MbeanServerNotificationFilter` in `javax.management.relation` package. The `HandBackObject` is a simple object which is delivered to the client whenever the notification is triggered.

Note While using custom filters, make sure that the class definition is available on both the server and the client side.

Running multiple partitions with MEJB

The sample pre-built MEJB jar in the <bas-install>/etc/prebuilt-ejbs/mejb directory can be deployed to several running partitions at the same time. However, make sure that you use a unique JNDI name for each partition. You can do this by using the DD editor. This is necessary in order to ensure that clients can uniquely identify the corresponding MEJBs. Also, on the client side the application container deployment descriptors should carry the JNDI name of the intended MEJB.

Locating the JMX agent

If you are unsure whether the JMX agent is running, use one of the following methods to locate it.

- **Use `osfind`.**

  To achieve uniqueness, the name of the JmxAgent running inside a Partition is derived as follows:

  ```
  <configName>_<hubName>_<partitionName>_JmxAgent
  ```

  For example, a Partition with the following specs for each of the components:

  ```
  <configName=ojms>_<hubName=XYZ2>_<partitionName=openjms>_JmxAgent
  ```

  will be registered with the following name in `osagent`:

  ```
  ojms_XYZ2_openjms_JmxAgent
  ```

  will show the following output on running `osfind`:

  ```
  prompt% set OSAGENT_PORT=<management_port>
  prompt% osfind
  osfind: Following are the list of Implementations started manually.
          HOST: 172.20.20.53
          REPOSITORY ID:
  RMI:javax.management.remote.rmi.RMIServer:0000000000000000
          OBJECT NAME: ojms_XYZ2_openjms_JmxAgent
  ```

- **Use the Borland Management Console.**
  When you right-click on the Partition, and choose Launch JMX Console, the feature will only work if the JMX Agent is properly configured for the Partition and the Partition is running.

Thread pools

Default thread pool

A Partition can process multiple concurrent application requests using Visibroker thread and connection management. Threads are assigned to process application requests from a thread pool configured through an entity called a server engine. Several server engines, and hence thread pool configurations, are available in Visibroker. By default, a Partition uses a thread pool for server engine IIOP.

Partition properties relevant to the configuration of this pool can be browsed and edited in the Partition Properties dialog of a Partition displayed in the Borland Management Console. For more information see “Visibroker properties” in the *Management Console User's Guide*.

Configuration of the default thread pool is achieved through the partition `vbroker.properties` configuration file. The necessary properties are automatically configured in AppServer Partitions. The configuration file is located under <install_dir>/var/domains/<domain-name>/configurations/<config>/mos/<partition>/adm/properties/vbroker.properties, and they are described in “Managing threads and connections” in the *VisiBroker for Java Developer's Guide*.

Auxiliary thread pool

The auxiliary thread pool is used internally by a Partition to ensure that the default thread pool is used exclusively for application requests, and to prevent issues such as distributed deadlocks. The auxiliary thread pool is defined through configuration of a VisiBroker server engine called `aux_se`.

Configuration of the auxiliary thread pool is achieved through the partition `vbroker.properties` configuration file. The necessary properties are automatically configured in AppServer Partitions. They are described in the following table.

Property	Default value	Description
`vbroker.se.aux_se.host`	`null`	Specifies the host name that can be used by this server engine. The default value, `null`, means use the host name from the system. Host names or IP addresses are valid values.
`vbroker.se.aux_se.proxyHost`	`null`	Specifies the proxy host name that can be used by this server engine. The default value, `null`, means use the host name from the system. Host names or IP addresses are valid values.
`vbroker.se.aux_se.scms`	`aux_tp`	Specifies the Server Connection Manager name.
`vbroker.se.aux_se.scm.aux_tp.manager.type`	`Socket`	Specifies the type of Server Connection Manager.
`vbroker.se.aux_se.scm.aux_tp.manager.connectionMax`	`0`	Specifies the maximum number of connections to the server. The default value, 0 (zero), indicates that there is no restriction.
`vbroker.se.aux_se.scm.aux_tp.manager.connectionMaxIdle`	`0`	Specifies the time in seconds that the server uses to determine whether an inactive connection should be closed.
`vbroker.se.aux_se.scm.aux_tp.listener.type`	`IIOP`	Specifies the type of protocol the listener is using.
`vbroker.se.aux_se.scm.aux_tp.listener.port`	`0`	Specifies the port number used with the host name property. The default value, 0 (zero), indicates that the system will pick a random port number.
`vbroker.se.aux_se.scm.aux_tp.listener.proxyPort`	`0`	Specifies the proxy port number used with the proxy host name property. The default value, 0 (zero), indicates that the system will pick a random port number.
`vbroker.se.aux_se.scm.aux_tp.listener.giopVersion`	`1.2`	This property can be used to resolve interoperability problems with older VisiBroker ORBs that cannot handle unknown minor GIOP versions correctly. Valid values for this property are `1.0`, `1.1` and `1.2`.
`vbroker.se.aux_se.scm.aux_tp.dispatcher.type`	`ThreadPool`	Specifies the type of thread dispatcher used in the Server Connection Manager.
`vbroker.se.aux_se.scm.aux_tp.dispatcher.threadMin`	`0`	Specifies the minimum number of threads that the Server Connection Manager can create.
`vbroker.se.aux_se.scm.aux_tp.dispatcher.threadMax`	`0`	Specifies the maximum number of threads that the Server Connection Manager can create. The default value, 0 (zero), implies no restriction.
`vbroker.se.aux_se.scm.aux_tp.dispatcher.threadMaxIdle`	`0`	Specifies the time in seconds before an idle thread is destroyed.
`vbroker.se.aux_se.scm.aux_tp.connection.tcpNoDelay`	`true`	When this property is set to `false`, this turns on buffering for the socket. The default value, `true`, turns off buffering so that all packets are sent as soon as they are ready.Valid values are `true` and `false`.

For example, the following entries are required for the Partition:

```
##
## Configuration file located under <install_dir>/var/domains/<domain-name>/
##        configurations/<config>/mos/<partition>/adm/properties/
vbroker.properties
##
##
⋮
vbroker.se.aux_se.host=null
vbroker.se.aux_se.proxyHost=null
vbroker.se.aux_se.scms=aux_tp
vbroker.se.aux_se.scm.aux_tp.manager.type=Socket
vbroker.se.aux_se.scm.aux_tp.manager.connectionMax=0
vbroker.se.aux_se.scm.aux_tp.manager.connectionMaxIdle=0
vbroker.se.aux_se.scm.aux_tp.listener.type=IIOP
vbroker.se.aux_se.scm.aux_tp.listener.port=0
vbroker.se.aux_se.scm.aux_tp.listener.proxyPort=0
vbroker.se.aux_se.scm.aux_tp.listener.giopVersion=1.2
vbroker.se.aux_se.scm.aux_tp.dispatcher.type=ThreadPool
vbroker.se.aux_se.scm.aux_tp.dispatcher.threadMin=0
#
# By default the thread pool size is unlimited. Actual thread
# usage is controlled through default VBJ thread pool, but
# the maximum number of active threads will never exceed
# that currently active for the default thread pool.
#
vbroker.se.aux_se.scm.aux_tp.dispatcher.threadMax=0
vbroker.se.aux_se.scm.aux_tp.dispatcher.threadMaxIdle=0
vbroker.se.aux_se.scm.aux_tp.connection.tcpNoDelay=false
⋮
```

By default, a Partition will use the `aux_se` server engine. However, if required, it can be disabled through configuration of the following Partition system property:

```
useAuxiliaryThreadPool=false
```

Clustering J2EE Applications with Borland AppServer 6.6

Borland AppServer is designed for high reliability and availability. Part of its reliability and availability comes from its clustering capabilities. For information on clustering J2EE applications with Borland AppServer, see the white paper, Clustering J2EE Applications with Borland AppServer 6.6, at `http://support.borland.com/entry.jspa?externalID=4304&categoryID=391`.

Chapter

4

Web components

This section provides information about the web components which are included in the Borland AppServer (AppServer). For more information, see “Installing Borland AppServer on Windows” or “Installing Borland AppServer on Solaris and HP-UX” in the *Installation Guide*.

Apache web server implementation

The AppServer implementation of the open-source Apache web server version 2.2 (an httpd server) is HTTP 1.1-compliant and is highly customizable through the Apache modules.

Apache configuration

The Apache web server comes pre-configured and ready-to-use when it is initially started. Many modules are dynamically loaded during the Apache startup. You can later customize its configuration for the IIOP connector, clustering, failover, and load balancing with one or more web container(s). You can use the Management Console to modify the configuration file, or you can use the directives in the plain text configuration file: `httpd.conf`.

By default, the Apache `httpd.conf` file is located in the following directory:

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
  mos/<apache_managedobject_name>/conf
```

Otherwise, for the location of the `httpd.conf` file, go to the `configuration.xml` file located in:

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>
```

and search for the Apache Managed Object, `apache-process` `httpd-conf` sub-element attribute:

```
httpd-conf=
```

For information about configuring the `httpd.conf` file for the IIOP connector/redirector, see, “Modifying the IIOP configuration in Apache” on page 43 .

Apache configuration syntax

When you edit the `httpd.conf` file, you must adhere to the following configuration syntax guidelines:

- The `httpd.conf` files contain one directive per line.
- To indicate that a directive continues onto the next line, use a back-slash "\" as the last character on a line.
- No other characters or white space must appear between the back-slash "\" and the end of the line.
- Arguments to directives are often case-sensitive, but directives are not case-sensitive.
- Lines which begin with the hash character "#" are considered comments.
- Comments cannot be included on a line after a configuration directive.
- Blank lines and white space occurring before a directive are ignored, so you can indent directives for clarity.

Running Apache web server on a privileged port

Processes which access privileged ports on UNIX hosts must have appropriate permissions; the process must be started with the permissions of user `root`. Typically, only some processes in a particular configuration need to be started with root permissions. The `setuser` script helps configure AppServer to allow Apache web server start as root or with root permissions. See "Using the setuser tool to manage ownership" in the *Installation Guide* for information about using the `setuser` tool and multi-user mode.

Before starting the configuration, gather the following information about the system on which Apache is installed:

1. AppServer installation directory
2. User and group names for the account that the Agent should become after shedding its root UID, that is, the installation owner.
3. User and group names for the system root account (typically root/sys)

The following steps describe the procedure for configuring the Apache web server to run on port 80.

1. Ensure that the Management Hub and the configuration containing the Apache web server are not running.
2. Enable multi-user mode on the AppServer installation.
   a. Edit the property

      ```
      agent.mum.enabled.root.mo=true
      ```

      in

      ```
      <install_dir>/var/domains/base/adm/properties/agent.config
      ```

   b. Become `root`.
   c. Run the setuser script:

      ```
      setuser -u <user> -g <group> +m
      ```

      where <user> and <group> are the attributes of the installation owner account (as described in B above).

3 Start the Management Hub.

4 Edit the Apache web server properties in the Management Console.

   a Right click the Apache web server MO, and select Properties.

   b In the Properties dialog, select the Apache Process Settings tab.

   c Click More settings to open the Advanced Process Settings dialog.

   d Select the Platform Specific Settings tab.

   e In the Unix Settings group enter the user and group names for the system root account in the Start as user and Start as group fields (as described in C above).

   f Click OK to close the Advanced Process Settings dialog.

   g In the Properties dialog, select the Files tab, and select the httpd.conf file.

   h Change the User and Group directives to the user and group name values for the account that owns the AppServer installation (as described in B above).

   i Change the Listen directive to 80.

   j Click OK to close the Apache Properties dialog.

5 Start the configuration.

Using the .htaccess files

The Apache web server allows for decentralized management of configuration through the `.htaccess` files placed inside the web tree. These files are specified in the `AccessFileName` directive.

Directives placed in `.htaccess` files apply to the directory where you place the file, and all sub-directories. The `.htaccess` files follow the same syntax as the main configuration files. Since `.htaccess` files are read on every request, changes made in these files take immediate effect. To find which directives can be placed in `.htaccess` files, check the Context of the directive. You can control which directives can be placed in `.htaccess` files by configuring the `AllowOverride` directive in the main configuration files.

Apache directory structure

After installing the Apache web server, by default, the following Apache-specific directory structure appears in:

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
   mos/<apache_managedobject_name>/
```

Table 4.1 Apache-specific directories

Apache-specific Directory Name	Description
`cgi-bin`	Contains all CGI scripts.
`conf`	Contains all configuration files.
`error`	Contains all error html documents.
`htdocs`	Contains all HTML documents and web pages.
`icons`	Contains the icon images in `.gif` format.
`logs`	Contains all log files.
`proxy`	Contains the proxies for your web application.

Borland web container implementation

The Borland web container supports development and deployment of web applications. The Borland web container, which is based on Tomcat 5.5.12., is included in the AppServer. For more information, see “Installing Borland AppServer on Windows” or “Installing Borland AppServer on Solaris and HP-UX” in the *Installation Guide*.

The Borland web container is a sophisticated and flexible tool that provides support for Servlets 2.4 and JSP 2.0 specifications.

As a “Partition service”, all the Borland web container configuration files are located in each of your Partitions' data directory under:

```
adm/tomcat/conf/
```

By default, a Partition's data directory is located in:

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
  mos/<partition_name>/
```

For example, for a Partition named “standard”, by default the Borland web container configuration files are located in:

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
  mos/standard/adm/tomcat/conf/
```

Otherwise, for the location of a Partition data directory, go to the `configuration.xml` file located in:

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
```

and search for the Partition Managed Object, `partition-process` sub-element `directory` attribute:

```
<partition-process directory=
```

Servlets and JavaServer Pages

A *servlet* is a Java program that extends the functionality of a web server, generating dynamic content and interacting with web clients using a request-response paradigm.

JavaServer Pages (JSP) are a further abstraction to the servlet model. JSPs are an extensible web technology that uses template data, custom elements, scripting languages, and server-side Java objects to return dynamic content to a client. Typically the template data is HTML or XML elements, and in many cases the client is a web browser.

Servlets and JSPs are server components that normally run within a web server. Servlets are written as web server extensions separate from the HTML page, while JSP embeds the Java code directly in the HTML. At runtime, the JSP Java code is automatically converted into a servlet.

Servlets process web requests, pass them into the back-end enterprise application systems, and dynamically render the results as HTML or XML client interfaces. Servlets also manage the client session information, so that users do not need to repeatedly input the same information.

Typical web application development process

In a typical development phase for a web application:

1 The web designer writes the JSP components, and the software developer creates the servlets for handling presentation logic.

2 In conjunction, other software engineers write Java source code for servlets and the `.jsp` and `.html` for processing client request to the server-side components (EJB application tier, CORBA object, JDBC object).

3 The Java class files, `.jsp` files, and the `.html` files are bundled with a deployment descriptor as a Web ARchive (WAR) file.

4 The WAR file (or web module) is deployed in the Borland web container as a web application.

For more information about using the AppServer Deployment Descriptor Editor (DDE) to create a Web ARchive (WAR) file, see “Adding WAR information” in the *Management Console User's Guide*.

Web application archive (WAR) file

In order for the Borland web container to deploy a web application, the web application must be packaged into a Web ARchive (WAR) file. This is achieved by using the standard Java Archive tool `jar` command.

The WAR file includes the `WEB-INF` directory. This directory contains files that relate to the web application. Unlike the document root directory of the web application, the files in the `WEB-INF` directory do not have direct interaction with the client. The `WEB-INF` directory contains the following:

Directory/File name	Contents
`/WEB-INF/web.xml`	the deployment descriptor
`/WEB-INF/web-borland.xml`	the deployment descriptor with Borland-specific extensions.
`/WEB-INF/classes/*`	the servlets and utility classes. The application class loader loads any class in this directory.
`/WEB-INF/lib/*.jar`	the Java ARchive (JAR) files which contain servlets, beans, and other utility classes useful to the web application. All JAR files are used by the web application class loader to load classes from.

Borland-specific DTD

For the Borland-specific DTD information, see the DTD documentation.

Adding ENV variables for the web container

You add web container ENV variables for a Partition the same way you set any ENV variables for any Partition service; you use the `<env-vars>` element and insert the xml code within the `partition-process` sub-element.

Note When adding web container ENV variables, be sure to type space-separated, value pairs.

By default, all `configuration.xml` files are located in the following directory:

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
```

To add web container ENV variables for a Partition Managed Object, use the `env-vars` element and `env-var` sub-element and the following syntax:

```
<managed-object name="standard"> ...>
    <partition-process ...>
        <env-vars ...>
          <env-var name="name" value="value"/>
        </env-vars>
        ⋮
</managed-object>
```

where `<name>` is the ENV variable name and `<value>` is the value you want to set for the named ENV variable.

For example:

```
<managed-object name="standard"> ...>
    <partition-process ...>
        <env-vars ...>
          <env-var name="ABC" value="val_abc"/>
        </env-vars>
        ⋮
</managed-object>
```

Microsoft Internet Information Services (IIS) web server

The Microsoft Internet Information Services (IIS) web server is not included with any AppServer product offerings. However, AppServer does include the IIOP redirector which provides connectivity from the Borland Tomcat-based web container to the IIS web server, and from the IIS web server to a CORBA server. The IIOP redirector is supported for the following IIS versions:

- Microsoft Windows 2000/IIS version 5.0
- Microsoft Windows XP/IIS version 5.1
- Microsoft Windows 2003/IIS version 6.0

For more information, see "IIS web server to Borland web container connectivity" on page 52.

IIS/IIOP redirector directory structure

After installing any of the AppServer products, by default, the following IIS/IIOP redirector-specific directory structure appears in:

```
<install_dir>/etc/iisredir2/
```

Table 4.2 IIS/IIOP redirector directories

IIS/IIOP redirector-specific directory name	Description
`conf`	Contains all configuration files.
`logs`	Contains all log files.

Smart Agent implementation

The Smart Agent is a service that helps in locating and mapping client programs and object implementation. The Smart Agent is automatically started with default properties. For information on configuring the Smart Agent, see “Using the Smart Agent” in the *VisiBroker for Java Developer's Guide*.

The Smart Agent is a dynamic, distributed directory service that provides facilities for both the client programs and object implementation. The Smart Agent maps client programs to the appropriate object implementation by correlating the object or service name used by the client program to bind to an object implementation. The object implementation is an object reference provided by a server, such as the Borland web container.

The Smart Agent must be started on at least one host within your local network. When your client program invokes an object (using the `bind` method), the Smart Agent is automatically consulted. The Smart Agent locates the specified object implementation so that a connection can be established between the client and the object implementation. The communication with the Smart Agent is transparent to the client program.

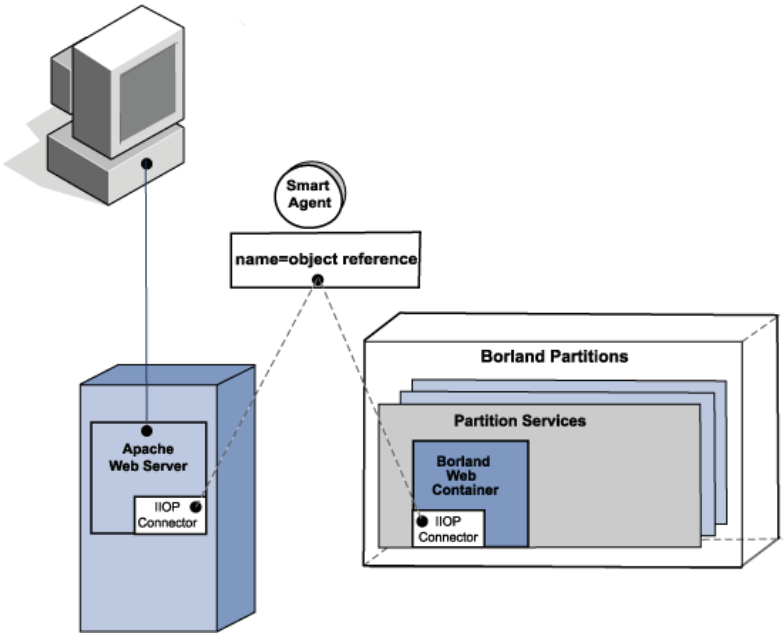
The following are examples of how the Smart Agent is used by the AppServer web components:

- Connecting Apache web server to a Borland web container.
- Connecting Borland web containers to Java Session Service (JSS).

Connecting an Apache web server to a Borland web container

As a distributed directory service, the Smart Agent registers an active ID of an object reference for the client programs to use. The following diagram shows the interaction between the client program binding to an object through the Smart Agent. In this example, the Apache web server is acting as a client and the Borland web container is acting as a server (and provides the object reference).

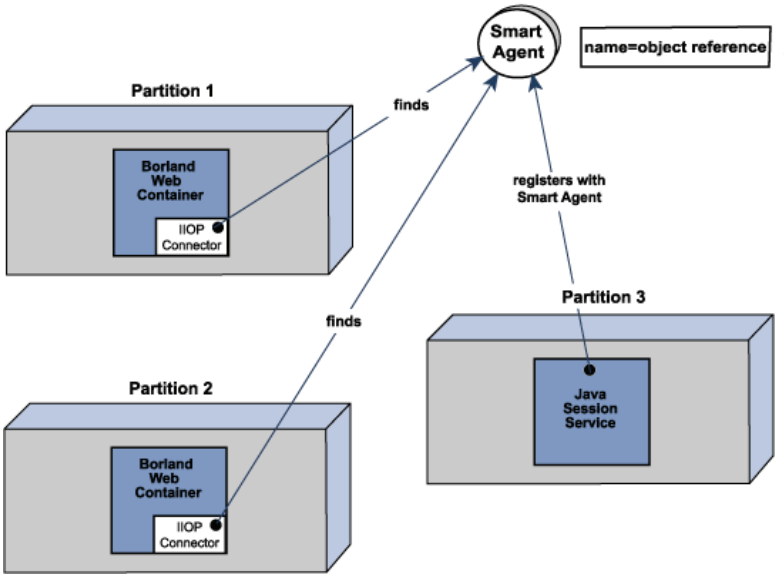
Figure 4.1 Client program binding to an object reference



Connecting Borland web containers to Java Session Service

In this scenario, there are multiple web containers that need to connect to a Java Session Service during start up. The Smart Agent is used to make a client/server connection. The following diagram shows multiple instances of the Borland web container. Each web container is acting as a client. During start up, the Smart Agent is consulted as a directory service to find and connect a JSS object reference. For more information about the Java Session Service (JSS), see Chapter 6, “Java Session Service (JSS) configuration.”

Figure 4.2 Connecting multiple web containers to a single JSS



Chapter

5

Web server to web container connectivity

This section describes the web server to web container IIOP connectivity provided in the Borland AppServer (AppServer). For more information, see “Installing Borland AppServer on Windows” or “Installing Borland AppServer on Solaris and HP-UX” in the *Installation Guide*.

For information about Apache to CORBA connectivity, see Chapter 8, “Apache web server to CORBA server connectivity.”

Apache web server to Borland web container connectivity

This section provides information about the following web components:

- an implementation of the open-sourced Apache web server version 2.2,
- the Tomcat-based Borland web container, and
- the IIOP connector, which provides connectivity from the Apache web server to the Tomcat-based Borland web container.

These web components are included in the AppServer. For more information, see “Installing Borland AppServer on Windows” or “Installing Borland AppServer on Solaris and HP-UX” in the *Installation Guide*.

Modifying the Borland web container IIOP configuration

The `server.xml` is the main configuration file for the Borland web container and is stored in your Partition's data directory:

```
adm/tomcat/conf/
```

For more information, see “Borland web container implementation” on page 36.

Within the `server.xml` file are the following lines of code that pertain to the IIOP connector configuration.

```
<Connector className="com.borland.catalina.connector.iiop.IiopConnector"
 name="tc_inst1 debug="0" minProcessors="5"
 maxProcessors="75" enableChunking="false" port="0"
 canBufferHttp10Data="true" downloadBufferSize="4096" shortSessionId="false" />
```

Use these lines of code and the following attributes to configure the Borland web container IIOP connector.

Table 5.1 IIOP connector attributes

Attribute	Default	Description
`name`	`tc_instl`	The name by which this connector can be reached by Apache and IIS web servers.
`debug`	0 (zero)	Integer that sets the level of debug information. When set to `0` (zero), the default, debug is turned off. To turn debug on, set to `1`. For very detailed debug messages, set to `99`.
`minProcessors`	`5`	The number of minimum threads previously created to service requests on this connector.
`maxProcessors`	`75`	The number of maximum threads that will be created on this connector to service requests.
`enableChunking`	`false`	Enables chunking behavior on the connector. To enable chunking, set this attribute to `true`. **Important:** To enable chunking, you must also set the servlet response header `Transfer-Encoding` value to `chunked`. For more information, see "Downloading large data" on page 48.
`downloadBufferSize`	`4096`	Defines the "chunked" buffer size employed when `enableChunking` is set to `true`. This directive accepts a numeric value >`0`. Essentially, the larger the number of bytes you set this directive to, the less the number of CORBA RPCs that are required to send the data to Apache or IIS. However, the larger you set this directive, the more memory will be consumed in servicing the transaction. Tuning this parameter allows you to fine-tune the performance characteristics. This enables the administrator to weigh the RPC costs against memory resource usage to optimize uploading on their system. **Note:** If an invalid value is presented (non numeric/negative number) then the default `4096` value is employed. For more information, see "Downloading large data" on page 48.
`port`	`0` (zero)	The IIOP connector port. If set to `0` (zero), the default, a random port gets picked. **Note:** If the corbaloc mechanism must be used to locate this connector from Apache or IIS, then `port` must be set to a value other than `0` (zero).

Table 5.1 IIOP connector attributes (continued)

Attribute	Default	Description
canBufferHttp10Data	true	When the HTTP protocol is less than 1.1 and the content length is not set on a servlet, the following two choices are available for the web container. It can buffer up the data, compute the content length, and then send the response or it can raise an error message. To avoid buffering the data and consuming memory, set this attribute to false.For more information, see "Browsers supporting only the HTTP 1.0 protocol" on page 49.
shortSessionId	false	To enable the use of the Borland "collapsed locator" feature that reduces the size of the IIOP connectors session id, set to true. By default, this attribute is disabled; set to false. The IIOP connector uses a stringified object reference (IOR) as a means to associate a client with a given Borland web container (service affinity). Some network routers and browser implementations are unable to cope with the large payload stored within the Session Id Cookie. To resolve this problem, Borland has developed the collapsed locator string that allows service affinity to be implemented using a much shorter session id string. Setting shortSessionId to true enables this feature, setting it to false or omitting the parameter entirely, defaults to the conventional IOR-based solution. **Note:** The "collapsed locator" is essentially an encoded CORBALOC string. Resolving a CORBALOC string to an object-reference is a more costly operation than the equivalent operation on an IOR. To alleviate this extra cost, the Apache web server will maintain a look-aside list of encoded CORBALOC strings to their resolved object references. This will incur a slight increase in memory use per Apache web server. **Important:** In order for the shortSessionID feature to work correctly, you **must** specify a port value for the IIOP connector; you cannot use the port default value of 0 (zero).

Modifying the IIOP configuration in Apache

The httpd.conf file is the global configuration file for the Apache web server. Within the httpd.conf file are the following lines which pertain to the IIOP connector.

Windows

```
LoadModule iiop2_module <install_dir>/lib/<apache_managedobject_name>/
     mod_iiop2.dll
   IIopLogFile <install_dir>/var/domains/<domain_name>/
configurations/<configuration_name>/mos/<apache_managedobject_name>/
logs/mod_iiop.log
   IIopLogLevel error
   IIopClusterConfig <install_dir>/var/domains/<domain_name>/
configurations/<configuration_name>/mos/<apache_managedobject_name>/
conf/WebClusters.properties
   IIopMapFile <install_dir>/var/domains/<domain_name>/
configurations/<configuration_name>/mos/<apache_managedobject_name>/
conf/UriMapFile.properties
```

Use these lines of code to configure the Apache web server IIOP connector.

Table 5.2 IIOP directives for Apache

Directive	default	Description
LoadModule	<install_dir>/lib/<apache_managedobject_name>/mod_iiop2.dll	Enables Apache 2.2 to load the IIOP connector. This directive instructs the Apache web server to load the Apache mod_iiop2 module from the location specified. Once the module is loaded, the following four directives are required to enable the IIOP connector to locate the web container(s) or CORBA server(s) it must communicate with and perform other functions.
IIopLogFile	<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<apache_managedobject_name>/logs/mod_iiop.log	Specifies the location where the IIOP connector writes log output.
IIopLogLevel	error	Specifies the level of log information to write. This directive can take one of the following: debug|warn|info|error.
IIopClusterConfig	<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<apache_managedobject_name>/conf/WebClusters.properties	Specifies the location of the "cluster" instance file. For CORBA servers, identifies the file that contains the "cluster" name by which they are known to the IIOP connector[1].
IIopMapFile	<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<apache_managedobject_name>/conf/UriMapFile.properties	Specifies the location of the URI-to-Instance mapping file. For CORBA servers, maps HTTP URIs to a specific "cluster" known to the IIOP connector.

[1] "cluster" is used to represent a CORBA server instance that is known to the system by a single name or URI. The IIOP connector is able to load-balance across multiple instances, hence the term "cluster" is used.

The following are examples of typical configurations of these 5 lines for the IIOP connector for Apache 2.2:

Windows example

```
  LoadModule iiop2_module C:/Borland/BDP/lib/myapache/mod_iiop2.dll
  IIopLogFile C:/Borland/BDP/var/domains/base/configurations/j2ee/mos/
myapache/logs/mod_iiop.log
  IIopLogLevel error
  IIopClusterConfig C:/Borland/BDP/var/domains/base/configurations/j2ee/
mos/myapache/conf/WebClusters.properties
  IIopMapFile C:/Borland/BDP/var/domains/base/configurations/j2ee/mos/
myapache/conf/UriMapFile.properties
```

Solaris example

```
  LoadModule iiop2_module /opt/Borland/BDP/lib/myapache/mod_iiop2.so
  IIopLogFile /opt/Borland/BDP/var/domains/base/configurations/j2ee/mos/
myapache/logs/mod_iiop.log
  IIopLogLevel error
  IIopClusterConfig /opt/Borland/BDP/var/domains/base/configurations/j2ee/
mos/myapache/conf/WebClusters.properties
  IIopMapFile /opt/Borland/BDP/var/domains/base/configurations/j2ee/mos/
myapache/conf/UriMapFile.properties
```

Additional Apache IIOP directives

The following optional additional directives are available for you to use to further customize your Apache IIOP configuration.

Table 5.3 Additional Apache IIOP directives

Directive	default	Description
`IIopChunkedUploading`	(commented out) `true`	Controls whether Apache attempts "chunked" uploads to the Borland web container IIOP connector. To enable Apache to "chunk" large size data that is greater than the `IIopUploadBufferSize` value, uncomment and make sure it is set to `true`. **Note:** "Chunked" upload must also be enabled on the web container by setting the `server.xml` attribute `enablechunking="true"`. If you want Apache to wait until it has collected all data before invoking the CORBA RPC to send the large size data to the Borland web container, leave commented out or set to `false`. For more information, see "Implementing chunked upload" on page 50.
`IIopUploadBufferSize`	(commented out) `4096`	Defines the "chunked" buffer size employed when `IIopChunkedUploading` is set to `true`. This directive accepts a numeric value >0. Essentially, the larger the number of bytes you set this directive to, the less the number of CORBA RPCs that are required to send the data to the Borland web container. However, the larger you set this directive, the more memory will be consumed in servicing the transaction. Tuning this parameter allows you to fine-tune the performance characteristics. This enables the administrator to weigh the RPC costs against memory resource usage to optimize uploading on their system. **Note:** If an invalid value is presented (non numeric/negative number) then the default `4096` value is employed. For more information, see "Implementing chunked upload" on page 50.

Apache IIOP connector configuration

The Apache IIOP connector has a set of configuration files that you must update with web server cluster information. By default, these IIOP connector configuration files are located in:

```
<install_dir>/var/domains/<domain_name>/configurations/
<configuration_name>/mos/<apache_managedobject_name>/conf
```

The two configuration files are:

Table 5.4 Apache IIOP connection configuration files

IIOP configuration file	Description
WebClusters.properties	Specifies the cluster(s) and the corresponding web container(s) for each cluster.
UriMapFile.properties	Maps URI references to the clusters defined in the `WebClusters.properties` file.

Note Modifying either of these configuration files can be done so without starting up or shutting down the Apache web server(s) or CORBA server(s) because the file is automatically loaded by the IIOP connector.

Adding new clusters

The `WebClusters.properties` file tells the IIOP connector:

- The name of each available cluster (`ClusterList`).
- The web container identification.
- Whether to provide automatic load balancing (`enable_loadbalancing`) for a particular cluster.

To add a new cluster, in the `WebClusters.properties` file:

1 add the name of the configured cluster to the `ClusterList`. For example:

```
ClusterList=cluster1,cluster2,cluster3
```

2 define each cluster by adding a line in the following format specifying the cluster name, the required `webcontainer_id` attribute, and any additional attributes (see the following Cluster definition attributes table). For example:

```
<clustername>.webcontainer_id = <id> <attribute>
```

Note Failover and smart session are always enabled, for more information see "Clustering web components" on page 61.

Table 5.5 Cluster definition attributes

Attribute	Required	Definition
`webcontainer_id`	yes	the object "bind" name or corbaloc string identifying the web container implementing the cluster.
`enable_loadbalancing`	no	To enable load balancing, do not include this attribute **or** include and set to `true`; load balancing is enabled by default. To disable load balancing, set to `false` indicating that this cluster instance should **not** employ load-balancing techniques. **Warning:** Ensure that when entering the `enable_loadbalancing` attribute you give it a legal value (`true` or `false`).

For example:

```
   ClusterList=cluster1,cluster2,cluster3
cluster1.webcontainer_id = tc_inst1
cluster2.webcontainer_id = corbaloc::127.20.20.2:20202,:127.20.20.3:20202/
        tc_inst2
   cluster2.enable_loadbalancing = true
   cluster3.webcontainer_id = tc_inst3
   cluster3.enable_loadbalancing = false
```

In the above example, the following three clusters are defined:

1 The first, uses the osagent naming scheme and is enabled for load balancing.

2 The second cluster employs the corbaloc naming scheme, and is also enabled for load balancing.

3 The third uses the osagent naming scheme, but has the load balancing features disabled.

Note To disable use of a particular cluster, simply remove the cluster name from the `ClusterList` list. However, we recommend you do not remove clusters with active http sessions attached to the web server (attached users), because requests to these "live" sessions will fail.

Note Modifications you make to the `WebClusters.properties` file automatically take effect on the next request. You do not need to restart your server(s).

Adding new web applications

Important By default, your web application is not made available through Apache. In order to make it available through Apache, you must add some information to the web application descriptor. For step-by-step instructions on how to do so, see “Web Deploy Paths” in the *Management Console User's Guide*.

For new applications that you have deployed to the Borland web container, you need to do the following to make them available through the Apache web server. Use the `UriMapFile.properties` file to map `HTTP URI` strings to web cluster names configured in the `WebClusters.properties` file (see “Adding new clusters” on page 46).

- In the `UriMapFile.properties` file, type:

  ```
  <uri-mapping> = <clustername>
  ```

  where `<uri-mapping>` is a standard URI string or a wild-card string, and `<clustername>` is the cluster name as it appears in the `ClusterList` entry in the `WebClusters.properties` file.

For example:

```
/examples = cluster1
/examples/* = cluster1

/petstore/index.jsp = cluster2
/petstore/servlet/* = cluster2
```

In this example:

- Any URI that starts with `/examples` will be forwarded to a web container running in the “cluster1” web cluster.
- URIs matching either `/petstore/index.jsp` or starting with `/petstore/servlet` will be routed to “cluster2”.

Note With the URI mappings, the wild-card “*” is only valid in the **last** term of the URI and may represent the follow cases:

- the whole term (and all inferior references) as in `/examples/*`.
- the filename part of a file specification as in `/examples/*.jsp`.

Note Modifications you make to the `UriMapFile.properties` file automatically take effect on the next request. You do not need to restart your server(s).

If the `WebCluster.properties` or `UriMapFile.properties` is altered, then it is automatically loaded by the IIOP connector. This means that the adding and removing of web applications and the altering of cluster configurations may be done so without starting up or shutting down the Apache web server(s) or Borland web container(s).

Large data transfer

This section details the AppServer options available to you for handling large data transfers between a client and the Borland web container with Apache 2.2 in between. The data to be transferred may be either:

- static content obtained from a file, or
- dynamically generated content

Typically, the content length is known in advance for static content, but is not known for dynamic content.

Downloading large data

The following modes are available for downloading large data from the Borland web container to the browser:

- Chunked download
- Non-chunked download

Implementing chunked download

In the *chunked* download mode, the Borland web container does not wait until it has all the data to send. As soon as servlet generates the data, the web container starts sending the data to the browser via Apache in fixed size buffers.

Because the data is flushed as soon as it is available, the chunked download mode of transfer has low memory requirements both on Apache and the Borland web container. The browser user sees data as it arrives rather than as one large lump at the end of the full transfer.

Enabling chunked download

To enable chunked download mode, you update the Borland web container `server.xml` file which is stored in your Partition's data directory:

```
adm/tomcat/conf/
```

For more information, see "Borland web container implementation" on page 36.

To enable the chunked download:

1. In the Borland web container `server.xml`, locate the `<Service name="IIOP">` section of the code.
2. For the IIOP service you want to enable chunked download, set `enableChunking="true"`
3. By default, the download buffer size is set to `4096`. To change it for an IIOP service, use the `downloadBufferSize` attribute as follows:

   ```
   downloadBufferSize=<value>
   ```

   Where `<value>` is a numeric value >`0`.

Note If an invalid value is presented (non numeric/negative number) then the default `4096` value is employed.

The chunked download mode of transfer has an overhead of an extra thread per each request.

Known content length versus unknown

Based on whether content length is known in advance or not, chunked download mode can take one of the following two paths:

- chunked download with known content length
- chunked download with unknown content length

Chunked download with known content length

In this case, a servlet or JSP knows the content length of the data in advance of the transfer. The servlet sets the `Content-Length` HTTP header before writing out the data. The Borland web container writes out a single response header followed by multiple chunks of data. Apache receives this from the web container and sends data in the same fashion to the browser.

The response header contains the following header:

```
Content-Length=<actual data size>
```

Chunked download with unknown content length

HTTP protocol version 1.1 adds a new feature to handle the case of data transfer when data length is **not** known in advance. This feature is called *HTTP chunking*. In this case, a servlet does not know the content length of the data in advance of a transfer. The servlet does **not** set the `Content-Length` HTTP header.

The Borland web container sends the data to the Apache web server in exactly the same way as in the case of the chunked download where the content length is known in advance; a single response header is sent followed by multiple data chunks. The response header contains the following header:

```
Transfer-Encoding="chunked"
```

If the browser protocol is HTTP 1.1 and the `Content-Length` header is not set by the servlet, the Borland web container automatically adds the `Transfer-Encoding="chunked"` header.

When an Apache web server sees this `Transfer-Encoding` header, it starts sending the data as “HTTP chunks”—a response header followed by multiple combinations of “chunked” header, “chunked” data, and “chunked” trailers.

Note Per the HTTP 1.1 specification, if a servlet sets both the `Content-Length` and `Transfer-Encoding` headers, the `Content-Length` header is dropped by the Borland web container.

Browsers supporting only the HTTP 1.0 protocol

If the browser only supports the HTTP 1.0 protocol or less and a servlet does not set the `Content-Length` header, the Borland web container can not automatically add the `Transfer-Encoding` header. The reason being that to the HTTP 1.0 protocol, the `Transfer-Encoding` header has no meaning. In this case, the Borland web container:

1. buffers all the data until the data is finished,
2. calculates the content length, and
3. sets the `Content-Length` header itself.

If you do not want the Borland web container to perform this buffering behavior, you can set the IIOP connector attribute `canBufferHttp10Data="false"`. By default, this attribute is set to `true`.

Note When the `canBufferHttp10Data` attribute is set to `false`, the following error message is sent to the browser:

```
Servlet did not set the Content-Length
```

Implementing non-chunked download

This is the default transfer of data mode for the IIOP connector. In the *non-chunked* download mode, the Borland web container waits until it has all the data to send. Then it calculates the content length and sets the `Content-Length` header to the actual content length. The Borland web container then sends the response header followed by a single huge data block.

This mode of transfer has high memory requirements both on the Apache web server and the Borland web container, because the data is cached until all of it is available. Only when all the data is transferred does the browser user see the data.

The non-chunked download mode of transfer has no overhead of extra thread per each request. This download mode works well under both the HTTP protocol versions 1.0 and 1.1, because the `Transfer-Encoding` header is never set in this mode.

Uploading large data

The following modes are available for uploading large data initiated by a client (which can be either a browser or a non-browser (such as Java) client that speaks HTTP):

- Chunked upload
- Non-chunked upload

The browser always sends the data to an Apache web server in a “chunked” fashion. *Chunked* and *non-chunked* upload refers to the data transfer mode between an Apache web server and a Borland web container.

Implementing chunked upload

By default, Apache will try to upload large size data in “chunks”. In this mode, Apache does not wait until it has all the data from the browser before it starts sending data to a Borland web container. Apache sends the data in fixed size buffers as the data becomes available from the browser.

Because the data is flushed as soon as possible, the chunked mode of upload transfer has low memory requirements both on Apache and the Borland web container.

The chunked mode of transfer has an overhead of an extra thread per each request on the Borland web container.

Enabling chunked upload

To enable chunked upload mode, you must update **both** of the following:

- the Borland web container `server.xml` file, which is stored in your Partition's data directory: `adm/tomcat/conf`

  For more information see “Borland web container implementation” on page 36.

- the Apache `httpd.conf` file, which by default is located in the following directory:

  ```
  <install_dir>/var/domains/<domain_name>/configurations/
  <configuration_name>/mos/<apache_managedobject_name>/conf
  ```

  For more information see “Apache configuration” on page 33.

To enable the chunked upload:

1. In the Borland web container `server.xml`, locate the `<Service name="IIOP">` section of the code.
2. By default, the `enableChunking` attribute is set to `false`. Change this value to `enableChunking="true"`
3. In the Apache `httpd.conf` file, locate and uncomment the following IIOP directive:

   ```
   #IIopChunkedUploading true
   ```

Note The chunked upload mode of transfer has an overhead of an extra thread per each request for the Borland web container.

Changing the upload buffer size

By default, `IIopUploadBufferSize` is set to `4096` bytes. To change this value:

1. In the Apache `httpd.conf`, locate the following commented out directive:

   ```
   #IIopUploadBufferSize 4096
   ```

2. Uncomment this directive and set as follows:

   ```
   IIopUploadBufferSize <value>
   ```

   where <value> is a numeric value >`0` (greater than zero).

Note If you specify an invalid value (non numeric/negative number) then the default `4096` value is employed.

Known content length versus unknown

Based on whether content length is known in advance or not, chunked upload mode can take one of the following two paths:

- chunked upload with known content length
- chunked upload with unknown content length

Chunked upload with known content length

In this case, the client knows the content length of the data in advance of the transfer. The client sets the `Content-Length` HTTP header before writing out the data. The client writes out a single response header followed by multiple chunks of data. Apache receives this from the browser and sends data in the same fashion to the Borland web container.

The response header contains the following header:

```
Content-Length=<actual data size>
```

Chunked upload with unknown content length

HTTP protocol version 1.1 adds a new feature to handle the case of data transfer when data length is **not** known in advance. This feature is called *HTTP chunking*.

In this case, a client does not know the content length of the data in advance of a transfer. The client does **not** set the `Content-Length` HTTP request header. Instead, the client sets the `Transfer-Encoding` HTTP request header to a value of `chunked` as follows:

```
Transfer-Encoding="chunked"
```

The client sends the data to the Apache web server as “HTTP chunks”; a single request header followed by multiple combinations of *chunk header*, *chunk data*, and *chunk trailer*.

When the Apache web server sees this `Transfer-Encoding` header, it strips out the chunk header and chunk trailers and sends the data as normal data chunks to the Borland web container.

At this time, no major browsers support uploading data without knowing the content length. In other words, browsers never add a `Transfer-Encoding="chunked″` header to an HTTP request. However, a non-browser client can add this header to an HTTP request.

Implementing non-chunked upload

This is the default transfer of data mode for the IIOP connector. In the *non-chunked* upload mode, the Apache web server waits until it has all the data to send. Then it calculates the content length and sets the `Content-Length` header to the actual content length. Apache then sends the request header followed by a single huge data block.

This mode of transfer has high memory requirements both on the Apache web server and the Borland web container, because the data is cached until all of it is available.

The non-chunked upload mode of transfer has no overhead of extra thread per each request (in the Borland web container). This download mode works well under both the HTTP protocol versions 1.0 and 1.1, because the `Transfer-Encoding` header is never set in this mode.

IIS web server to Borland web container connectivity

This section describes the Tomcat-based Borland web container, its IIOP connector, as well as the IIS/IIOP redirector which provides connectivity from the Microsoft Internet Information Services (IIS) web server (not included with AppServer) to the Borland web container. These features are provided in AppServer. For more information, see “Installing Borland AppServer on Windows” or “Installing Borland AppServer on Solaris and HP-UX” in the *Installation Guide*.

Modifying the IIOP configuration in the Borland web container

The `server.xml` is the main configuration file for the Borland web container and is stored in your Partition's data directory:

```
adm/tomcat/conf/
```

Within the `server.xml` file is a section that pertains to the IIOP connector configuration. For detailed configuration information, see “Modifying the Borland web container IIOP configuration” on page 41.

Microsoft Internet Information Services (IIS) server-specific IIOP configuration

Before the IIS/IIOP redirector can be used on your system, you need to complete the following:

- Configure your Windows 2003/XP/2000 system on which IIS is running
- Configure the IIS/IIOP redirector

How to Configure your Windows 2003/XP/2000 system on which IIS is running

1 Add the `OSAGENT_PORT` required environment variable to the SYSTEM environment.

The IISredirectory relies on VisiBroker to provide the IIOP communication layer between IIS and the Borland web container. In order to function, VisiBroker requires that the following environment variable be defined as follows:

Environment Variable	Value	Description
`OSAGENT_PORT`	`14000`	The numeric value of the OSAGENT port number used by your AppServer.

Important After setting the `OSAGENT_PORT` environment variable, in order for IIS to detect it, you must reboot the Windows system.

2 Add the IIS/IIOP redirector as an ISAPI filter.

a Right-click My Computer and choose Manage.

The Computer Management dialog appears.

b Expand the tree, expand the Services and Applications node.

c Expand the Internet Information Services node.

d Right-click the Default Web Site node and choose Properties.

The Default Web Site Properties dialog appears.

e Go to the ISAPI Filters tab.

f Click Add.

g In the Filter Properties dialog, type a Filter Name and the path for the Executable in the corresponding entry boxes.

By convention, the name of the filter should reflect its task, for example:

```
iisredir2
```

Also, the executable should point to the `iisredir2.dll` in the `<install_dir>\bin`. For example:

```
C:\borland\BDP\bin\iisredir2.dll
```

h Click OK.

Your new ISAPI filter appears on the list. You do not need to change the filter Priority.

i Click OK.

3 Add a “borland” virtual directory to your IIS web site.

a In the Computer Management dialog, right-click Default Web Site and choose New|Virtual Directory.

The Virtual Directory Creation Wizard appears.

b Click Next.

c For the Alias, enter “borland”.

The `borland` virtual directory is required to allow the IIS/IIOP redirector extension to be located by the IIS web server when it responds to a URI of:

```
http://localhost/borland/iisredir2.dll
```

d For the Directory, browse to `<install_dir>\bin`.

e Click Next to proceed.

f For Access Permissions, select “Execute” in addition to “Read” and “Run scripts” which are selected by default.

g Click Next.

h Click Finish.

4 **Windows 2003 only:** Configure ISAPI extension permissions for Windows 2003.

The IIS version limits which application extensions may be loaded into IIS. You have the choice of enabling all extensions or selectively picking which ISAPI extensions that may be run in your IIS installation. The following procedure enables just the `iisredir2` extension.

a In the Computer Management dialog, open “Services and Applications”.

b Open “Internet Information Service”.

c Open “Web Service Extensions” and click Add a new Service Extension.

d Name the extension “iisredir2.dll”.

e Browse (using the add button) to <install>\bin\iisredir2.dll

f Select this file.

g Check the Extension allowed checkbox.

h Click OK.

5 Restart IIS by stopping **then** starting the IIS Service:

a In the Computer Management dialog, right-click the Internet Information Services node and choose Restart IIS.

b In the Stop/Start/Reboot dialog, from the drop-down choose “Stop Internet Services on <name of your IIS web server>"

c Click OK.

The web service unloads any dlls loaded by the IIS administrator.

d After shut down of the server is complete, right-click the Internet Information Services node and choose Restart IIS.

e In the Stop/Start/Reboot dialog, choose “Start Internet Services on <your IIS web server name>”.

f Click OK.

The web service reloads any dlls loaded by the IIS administrator.

6 Make sure the `iisredir2` filter is active.

a In the Computer Management dialog, right-click the Default Web Site node and choose Properties.

b In the Default Web Site Properties dialog, go to the ISAPI Filters tab.

c The `iisredir2` filter should be marked with a green up-pointing arrow indicating that it has been activated.

If not, then check the `iisredir2.log` file for details of why it may not have loaded correctly. This file can be found in:

```
<install_dir>\etc\iisredir2\logs
```

d To exit, click OK.

7 Attempt to access the `\examples` context via the IIS web-server.

If you have followed the previous steps, the `\examples` context should be accessible following a restart of your IIS Server.

Note In the example the port number of the web server should match that configured for your site. For instance, if your IIS administrator has configured IIS to listen on port `6060`, then a valid URL is:

```
http://localhost:6060/examples
```

Of course, if your IIS is configured as per Microsoft defaults, then it listens on port `80`, in which case you may dispense with a port number. For example:

```
http://localhost/examples
```

IIS/IIOP redirector configuration

The IIS/IIOP redirector has a set of configuration files that you must update with web server cluster information. By default, these IIOP redirector configuration files are located in the following directory:

```
<install_dir>/etc/iisredir2/conf
```

The configuration files are:

Table 5.6 IIS/IIOP redirector configuration files

IIOP configuration file	Description
WebClusters.properties	Specifies the cluster(s) and the corresponding web container(s) for each cluster.
UriMapFile.properties	Maps URI references to the clusters defined in the `WebClusters.properties` file.

Note Modifying either of these configuration files can be done so without starting up or shutting down the IIS web server(s) or Borland web container(s) because the file is automatically loaded by the IIOP redirector.

Adding new clusters

The `WebClusters.properties` file tells the IIOP redirector:

- the name of each available cluster: (`ClusterList`).
- the web container identification.
- whether to provide automatic load balancing (`enable_loadbalancing`) for a particular cluster.

To add a new cluster, in the `WebClusters.properties` file:

1 add the name of the configured cluster to the `ClusterList`. For example:

```
ClusterList=cluster1,cluster2,cluster3
```

2 define each cluster by adding a line in the following format specifying the cluster name, the required `webcontainer_id` attribute, and any additional attributes (see the following Cluster definition attributes table). For example:

```
<clustername>.webcontainer_id = <id> <attribute>
```

Note Failover and smart session are always enabled, for more information see "Clustering web components" on page 61.

Table 5.7 Cluster definition attributes

Attribute	Required	Definition
`webcontainer_id`	yes	the object "bind" name or corbaloc string identifying the web container(s) implementing the cluster.
`enable_loadbalancing = true|false`	no	To enable load balancing, do not include this attribute **or** include and set to `true`; load balancing is enabled by default. To disable load balancing, set to `false` indicating that this cluster instance should **not** employ load-balancing techniques. **Warning:** Ensure that when entering the `enable_loadbalancing` attribute you give it a legal value (`true` or `false`).

For example:

```
   ClusterList=cluster1,cluster2,cluster3
cluster1.webcontainer_id = tc_inst1
cluster2.webcontainer_id = corbaloc::127.20.20.2:20202,:127.20.20.3:20202/
     tc_inst2
   cluster2.enable_loadbalancing = true
   cluster3.webcontainer_id = tc_inst3
   cluster3.enable_loadbalancing = false
```

In the above example, the following three clusters are defined:

1 The first, uses the osagent naming scheme and is enabled for load balancing.

2 The second cluster employs the corbaloc naming scheme, and is also enabled for load balancing.

3 The third uses the osagent naming scheme, but has the load balancing features disabled.

Note To disable use of a particular cluster, simply remove the cluster name from the `ClusterList` list. However, we recommend you do not remove clusters with active http sessions attached to the web server (attached users), because requests to these "live" sessions will fail.

Note Modifications you make to the `WebClusters.properties` file automatically take effect on the next request. You do not need to restart your server(s).

Adding new web applications

Important By default, your web applications are not made available through IIS. In order to make a web application available through IIS, you must add some information to the web application descriptor. For step-by-step instructions on how to do so, see "Web Deploy Paths" in the *Management Console User's Guide*.

The `\examples` context is useful for verifying your IIS/IIOP installation configuration, however, for new applications that you have deployed to the Borland web container, you need to do the following to make them available through the IIS web server. Use the `UriMapFile.properties` file to map `HTTP URI` strings to web cluster names configured in the `WebClusters.properties` file (see "Adding new clusters" on page 55).

- In the `UriMapFile.properties` file, type:

  ```
  <uri-mapping> = <clustername>
  ```

  where `<uri-mapping>` is a standard URI string or a wild-card string, and `<clustername>` is the cluster name as it appears in the `ClusterList` entry in the `WebClusters.properties` file.

For example:

```
/examples = cluster1
/examples/* = cluster1

/petstore/index.jsp = cluster2
/petstore/servlet/* = cluster2
```

In this example:

- Any URI that starts with `/examples` will be forwarded to a web container running in the "cluster1" web cluster.
- URIs matching either `/petstore/index.jsp` or starting with `/petstore/servlet` will be routed to "cluster2".

Note With the URI mappings, the wild-card "*" is only valid in the last term of the URI and may represent the follow cases:

- the whole term (and all inferior references) as in `/examples/*`.
- the filename part of a file specification as in `/examples/*.jsp`.

Note Modifications you make to the `UriMapFile.properties` file automatically take effect on the next request. You do not need to restart your server(s).

If the `WebCluster.properties` or `UriMapFile.properties` is altered, then it is automatically loaded by the IIOP redirector. This means that the adding and removing of web applications and the altering of cluster configurations may be done so without starting up or shutting down the IIS web server(s) or Borland web container(s).

Chapter

6

Java Session Service (JSS) configuration

The Java Session Service (JSS) is a service that stores information pertaining to a specific user session. JSS is used to store session information for recovery in case of container failure.

Borland provides an Interface Definition Language (IDL) interface for the use of JSS. Two implementations are bundled, one using DataExpress and another with any JDBC capable database.

JSS provides a mechanism to easily store session information in a database. For example, in a shopping cart scenario, information about your session (the number of items in the shopping cart, and such) is stored by the JSS. So, if a session is interrupted by a Borland web container unexpectedly going down, the session information is recoverable by another Borland web container instance through the JSS. The JSS must be running on the local network. Any web container (within the cluster configuration) finds the JSS and connects to it and continues session management.

For more information about the Borland web container, see “Borland web container implementation” on page 36 .

Session management with JSS

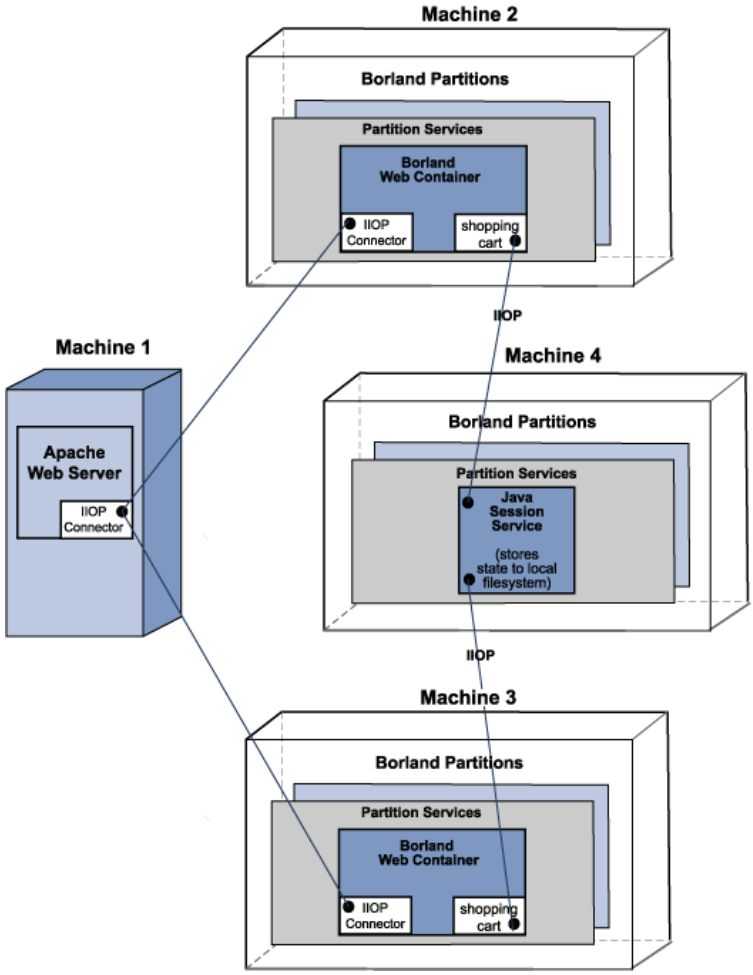
The following diagrams show typical landscapes of web components and how session information is managed by the JSS. The JSS session management is completely transparent to the client.

In the JSS Management with a Centralized JSS and Two Web Containers diagram, there are four virtual machines:

- The first machine hosts the Apache web server,
- two other machines contain an instance of the Borland web container,
- and the fourth machine hosts the JSS and relational database (JDataStore or a JDBC datasource).

If an interruption occurs between the Apache web server (Machine 1) which is passing a client request to the first web container instance (Machine 2), then the second web container instance (Machine 3) can continue processing the client request by retrieving the session information from the JSS (Machine 4). The items in the Shopping Cart are retained and the client request continues to be processed.

Figure 6.1 JSS Management with a Centralized JSS and Two Web Containers

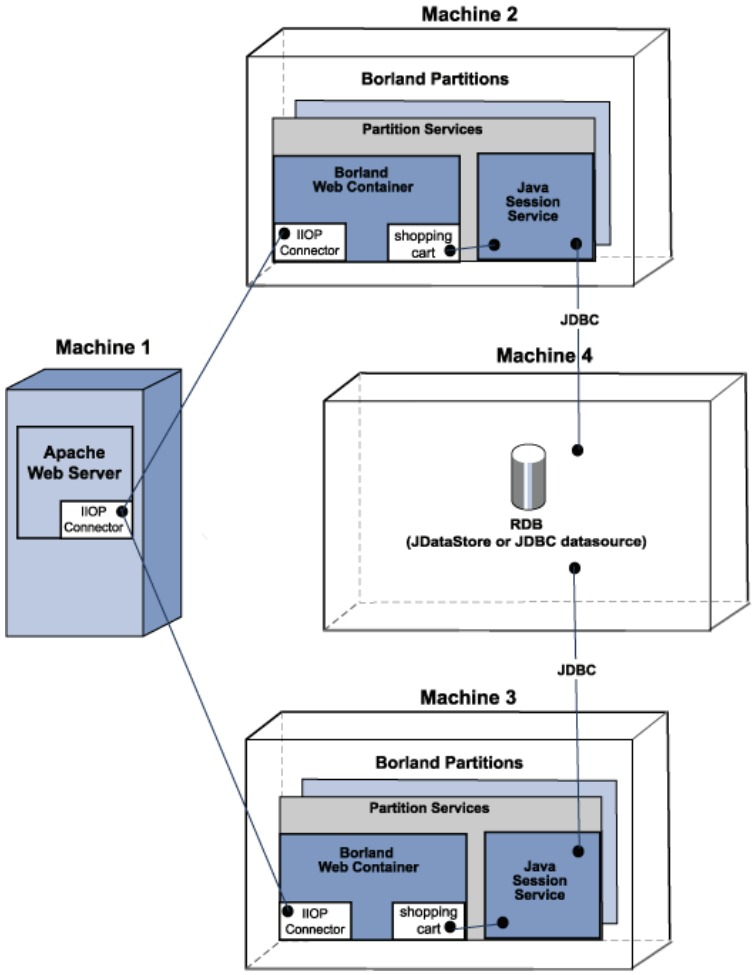


In the JSS Management with Two Web Containers and a Centralized Backend Datastore diagram, are the following four virtual machines:

- The first machine hosts the Apache web server,
- the two other machines contain an instance of the Borland web container as well as each hosting the JSS,
- and the fourth machine hosts the relational database (JDataStore or a JDBC datasource).

If an interruption occurs between the Apache web server (Machine 1) which is passing a client request to the first web container instance (Machine 2), then the second web container instance (Machine 3) can continue processing the client request by retrieving the session information from the JSS (Machine 4). The items in the Shopping Cart are retained and the client request continues to be processed.

Figure 6.2 JSS Management with Two Web Containers and a Centralized Backend Datastore



Managing and configuring the JSS

The JSS configuration is defined through its properties. The Borland AppServer (AppServer) is designed to work with any J2EE certified JDBC 2 driver; however, AppServer is only certified with and supported on JDataStore and Oracle.

- If JSS is configured to use a JDataStore file, the database tables are automatically created by JSS.
- If JSS is configured to use a JDBC datasource, three database tables needs to be pre-created in the backend database by your system administrator using the following SQL statements:

```
CREATE TABLE "JSS_KEYS" ("STORAGE_NAME" java_string primary key, "KEY_BASE"
java_float);
CREATE TABLE "JSS_WEB" ("KEY" java_string primary key, "VALUE"
java_serializable, "EXPIRATION" java_float);
CREATE TABLE "JSS_EJB" ("KEY" java_string primary key, "VALUE"
java_serializable, "EXPIRATION" java_float);
```

Note When using the above SQL statements, make sure to substitute the equivalent data types for your database.

The JSS can run as part of the Partition side-by-side with other Partition services.

Configuring the JSS Partition service

As a “Partition service”, JSS configuration information is located in each Partition's data directory in the `partition.xml` file. By default, this file is located in the following directory:

```
<install_dir>/var/domains/base/configurations/<configuration_name>/mos/
<partition_name>/adm/properties.
```

For example, for a Partition named “MyPartition”, by default the JSS configuration information is located in:

```
<install_dir>/var/domains/base/configurations/<configuration_name>/mos/
mypartition/adm/properties/partition.xml
```

For more information see “<service> element” on page 344.

Otherwise, for the location of a Partition data directory, go to the `configuration.xml` file located in:

```
<install_dir>/var/domains/base/configurations/<configuration_name>/
```

and search for the Partition Managed Object directory attribute:

```
<partition-process directory=
```

For a listing and description of the session service (JSS) level properties, see “Java Session Service (JSS) Properties” on page 358.

Chapter

7

Clustering web components

This section discusses the clustering of multiple web components which includes Apache web servers and the Tomcat-based Borland web containers. In a typical deployment scenario, you can use multiple Borland Partitions to work together in providing a scalable *n*-tier solution.

Each Borland Partition can have the same or different services. Depending on your clustering scheme, these services can be turned off or on. In any case, leveraging these resources together or clustering, makes deployment of your web application more efficient. Clustering of the web components involves session management, load balancing and fault tolerance (failover).

Stateless and stateful connection services

Interaction between the client and server involves two types of services: stateless and stateful. A *stateless service* does not maintain a state between the client and the server. There is no “conversation” between the server and the client while processing a client request. In a *stateful service*, the client and server maintains a dialog of information.

For information about the location of the Borland web container configuration files, see “Borland web container implementation” on page 36.

The Borland IIOP connector

The IIOP connector is software that is designed to allow an http web server to redirect requests to the Borland web container. The Borland AppServer (AppServer) includes the IIOP connector for the Apache 2.2 and Microsoft Internet Information Server(IIS) versions 5.0, 5.1 and 6.0 web servers. The job of handling the redirection of http requests is split between two components:

- a native library running on the web server.
- a jar file running of the web container.

The AppServer supports clustering of web components. The Borland IIOP connector uses the IIOP protocol. The following unique features are provided:

- Load Balancing
- Fault tolerance (failover)
- Smart Session handling

Load balancing support

Load balancing is the ability to direct http requests across a set of web containers. This enables the system administrator to spread the load of the http traffic across multiple web containers. Load balancing techniques can significantly improve the scalability of a given system. The Borland IIOP connector can be configured to offer load balancing in the following two ways:

- OSAgent based load balancing
- Corbaloc based load balancing

OSAgent based load balancing

This is simple to achieve and requires the least amount of configuration. In this setup, you start a number of Borland web container instances and name the IIOP connector in those Borland web container with the same name.

For more information about setting the `name` attribute, see "Modifying the Borland web container IIOP configuration" on page 41.

Apache does load balancing across Borland web container instances for each request. Essentially, Apache does a new bind for each request. The newly started Borland web container containers can be dynamically discovered.

Important All Borland web containers and Apache must be running in the same ORB domain; osagent based load balancing is not possible in cases where you are using different Partitions on different ORB domains.

Corbaloc based load balancing

This approach uses a static configuration of the web containers that make up the cluster. However, it can span ORB domains. In this case you specify the locations where the web containers are running using the CORBA corbaloc semantics. For example:

```
corbaloc::172.20.20.28:30303,:172.20.20.29:30304/tc_inst1
```

In the above corbaloc example string:

- two TCP/IP endpoints are configured for a web container named "`tc_inst1`"
- a web container with an object name of "`tc_inst1`" is running on host `172.20.20.28` with its IIOP connector at port `30303`
- there is another web container running with the same object name on host `172.20.20.29` with it's IIOP connector listening on port `30304`.

For more information about setting the `port` attribute, see "Modifying the Borland web container IIOP configuration" on page 41.

The web server side IIOP connector converts this corbaloc string into CORBA objects using `orb.string_to_object` and uses the underlying features of VisiBroker to load balance across these "endpoints" specified in the corbaloc string. There can be any number of endpoints.

Note All of the listed web containers do not have to be running for the load balancing to function. The ORB simply moves on to the next endpoint until a valid connection is obtained.

However, corbaloc based load balancing does require the web container's IIOP connector be started at a known port and be available for corbaloc kind of object naming. The following is a snippet of the web container IIOP connector configuration that is required:

```
<Connector className="org.apache.catalina.connector.iiop.IiopConnector"
name="tc_inst1"  port="30303"/>
```

This snippet starts the IIOP connector at port `30303` and names the Borland web container object "`tc_inst1`". The `port` attribute is optional. However, if you do not specify the port, a random port gets picked up by the ORB and you will be unable to use the corbaloc scheme to locate the object.

Your organization can impose policies on how to name web containers and the IIOP port or port ranges used.

Fault tolerance (failover)

Failover using osagent bind naming and corbaloc naming is automatic in both cases. In corbaloc naming, the next configured endpoint in the corbaloc name string is tried and so on in a cyclic fashion until all endpoints in the corbaloc string are tried.

For osagent bind naming, the osagent automatically redirects the client to an alternative (but equivalent) object instance.

Note If there is no object available to the osagent, or none of the endpoints specified in the corbaloc name string are running, then the http request fails.

Smart session handling

When there is no session involved, the IIOP connector can do indiscriminate round robining. However, when sessions are involved, it is important that Apache routes its session requests to the web container that initiated the session.

In other http-to-servlet redirectors (and in the earlier version of the IIOP connector) this is achieved by maintaining a list of sessions-ids-to-web-container-id's in the web server's cache. This presents numerous issues with maintaining the state of this list. This list can be very large and wasteful of system resources. It can become out of date, for example, sessions can timeout and, in general, is an inefficient and problematic facet of the web server to web container redirection paradigm.

The IIOP connector resolves this by utilizing a technique called "smart session ids". This is where the IOR of the web container is embedded within the session id returned by the web container as part of the session cookie (or URL in the case of url-rewriting).

When the web container generates the session ID, it first determines if the request originated from the IIOP connector. If so, it obtains the stringified IOR of the IIOP connector through which the request is received. The web container generates the normal session ID as it always generates, but pre-fixes the stringified IOR in front of it. For example:

```
Stringified IOR: IOR:xyz
Normal session ID: abc
The new session ID: xyz_abc
```

In the case where the original web container has stopped running, failover is employed to locate another instance of an equivalent web container.

In the case of corbaloc identified web containers, where automatic osagent failover is not guaranteed, the IIOP connector performs a manual "rebind" to obtain a valid reference to the running equivalent web container.

Obviously, if there are no other running instances of the web container, then the http request fails.

The new web container obtains the old state from the session database and continues to service the request. When returning the response the new web container changes the session ID to reflect its IOR. This should be transparent to Apache as it does not look at the session ID on the way back to the browser client.

Setting up your web container with JSS

To properly failover when sessions are involved, you must set up the web containers with the same JSS backend.

Modifying a Borland web container for failover

In the Borland web container configuration file, `server.xml`, you need to add an entry similar to the following code sample for each web application. For more information about the `server.xml` file, see "Modifying the Borland web container IIOP configuration" on page 41.

```
<Manager className="org.apache.catalina.session.PersistentManager">
           <Store className="org.apache.catalina.session.BorlandStore"
           storeName="jss_factory"/>
     </Manager>
```

The preceding code specifies the use of a `PersistentManager` with a storage class `BorlandStore`. It also specifies the connection to a `BorlandStore` factory named `jss_factory`. There must be a JSS with that factory name running in the local network.

For a description of `jss.factoryName`, see "Java Session Service (JSS) Properties" on page 358.

Session storage implementation

There are two methods of implementing session storage for your clustered web components:

- Programmatic implementation
- Automatic implementation

Programmatic implementation

The *Programmatic implementation* assumes that each time you change the session attributes, you call `session.SetAttribute()` to notify the Borland web container that you have changed the session attributes.

This is a common operation in servlet development and when executed, there is no need to modify the `server.xml` file. Each time you change the session data, it is immediately written to the database through the JSS. Then if your web container instance unexpectedly goes down, the next web container instance designated to pick up the session accesses the session data. In essence, the Programmatic implementation guarantees to save changes immediately.

Automatic implementation

The *Automatic implementation* lets you store the session data periodically to JSS, regardless of whether the data has changed. By using this implementation, you do not need to notify the web container that the session attribute has changed.

For example, you can change state without calling `setAttribute ()` as depicted in the following code example:

```
Object myState = session.getAttribute("myState");

// Modify mystate here and do not call setAttribute ()
```

Your configuration file, `server.xml`, will have the following code snippet:

```
<Manager className=
"org.apache.catalina.session.PersistentManager"
         maxIdleBackup="xxx">
<Store className=
"org.apache.catalina.session.BorlandStore"
storeName="jss_factory">
</Manager>
```

where `xxx` is the time interval in seconds that you want the session data to be stored.

For more information about the `server.xml` file, see "Modifying the Borland web container IIOP configuration" on page 41.

Note When using the Automatic implementation, you need to consider the following limitations:

1. If the web container goes down between two save intervals, the latest changes are not visible for the next web container instance. This is an important concern when defining the time interval value for the heartbeat.
2. The data is saved at the specified time interval no matter if the data is changed or not. This can be wasteful if a session frequently does not change and the defined time interval value is set too low.

Using HTTP sessions

The HyperText Transfer Protocol (HTTP) is a stateless protocol. In the client/server paradigm, it means that all client requests that the Apache web server receives are viewed as independent transactions. There is no relationship between each client request. This is a typical stateless connection between the client and the server.

However, there are times when the client deems it necessary to have a session concept for transaction completeness. A *session concept* typically means having a stateful interaction between the client and server. An example of the session concept is shopping online with an interactive shopping cart. Every time you add a new item into your shopping cart, you expect to see that new item added to a list of previously added items. HTTP is not usually regarded for handling client request in a stateful manner. But it can.

AppServer supports the HTTP sessions through two methods of implementations:

- Cookies: The web server send a cookie to identify a session. The web browser keeps sending back the same cookie with future requests. This cookie helps the server-side components to determine how to handle the transaction for a given session.
- URL rewriting: The URL that the user clicks on is dynamically rewritten to have session information.

Chapter 8

Apache web server to CORBA server connectivity

The Apache IIOP connector can be configured to enable your web server to communicate with any standalone CORBA server implementing the `ReqProcessor` Interface Definition Language (IDL). This means you can easily put a web-based front-end on almost any CORBA server.

For more information, see “Installing Borland AppServer on Windows” or “Installing Borland AppServer on Solaris and HP-UX” in the *Installation Guide*.

Web-enabling your CORBA server

The following steps are required to make your CORBA server accessible over the internet:

- Determine the urls for your CORBA methods
- Implement the ReqProcessor IDL

Determining the urls for your CORBA methods

In order to make your CORBA server accessible over the internet, you need to:

1. Decide what business operations you want to expose.
2. Provide a url for those business operations (CORBA methods).

For example, your banking company's CORBA server is implementing the methods: `debit()`, `credit()`, and `balance()` and you want to expose these business methods to users through the internet. You need to map each of the CORBA server operations to what the user types in a browser.

Your bank company web site is `http://www.bank.com`.

To provide a url for each of the business operations you want to expose to the internet users:

1 Append the web application name to the company root url.

For example:

```
http://www.bank.com/accounts
```

where `accounts` is the web application name.

Important By default, your web application is not made available through the web server. In order to make it available through Apache, you must add some information to the web application descriptor. For step-by-step instructions on how to do so, see "Web Deploy Paths" in the *Management Console User's Guide*.

2 Append a name that is meaningful to users for the method in the web application that you want to expose.

For example:

```
http://www.bank.com/accounts/balance
```

where `balance` is the meaningful name for the `balance()` method.

Implementing the ReqProcessor IDL in your CORBA server

The `ReqProcessor` IDL allows communication between a web server and a CORBA server using IIOP. Once you implement the `ReqProcessor` IDL in your CORBA server, http requests can be passed from your web server to your CORBA server.

In implementing this IDL, you must expect the request url as part of the `HttpRequest` and invoke the appropriate CORBA method in response to that url.

IDL Specification for ReqProcessor Interface

```
*/
module apache {
        struct NameValue {
                string name;
                string value;
    };
        typedef sequence<NameValue> NVList;
    typedef sequence<octet> OctetSequence_t;

    struct HttpRequest {
                string  authType;     // auth type (BASIC,FORM etc)
                string  userid;       // username associated with request
                string  appName;      // application name (context path)
                string  httpMethod;   // PUT, GET etc,
                string  httpProtocol; // protocol HTTP/1.0, HTTP/1.1 etc
                string  uri;          // URI associated with request
                string  args;      // query string associated with this request
                string  postData;  // POST (form) data associated with request
                boolean isSecure;  // whether client specified https or http
                string  serverHostname; // server hostname specified with URI
  string  serverAddr; // [optionally] server IP address specified with URI
                long    serverPort; // server port number specified with URI
               NVList headers; // headers associated with this request format:
                   header-name:value
        };
```

```
        struct HttpResponse {
                long    status;                 // HTTP status, OK etc.
                boolean isCommit; // server intends to commit this request
                NVList  headers;                // header array
                OctetSequence_t data;   // data buffer
        };

interface ReqProcessor {
        HttpResponse process(in HttpRequest req);
  };
};
```

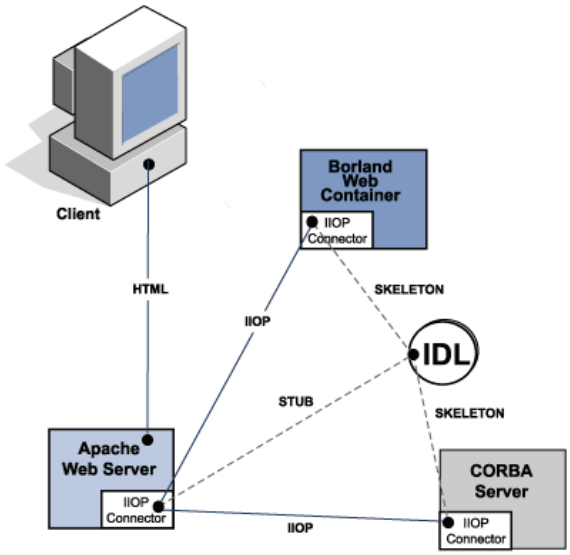
The process() method

The `ReqProcessor` IDL includes the `process()` method which your Apache web server calls for internet requests. The web server passes the user's request as an argument to the `process()` method. Basically, the input for the `process()` method is a request from a browser: `HttpRequest`, and the output for the `process()` method is an html page contained in: `HttpResponse`.

Configuring your Apache web server to invoke a CORBA server

Before an Apache web server can invoke a CORBA server, you must modify the lines of code that pertain to the IIOP connector in the `httpd.conf` file. For detailed information, see “Modifying the IIOP configuration in Apache” on page 43.

Figure 8.1 Connecting from Apache to a CORBA server



Apache IIOP configuration

The Apache IIOP connector has a set of configuration files that you must update with web server cluster information. By default, these IIOP connector configuration files are located in:

```
<install_dir>/var/domains/<domain_name>/configurations/
<configuration_name>/mos/<apache_managedobject_name>/conf
```

Note “cluster” is used to represent a CORBA object instance(s) that is known to the system by a single name or URI. The IIOP connector is able to load-balance across multiple instances, hence the term “cluster” is used.

The two configuration files are:

Table 8.1 Apache IIOP connection configuration files

IIOP configuration file	Description
WebClusters.properties	Specifies the cluster(s) and the corresponding CORBA server(s) for each cluster.
UriMapFile.properties	Maps URI references to the clusters defined in the `WebClusters.properties` file.

Modifying either of these configuration files can be done so without starting up or shutting down the Apache web server(s) or CORBA server(s) because the file is automatically loaded by the IIOP connector.

Adding new CORBA servers (clusters)

CORBA servers are known as “clusters” to the IIOP connector. To configure your CORBA server for use with the IIOP connector, you need to define and add a “cluster” to the `WebClusters.properties` file.

The `WebClusters.properties` file tells the IIOP connector:

- The name of each available cluster (`ClusterList`).
- The web container identification.
- Whether to provide automatic load balancing (`enable_loadbalancing`) for a particular cluster.

To add a new cluster:

- In the `WebClusters.properties` file:
  - **a** add the name of the configured cluster to the `ClusterList`. For example:

    ```
    ClusterList=cluster1,cluster2,cluster3
    ```

  - **b** define each cluster by adding a line in the following format specifying the cluster name, the required `webcontainer_id` attribute, and any additional attributes (see the following Cluster definition attributes table). For example:

    ```
    <clustername>.webcontainer_id = <id> <attribute>
    ```

Note Failover and smart session are always enabled, see “Clustering web components” on page 61.

Table 8.2 Cluster definition attributes

Attribute	Required	Definition
`webcontainer_id`	yes	the object “bind” name or corbaloc string identifying the web container implementing the cluster.
`enable_loadbalancing`	no	Load balancing is enabled by default; to enable load balancing, do not include this attribute **or** include and set to `true`. To disable load balancing, set to `false` indicating that this cluster instance should **not** employ load-balancing techniques. **Warning:** Ensure that when entering the `enable_loadbalancing` attribute you give it a legal value (`true` or `false`).

For example:

```
ClusterList=cluster1,cluster2,cluster3
cluster1.webcontainer_id = tc_inst1
cluster2.webcontainer_id = corbaloc::127.20.20.2:20202,:127.20.20.3:20202/
tc_inst2
   cluster2.enable_loadbalancing = true
   cluster3.webcontainer_id = tc_inst3
   cluster3.enable_loadbalancing = false
```

In the above example, the following three clusters are defined:

1. The first, uses the osagent naming scheme and is enabled for load balancing.
2. The second cluster employs the corbaloc naming scheme, and is also enabled for load balancing.
3. The third uses the osagent naming scheme, but has the load balancing features disabled.

Note To disable use of a particular cluster, simply remove the cluster name from the `ClusterList` list. However, we recommend you do not remove clusters with active http sessions attached to the CORBA server (attached users), because requests to these "live" sessions will fail.

Note Modifications you make to the `WebClusters.properties` file automatically take effect on the next request. You do not need to restart your server(s).

Mapping URIs to defined clusters

Once the cluster entry is defined, all that remains is to identify which HTTP requests received by the web server need to be forwarded to your CORBA server. Use the `UriMapFile.properties` file to map http uri strings to web cluster names (CORBA instances) configured in the `WebClusters.properties` file.

- In the `UriMapFile.properties` file, type:

  ```
  <uri-mapping> = <clustername>
  ```

  where `<uri-mapping>` is a standard URI string or a wild-card string, and `<clustername>` is the cluster name as it appears in the `ClusterList` entry in the `WebClusters.properties` file.

For example:

```
/examples = cluster1
/examples/* = cluster1

/petstore/index.jsp = cluster2
/petstore/servlet/* = cluster2
```

In this example:

- Any URI that starts with `/examples` will be forwarded to a CORBA server running in the "cluster1" web cluster.
- URIs matching either `/petstore/index.jsp` or starting with `/petstore/servlet` will be routed to "cluster2".

Note With the URI mappings, the wild-card "*" is only valid in the **last** term of the URI and may represent the follow cases:

- the whole term (and all inferior references) as in `/examples/*`.
- the filename part of a file specification as in `/examples/*.jsp`.

Note Modifications you make to the `UriMapFile.properties` file automatically take effect on the next request. You do not need to restart your server(s).

If the `WebCluster.properties` or `UriMapFile.properties` is altered, then it is automatically loaded by the IIOP connector. This means that modifications to either of these files can be done so without starting up or shutting down the web server(s) or CORBA server(s).

Chapter

9

Borland AppServer Web Services

The Borland AppServer (AppServer) provides an out-of-the-box web services capability in all Borland Partitions.

Web Services Overview

A *Web Service* is an application component that you can describe, publish, locate, and invoke over a network using standardized XML messaging. Defined by new technologies like Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Universal Discovery, Description and Integration (UDDI), this is a new model for creating e-business applications from reusable software modules that are accessed on the World Wide Web.

Web Services Architecture

The standard Web Service architecture consists of the three roles that perform the web services publish, find, and bind operations:

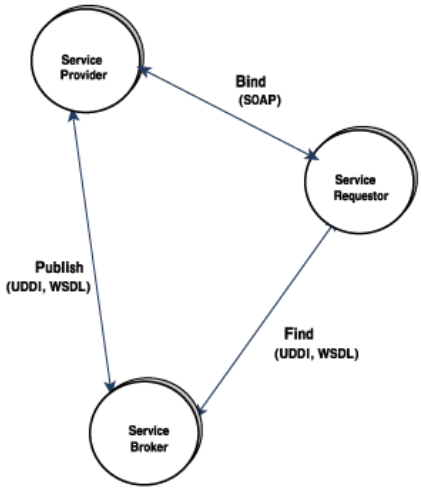
- The *Service Provider* registers all available web services with the Service Broker.
- The *Service Broker* publishes the web services for the Service Requestor to access. The information published describes the web service and its location.
- The *Service Requestor* interacts with the Service Broker to find the web services. The Service Requestor can then bind or invoke the web services.

The Service Provider hosts the web service and makes it available to clients via the Web. The Service Provider publishes the web service definition and binding information to the Universal Description, Discovery, and Integration (UDDI) registry. The Web Service Description Language (WSDL) documents contain the information about the web service, including its incoming message and returning response messages.

The Service Requestor is a client program that consumes the web service. The Service Requestor finds web services by using UDDI or through other means, such as email. It then binds or invokes the web service.

The Service Broker manages the interaction between the Service Provider and Service Requestor. The Service Broker makes available all service definitions and binding information. Currently, SOAP (an XML-based, messaging and encoding protocol format for exchange of information in a decentralized, distributed environment) is the standard for communication between the Service Requestor and Service Broker.

Figure 9.1 Standard Web Services Architecture



Web Services and Partitions

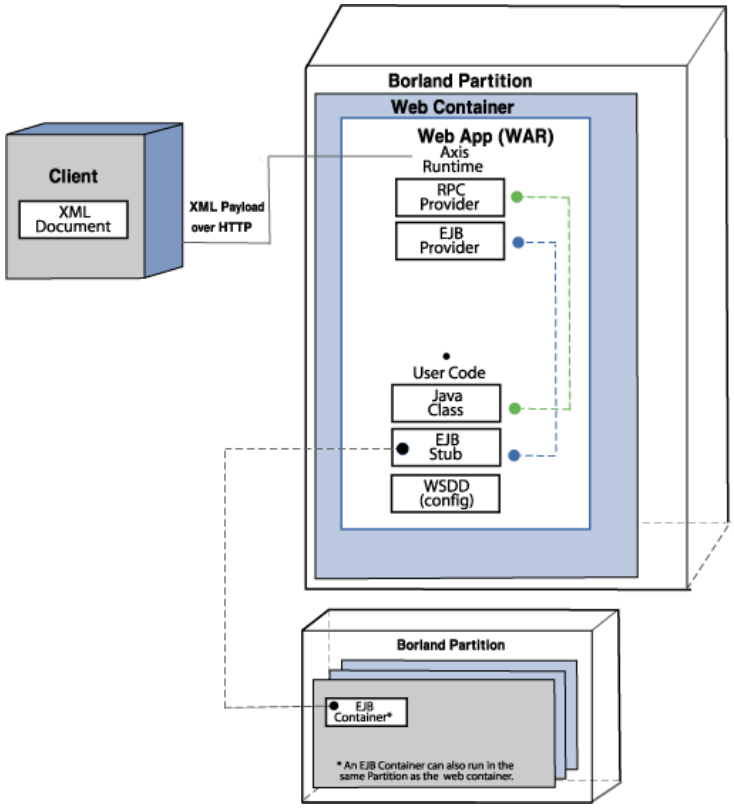
All AppServer Partitions are configured to support web services. You simply need to start a Partition and deploy WARs (or EARs containing WARs) containing web services.

Additionally, you can expose a previously deployed stateless session bean as a web service. For more information, see “Export EJB as a Web Service Wizard” in the *Management Console User's Guide*.

The Borland web services is based on the Apache Axis technology and supports dispatch of incoming SOAP web services requests to the following “Web Service providers”:

- EJB providers
- RPC/Java providers
- MDB/Java providers

Figure 9.2 Borland Web Services Architecture



Web Service providers

The Borland web services engine includes a number of providers. A *provider* is the link that connects a client web service request to the user's class on the server side.

All providers do the following:

- Create an instance of an object on which they can invoke methods. The exact way of creating this object differs from provider to provider.
- Invoke the methods on that object and pass all the parameters that the XML client sent.
- Pass the return value to the Axis Runtime engine, which then converts it to XML and sends it back to the client.

Specifying web service information in a deploy.wsdd file

When installing a new web service, you must name the web service and specify which provider the service is going to use. Each provider takes different parameters. The following describes the service providers and the required parameters for each.

Java:RPC provider

This provider assumes that the class serving the web service is in the application archive (WAR). When a web service request arrives, the RPC provider:

1. Loads the Java class associated with the service.
2. Creates a new instance of the object.
3. Invokes the specified method using reflection.

The parameters are:

- `className`: The name of the class that is loaded when a request arrives on this service.
- `allowedMethods`: The methods that are allowed to be invoked on this class. The class can have more methods than listed here; the methods listed here are available for remote invocation.

Example:

```
<service name="Animal" provider="java:RPC">
 <parameter name="className" value="com.borland.examples.web
services.java.Animal"/>
 <parameter name="allowedMethods" value="talk sleep"/>
</service>
```

Java:EJB provider

This provider assumes that the class serving the web service is an EJB.

Note You can expose a previously deployed stateless session bean as a web service. For more information, see “Export EJB as a Web Service Wizard” in the *Management Console User's Guide*.

When a web service request arrives:

1. The EJB provider looks up the bean name in JNDI initial context.
2. Locates the home class and creates a bean.
3. Invokes the specified method using reflection on the EJB stub.

The actual EJB itself must be deployed to any Partition before a client can access it.

The essential parameters are:

- `beanJndiName`: The name of the bean in JNDI.
- `homeInterfaceName`: The fully specified class of the home interface. This class must be present in the WAR.
- `className`: The name of the EJB remote interface.
- `allowedMethods`: The methods that are allowed to be invoked on this EJB, separated by spaces. The EJB can have more methods than listed here; the methods listed here are available for remote invocation.

Example:

```
<service name="Animal" provider="java:EJB">
   <parameter name="beanJndiName" value="Animal"/>
   <parameter name="homeInterfaceName"
value="com.borland.examples.webservices.ejb.AnimalHome"/>
   <parameter name="className"
value="com.borland.examples.webservices.ejb.Animal"/>
   <parameter name="allowedMethods" value="talk sleep"/>
     </service>
```

How Borland Web Services work

1. The web services server receives an XML SOAP message from a client.
2. It then:
   a. Interprets the SOAP message.
   b. Extracts the SOAP service name.
   c. Determines the appropriate provider who can respond to this service.
3. The mapping between the SOAP service and the type of provider is obtained from the Web Service Deployment Descriptor (WSDD) as part of WAR deployment.
4. The message is then passed onto the right provider. For information about the different ways in which each provider deals with the message, see "Java:RPC provider" on page 75 and "Java:EJB provider" on page 76.

Web Service Deployment Descriptors

Web services are deployed as part of a WAR. A single WAR can contain multiple web services. You can also deploy multiple WARs with each containing many web services.

The difference between a normal WAR and a WAR containing web services is the presence of an extra descriptor named `server-config.wsdd` in the `WEB-INF` directory. The `server-config.wsdd` file provides configuration information (the name of the web service, the provider, any corresponding Java classes and allowed methods).

There is one WSDD file per WAR and it contains information about all available web services within that WAR.

The typical component structure of a WAR containing web services has the following elements:

- `WEB-INF/web.xml`
- `WEB-INF/server-config.wsdd`
- `WEB-INF/classes/<classes corresponding to your web services are located here>`
- `WEB-INF/lib/<classes corresponding to your web services are located here in the packed JAR form>`

The `WEB-INF/lib` also contains some standard JARs that are necessary for the Axis Runtime engine.

To publish your Java classes as a web service, use the WSDD format to define the items that you want to deploy to the Partition. For example, an entry corresponding to a service named "BankService" can be:

```
<service name="BankService" provider="java:RPC">
   <parameter name="allowedMethods" value="create_account query_account"/>
   <parameter name="className" value="com.fidelity.Bank"/>
</service>
```

In this case, the `com.fidelity.Bank` Java class links to web service `BankService`. The class `com.fidelity.Bank` can have a number of public methods, but only the methods `create_account` and `query_account` are available through the web service.

Creating a server-config.wsdd file

To create the `server-config.wsdd`:

- Use JBuilder to generate the deployment descriptor as part of your WAR.

or

1. Use a text editor to write a `deploy.wsdd` file. Refer to the `deploy.wsdd` file in `<install_dir>/examples/webservices/java/server`.
2. Run the Tools Overview with the `deploy.wsdd` file by typing:

   ```
   prompt>java org.apache.axis.utils.Admin server deploy.wsdd
   ```

   The `server-config.wsdd` file is packaged as part of the WAR.

Viewing and Editing WSDD Properties

You can view and edit the properties of any web service deployment descriptor (WSDD) (`server-config.wsdd` file) that is packaged in a WAR file using either the Borland Management Console or the DDEditor. For more information, see “Viewing Web Services WSDD properties” or “Web Services” in the *Management Console User's Guide*.

Packaging Web Service Application Archives

To Create a WAR that can be deployed to the web services archive:

1. Make sure your web service classes are in `WEB-INF/classes` or `WEB-INF/lib`.
2. Copy the Axis toolkit libraries to `WEB-INF/lib`. The Axis libraries can be found in: `<install_dir>/lib/axis`
3. Copy the `web.xml` necessary for the Axis tool kit to WEB-INF directory. The `web.xml` can be found in: `<install_dir>/etc/axis`
4. Create a `deploy.wsdd` that has deployment information about your web services.
5. Run the Axis Admin tool on this `deploy.wsdd` to generate the `server-config.wsdd` as follows:

   ```
   java org.apache.axis.utils.Admin server deploy.wsdd
   ```

6. Copy this `server-config.wsdd` to `WEB-INF`
7. JAR your web application into a WAR file.

Borland Web Services examples

To help you get started with developing and deploying web services, we provide samples and examples for the Borland web services engine. These examples are included in your AppServer installation at:

```
<install_dir>/examples/webservices/
```

The examples that illustrate the different web service providers are located in the web services examples directory in the `Java`, `EJB`, `MDB` or `VisiBroker` folder.

Your AppServer installation also includes several Apache Axis samples in:

```
<install_dir>/examples/webservices/axis/samples/
```

Using the Web Service provider examples

The AppServer examples must be built before they are deployed and deployed before they are run. Building the examples involves generating the necessary WSDL files and packaging the application's code and descriptors into a deployable unit, in this case a WAR. This WAR can then be deployed to a Borland Partition. The application is run by invoking its client from a command-line. Building and running the examples is automated through the use of the Apache Ant utility, while deployment is performed using tools provided with AppServer.

Steps to build, deploy, and run the examples

1 **Build.** You can build all of the examples simultaneously or build each one individually. To build them all simultaneously, navigate to the:

```
/examples/webservices/
```

directory and execute the Ant command. For example:

```
C:/BDP/examples/webservices>Ant
```

builds all the examples.

To build an individual example, navigate to its specific directory and execute the Ant command.

For example:

```
C:/BDP/examples/webservices/java>Ant
```

builds only the Java Provider example.

2 **Deploy.** You deploy the examples to a running instance of AppServer. You can use the `ant deploy` target, or any of the following to deploy your WAR and JAR:

- iastool command-line utility; for more information see “iastool command-line utility” on page 311.
- Deployment Wizard; see “Deployment Wizard” in the *Management Console User's Guide*.

3 **Run.** To run an example, navigate to its directory and use the `ant run-client` command.

For example, to run the Java Provider client:

```
C:/BDP/examples/webservices/java>Ant run-client
```

Apache Axis Web Service samples

The Apache Axis web service samples are already deployed in the `axis-samples.war` file present in the Borland Partition. Since these are already pre-deployed, you do not need to use the Apache Axis deploy commands suggested in the Apache Axis User's Guide.

The Apache Axis User's Guide is also provided with the AppServer installation and is located in:

```
<install_dir>/doc/axis/user-guide.html
```

These samples illustrate the capabilities of Axis. They are unmodified from the original Apache Axis implementation and are not guaranteed to run.

Tools Overview

This section describes the various tools used in examples.

Apache ANT tool

The Apache ANT utility is a platform-independent, java-based build tool used to build the examples.

The XML build script `build.xml` is used to drive the tool. This `build.xml` file describes the various targets available for a project and the commands executed in response to those targets. The AppServer conveniently provides all necessary JARs and scripts to run the Apache Ant tool.

Java2WSDL tool

The Java2WSDL is an Apache Axis utility class that generates WSDL corresponding to a Java class. This class can accept a number of command line arguments. You can get the full usage help by running this utility without arguments as follows:

```
java org.apache.axis.wsdl.Java2WSDL
```

Note You must set your CLASSPATH to include all jar files in the `<install-dir>\lib\axis` directory, before you run the following command.

WSDL2Java tool

The WSDL2Java is an Apache Axis utility class that generates Java classes from a WSDL file. This tool can generate java stubs (used on the client side), or java skeletons (used on the server side). The generated files make it easy to develop your client or server for a given WSDL.

This class can accept a number of command line arguments. You can get the full usage help by running this utility without arguments as follows:

```
java org.apache.axis.wsdl.WSDL2Java
```

Note You must set your CLASSPATH to include all jar files in the `<install-dir>\lib\axis` directory, before you run the following command.

Axis Admin tool

The Apache Admin tool is a utility class that generates WAR level global configuration files based on deployment information specific to some web services.

The input to this utility is a XML file (typically named `deploy.wsdd`) containing deployment information about one or more web services. The Apache Admin utility adds some global definitions that are necessary and writes an output file. Use this tool as follows:

```
java org.apache.axis.utils.Admin server|client deployment-file
```

Note You must set your CLASSPATH to include all jar files in the `<install-dir>\lib\axis` directory, before you run the command.

This tool generates `server-config.wsdd` or `client-config.wsdd` based on what option you choose.

Chapter 10

Writing enterprise bean clients

Client view of an enterprise bean

A client of an enterprise bean is an application—a stand-alone application, an application client container, servlet, or applet—or another enterprise bean. In all cases, the client must do the following things to use an enterprise bean:

- Locate the bean's home interface. The EJB specification states that the client should use the JNDI (Java Naming and Directory Interface) API to locate home interfaces.
- Obtain a reference to an enterprise bean object's remote interface. This involves using methods defined on the bean's home interface. You can either create a session bean, or you can create or find an entity bean.
- Invoke one or more methods defined by the enterprise bean. A client does not directly invoke the methods defined by the enterprise bean. Instead, the client invokes the methods on the enterprise bean object's remote interface. The methods defined in the remote interface are the methods that the enterprise bean has exposed to clients.

Initializing the client

The SortClient application imports the necessary JNDI classes and the SortBean home and remote interfaces. The client uses the JNDI API to locate an enterprise bean's home interface.

A client application can also use logical names (as recommended in the various J2EE specifications) to access resources such as database connections, remote enterprise beans, and environment variables. The container, per the J2EE specification, exposes these resources as administered objects in the local JNDI name space (that is, `java:comp/env`).

Locating the home interface

A client locates an enterprise bean's home interface using JNDI, as shown in the code sample below. The client first needs to obtain a JNDI initial naming context. The code instantiates a new `javax.naming.Context` object, which in our example it calls initialContext. Then, the client uses the context `lookup()` method to resolve the name to a home interface. Note that the initialization of the initial naming context factory is EJB container/server specific.

A client application can also use logical names to access a resource such as the home interface. See "Initializing the client" on page 81 for more information.

The context's `lookup()` method returns an object of type `java.lang.Object`. Your code must cast this returned object to the expected type. The following code sample shows a portion of the client code for the sort example. The `main()` routine begins by using the JNDI naming service and its context lookup method to locate the home interface. You pass the name of the remote interface, which in this case is `sort`, to the `context.lookup()` method. Notice that the program eventually casts the results of the `context.lookup()` method to SortHome, the type of the home interface.

```
// SortClient.java
import javax.naming.InitialContext;
import SortHome; // import the bean's home interface
import Sort; // import the bean's remote interface
public class SortClient {
   ⋮
   public static void main(String[] args) throws Exception {
      javax.naming.Context context;

      // preferred JNDI context lookup
      // get a JNDI context using a logical JNDI name in the local JNDI context,
         i.e.,ejb-ref
      javax.naming.Context context = new javax.naming.InitialContext();
      Object ref = context.lookup("java:comp/env/ejb/Sort");
      SortHome home = (SortHome) javax.rmi.PortableRemoteObject.narrow
         (ref, SortHome.class);
      Sort sort = home.create();
      ... //do the sort and merge work
      sort.remove();
   }
}
```

The `main()` routine of the client program throws the generic exception coded this way, the SortClient program does not have to catch any exceptions that might occur, though if an exception occurs it will terminate the program.

Obtaining the remote interface

Now that we have obtained the home interface of an enterprise bean we can get a reference to the enterprise bean's remote interface. To do this, we use the home interface's create or finder methods. The exact method to invoke depends on the type of the enterprise bean and the methods the enterprise bean provider has defined in the home interface.

For example, the first code sample shows how SortClient obtains a reference to the Sort remote interface. Once SortClient obtains the reference to the home interface and casts it to its proper type (SortHome), then the code can create an instance of the bean and call its methods. It calls the home interface's `create()` method, which returns a reference to the bean's remote interface, Sort. (Because SortBean is a stateless session bean, its home interface has only one `create()` method and that method by definition takes no parameters.) SortClient can then call the methods defined on the remote interface—`sort()` and `merge()`—to do its sorting work. When the work finishes, the client calls the remote interface's `remove()` method to remove the instance of the enterprise bean.

Session beans

A client obtains a reference to a session bean's remote interface by calling one of the create methods on the home interface.

All session beans must have at least one `create()` method. A stateless session bean must have only one `create()` method, and that method must have no arguments. A stateful session bean can have one `create()` method, and may have additional `create()` methods whose parameters vary. If a `create()` method does have parameters, the values of these parameters are used to initialize the session bean.

The default `create()` method has no parameters. For example, the sort example uses a stateless session bean. It has, by definition, one `create()` method that takes no parameters:

```
Sort sort = home.create();
```

The cart example, on the other hand, uses a stateful session bean, and its home interface, CartHome, implements more than one `create()` method. One of its `create()` methods takes three parameters, which together identify the purchaser of the cart contents, and returns a reference to the Cart remote interface. The CartClient sets values for the three parameters—`cardHolderName`, `creditCardNumber`, and `expirationDate`—then calls the `create()` method. This is shown in the code sample below:

```
Cart cart;
   {
      String cardHolderName = "Jack B. Quick";
      String creditCardNumber = "1234-5678-9012-3456";
      Date expirationDate = new GregorianCalendar(2001,
         Calendar.JULY, 1).getTime();
      cart = home.create(cardHolderName, creditCardNumber, expirationDate);
   }
```

Session beans do not have finder methods.

Entity beans

A client obtains a reference to an entity object either through a find operation or a create operation. Recall that an entity object represents some underlying data stored in a database. Because the entity bean represents persistent data, entity beans typically exist for quite a long time; certainly for much longer than the client applications that call them. Thus, a client most often needs to find the entity bean that represents the piece of persistent data of interest, rather than creating a new entity object, which would create and store new data in the underlying database.

A client uses a find operation to locate an existing entity object, such as a specific row within a relational database table. That is, find operations locate data entities that have previously been inserted into data storage. The data may have been added to the data store by an entity bean or it may have been added outside of the EJB context, such as directly from within the database management system (DBMS). Or, in the case of legacy systems, the data may have existed prior to the installation of the EJB container.

A client uses an entity bean object's `create()` method to create a new data entity that will be stored in the underlying database. An entity bean's `create()` method inserts the entity state into the database, initializing the entity's variables according to the values in the `create()` method's parameters. A `create()` method for an entity bean always returns the remote interface, but the corresponding `ejbCreate()` method returns primary key of the entity instance.

Every entity bean instance must have a primary key that uniquely identifies it. An entity bean instance can also have secondary keys that can be used to locate a particular entity object.

Find methods and primary key class

The default find method for an entity bean is the `findByPrimaryKey()` method, which locates the entity object using its primary key value. Its signature is as follows:

```
<remote interface> findByPrimaryKey( <key type> primaryKey )
```

Every entity bean must implement a `findByPrimaryKey()` method. The `primaryKey` parameter is a separate primary key class that is defined in the deployment descriptor. The key type is the type for the primary key, and it must be a legal value type in RMI-IIOP. The primary key class can be any class—a Java class or a class you've written yourself.

For example, you have an Account entity bean that defines the primary key class AccountPK. AccountPK is a String type, and it holds the identifier for the Account bean. You can obtain a reference to a specific Account entity bean instance by setting the AccountPK to the account identifier and invoking the `findByPrimaryKey()` method, as shown in the following code sample.

```
AccountPK accountPK = new AccountPK("1234-56-789");
Account source = accountHome.findByPrimaryKey( accountPK );
```

The bean provider can define additional finder methods that a client can use.

Create and remove methods

A client can also create entity beans using create methods defined in the home interface. When a client invokes a `create()` method for an entity bean, the new instance of the entity object is saved in the data store. The new entity object always has a primary key value that is its identifier. Its state may be initialized to values passed as parameters to the `create()` method.

Keep in mind that an entity bean exists for as long as data is present in the database. The life of the entity bean is not bound by the client's session. The entity bean can be removed by invoking one of the bean's `remove()` methods—these methods remove the bean and the underlying representation of the entity data from the database. It is also possible to directly delete an entity object, such as by deleting a database record using the DBMS or with a legacy application.

Invoking methods

Once the client has obtained a reference to the bean's remote interface, the client can invoke the methods defined in the remote interface for this enterprise bean. The methods pertaining to the bean's business logic are of most interest to the client. There are also methods for getting information about the bean and its interfaces, getting the bean object's handle, testing if one bean is identical to another bean, and methods for removing the bean instance.

The next code sample illustrates how a client calls methods of an enterprise bean, in this case, a cart session bean. We pick up the client code from the point where it has created a new session bean instance for a card holder and retrieved a Cart reference to the remote interface. At this point, the client is ready to invoke the bean methods.

First, the client creates a new book object, setting its title and price parameters. Then, it invokes the enterprise bean business method `addItem()` to add the book object to a shopping cart. The `addItem()` method is defined on the CartBean session bean, and is made public through the Cart remote interface. The client adds other items (not shown here), then calls its own `summarize()` method to list the items in the shopping cart. This is followed by the `remove()` method to remove the bean instance. Notice that a client calls the enterprise bean methods in the same way that it invokes any method, such as its own method `summarize()`.

```
⋮
Cart cart;
{
   ⋮
   // obtain a reference to the bean's remote interface
   cart = home.create(cardHolderName, creditCardNumber, expirationDate);
}
// create a new book object
Book knuthBook = new Book("The Art of Computer Programming", 49.95f);
// add the new book item to the cart
cart.addItem(knuthBook);

⋮
// list the items currently in the cart
summarize(cart);
cart.removeItem(knuthBook);
⋮
```

Removing bean instances

The `remove()` method operates differently for session beans than for entity beans. Because a session object exists for one client and is not persistent, a client of a session bean should call the `remove()` method when finished with a session object. There are two `remove()` methods available to the client: the client can remove the session object with the `javax.ejb.EJBObject.remove()` method, or the client can remove the session handle with the `javax.ejb.EJBHome.remove(Handle handle)` method. See "Using a bean's handle" on page 85 for more information on handles.

While it is not required that a client remove a session object, it is considered to be good programming practice. If a client does not remove a stateful session bean object, the container eventually removes the object after a certain time, specified by a timeout value. The timeout value is a deployment property. However, a client can also keep a handle to the session for future reference.

Clients of entity beans do not have to deal with this problem as entity beans are only associated with a client for the duration of a transaction and the container is in charge of their life cycles, including their activation and passivation. A client of an entity bean calls the bean's `remove()` method only when the entity object is to be deleted from the underlying database.

Using a bean's handle

A handle is an another way to reference an enterprise bean. A handle is a serializable reference to a bean. You can obtain a handle from the bean's remote interface. Once you have the handle, you can write it to a file (or other persistent storage). Later, you can retrieve the handle from storage and use it to reestablish a reference to the enterprise bean.

However, you can only use the remote interface handle to recreate the reference to the bean; you cannot use it to recreate the bean itself. If another process has removed the bean, or the system crashed or shutdown and removed the bean instance, then an exception is thrown when the client application tries to use the handle to reestablish its reference to the bean.

When you are not sure that the bean instance will still be in existence, rather than using a handle to the remote interface, you can store the bean's home handle and recreate the bean object later by invoking the bean's create or find methods.

After the client creates a bean instance, it can use the `getHandle()` method to obtain a handle to this instance. Once it has the handle, it can write it to a serialized file. Later, the client program can read the serialized file, casting the object that it reads in to a Handle type. Then, it calls the `getEJBObject()` method on the handle to obtain the bean reference, casting the results of `getEJBObject()` to the correct type for the bean.

To illustrate, the CartClient program might do the following to utilize a handle to the CartBean session bean:

```
import java.io;
import javax.ejb.Handle;
⋮
Cart cart;
⋮
cart = home.create(cardHolderName, creditCardNumber, expirationDate);
// call getHandle on the cart object to get its handle
cartHandle = cart.getHandle();
// write the handle to serialized file
FileOutputStream f = new FileOutputStream ("carthandle.ser");
ObjectOutputStream o = new ObjectOutputStream(f);
o.writeObject(myHandle);
o.flush();
o.close();
⋮
// read handle from file at later time
FileInputStream fi = new FileInputStream ("carthandle.ser");
ObjectInputStream oi = new ObjectInputStream(fi);
//read the object from the file and cast it to a Handle
cartHandle = (Handle)oi.readObject();
oi.close();
⋮
// Use the handle to reference the bean instance
try {
   Object ref = context.lookup("cart");
   Cart cart1 = (Cart) javax.rmi.PortableRemoteObject.narrow(ref, Cart.class);
   ⋮
} catch (RemoteException e) {
   ⋮
}
⋮
```

When finished with the session bean handle, the client can remove it with the `javax.ejb.EJBHome.remove(Handle handle)` method.

Managing transactions

A client program can manage its own transactions rather than letting the enterprise bean (or container) manage the transaction. A client that lmanages its own transaction does so in exactly the same manner as a session bean than manages its own transaction.

When a client manages its own transactions, it is responsible for delimiting the transaction boundaries. That is, it must explicitly start the transaction and end (commit or roll back) the transaction.

A client uses the `javax.transaction.UserTransaction` interface to manage its own transactions. It must first obtain a reference to the `UserTransaction` interface, using JNDI to do so. Once it has the `UserTransaction` context, the client uses the `UserTransaction.begin()` method to start the transaction, followed later by the `UserTransaction.commit()` method to commit and end the transaction (or `UserTransaction.rollback()` to rollback and end the transaction). In between, the client does its queries and updates.

This code sample shows the code that a client would implement to manage its own transactions. The parts that pertain specifically to client-managed transactions are highlighted in bold.

```
⋮
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
⋮
public class clientTransaction {
   public static void main (String[] argv) {
      UserTransaction ut = null;
      InitialContext initContext = new InitialContext();
      ⋮
      ut = (UserTransaction)initContext.lookup("java:comp/UserTransaction");
      // start a transaction
      ut.begin();
      // do some transaction work
      ⋮
      // commit or rollback the transaction
      ut.commit(); // or ut.rollback();
      ⋮
   ]
]
```

Getting information about an enterprise bean

Information about an enterprise bean is referred to as metadata. A client can obtain metadata about a bean using the enterprise bean's home interface `getMetaData()` method.

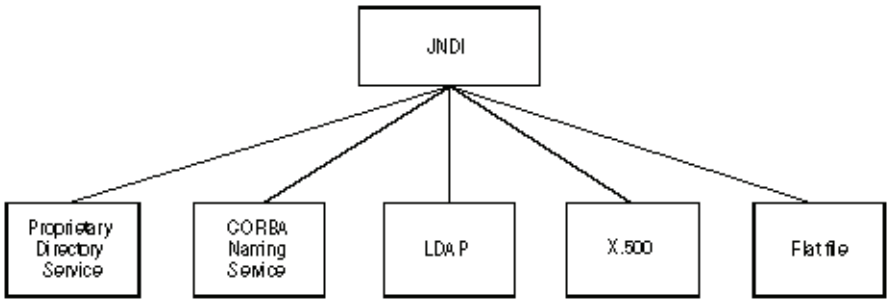
The `getMetaData()` method is most often used by development environments and tool builders that need to discover information about an enterprise bean, such as for linking together beans that have already been installed. Scripting clients might also want to obtain metadata on the bean.

Once the client retrieves the home interface reference, it can call the `getEJBMetaData()` method on the home interface. Then, the client can call the `EJBMetaData` interface methods to extract such information as:

- The bean's `EJBHome` home interface, using `EJBMetaData.getEJBHome()`.
- The bean's home interface class object, including its interfaces, classes, fields, and methods, using `EJBMetaData.getHomeInterfaceClass()`.
- The bean's remote interface class object, including all class information, using `EJBMetaData.getRemoteInterfaceClass()`.
- The bean's primary key class object, using `EJBMetaData.getPrimaryKeyClass()`.
- Whether the bean is a session bean or an entity bean, using `EJBMetaData.isSession()`. The method returns true if this is a session bean.
- Whether a session bean is stateless or stateful, using `EJBMetaData.isStatelessSession()`. The method returns true if the session bean is stateless.

Support for JNDI

The EJB specification defines the JNDI API for locating home interfaces. JNDI is implemented on top of other services, including CORBA's Naming Service, LDAP/ X.500, flat files, and proprietary directory services. The diagram below illustrates the different implementation choices. Typically, the EJB server provider selects a particular implementation of JNDI.



The technology implemented beneath JNDI is of no concern to the client. The client needs to use only the JNDI API.

EJB to CORBA mapping

There are a number of aspects to the relationship between CORBA and Enterprise JavaBeans. Three important ones are the implementation of an EJB container/server with an ORB, the integration of legacy systems into an EJB middle tier, and the access of enterprise beans from non-Java components, specifically clients. The EJB specification is currently only concerned with the third aspect.

CORBA is a very suitable and natural platform on which to implement an EJB infrastructure. CORBA addresses all of the concerns of the EJB specification with the CORBA Core specification or the CORBA Services:

- **Support for distribution.** CORBA Core and CORBA Naming Service
- **Support for transactions.** CORBA Object Transaction Service
- **Support for security.** CORBA Security Specification, including IIOP-over-SSL

Additionally, CORBA allows the integration of non-Java components into an application. These components can be legacy systems and applications, plus different kinds of clients. Back-end systems can be easily integrated using OTS and any programming language for which an IDL mapping exists. This requires an EJB container to provide OTS and IIOP APIs.

The EJB specification is concerned with the accessibility of enterprise beans from non-Java clients and provides an EJB to CORBA mapping. The goals of the EJB/CORBA mapping are:

- Supporting interoperability between clients written in any CORBA-supported programming language and enterprise beans running on a CORBA-based EJB server.
- Enabling client programs to mix and match calls to CORBA objects and enterprise beans within the same transaction.
- Supporting distributed transactions involving multiple enterprise beans running on CORBA-based EJB servers provided by different vendors.

The mapping is based on the Java-to-IDL mapping. The specification includes the following parts: mapping of distribution-related aspects, the mapping of naming conventions, the mapping of transactions, and the mapping of security. We explain each of these aspects in the following sections. Since the mapping uses new IDL features introduced by the OMG's Object-by-Value specification, interoperability with other programming languages requires CORBA 2.3-compliant ORBs.

Mapping for distribution

An enterprise bean has two interfaces that are remotely accessible: the remote interface and the home interface. Applying the Java/IDL mapping to these interfaces results in corresponding IDL specifications. The base classes defined in the EJB specification are mapped to IDL in the same manner.

For example, look at the IDL interface for an ATM enterprise session bean that has methods to transfer funds between accounts and throws an insufficient funds exception. By applying the Java/IDL mapping to the home and the remote interface, you get the following IDL interface.

```
module transaction {
   module ejb {
   valuetype InsufficientFundsException : ::java::lang::Exception {};
   exception InsufficientFundsEx {
      ::transaction::ejb::InsufficientFundsException value;
   };
   interface Atm : ::javax::ejb::EJBObject{
   void transfer (in string arg0, in string arg1, in float arg2)
      raises (::transaction::ejb::InsufficientFundsEx);
   };
   interface AtmHome : ::javax::ejb::EJBHome {
      ::transaction::ejb::Atm create ()
      raises (::javax::ejb::CreateEx);
   };
};};};};
```

Mapping for naming

A CORBA-based EJB runtime environment that wants to enable any CORBA clients to access enterprise beans must use the CORBA Naming Service for publishing and resolving the home interfaces of the enterprise beans. The runtime can use the CORBA Naming Service directly or indirectly via JNDI and its standard mapping to the CORBA Naming Service.

JNDI names have a string representation of the following form "`directory1/directory2/.../directoryN/objectName`". The CORBA Naming Service defines names as a sequence of name components.

```
typedef string Istring;
   struct NameComponent {
      Istring id;
      Istring kind;
   };
typedef sequence<NameComponent> Name;
```

Each "/" separated name of a JNDI string name is mapped to a name component; the leftmost component is the first entry in the CORBA Naming Service name.

A JNDI string name is relative to some naming context, which calls the JNDI root context. The JNDI root context corresponds to a CORBA Naming Service initial context. CORBA Naming Service names are relative to the CORBA initial context.

A CORBA program obtains an initial CORBA Naming Service naming context by calling `resolve_initial_references("NameService")` on the ORB (pseudo) object. The CORBA Naming Service does not prescribe a rooted graph for organizing naming context and, hence, the notion of a root context does not apply. The initialization of the ORB determines the context returned by `resolve_initial_references()`.

For example, a C++ Client can locate the home interface to the ATMSession bean, which has been registered with a JNDI string name "transaction/corbaEjb/atm". You first obtain the initial naming context.

```
Object_ptr obj = orb->resolve_initial_references("NameService");
NamingContext initialNamingContext= NamingContext.narrow( obj );
if( initialNamingContext == NULL ) {
   cerr << "Couldn't initial naming context" << endl;
   exit( 1 );
}
```

Then you create a CORBA Naming Service name and initialize it according to the mapping explained previously.

```
Name name = new Name( 1 );
name[0].id = "atm";
name[0].kind = "";
```

Now resolve the name on the initial naming context. Assume that you have successfully performed the initialization and that you have the context of the naming domain of the enterprise bean. We narrow the resulting CORBA object to the expected type and make sure that the narrow was successful.

```
Object_ptr obj = initialNamingContext->resolve( name );
ATMSessionHome_ptr atmSessionHome = ATMSessionHome.narrow( obj );
if( atmSessionHome == NULL ) {
   cerr << "Couldn't narrow to ATMSessionHome" << endl;
   exit( 1 );
}
```

Mapping for transaction

A CORBA-based enterprise bean runtime environment that wants to enable a CORBA client to participate in a transaction involving enterprise beans must use the CORBA Object Transaction Service for transaction control.

When an enterprise bean is deployed it can be installed with different transaction policies. The policy is defined in the enterprise bean's deployment descriptor.

The following rules have been defined for transactional enterprise beans: A CORBA client invokes an enterprise through stubs generated from the IDL interfaces for the enterprise bean's remote and home interface. If the client is involved in a transaction, it uses the interfaces provided by CORBA Object Transaction Service. For example, a C++ client could invoke the ATMSession bean from the previous example as follows:

```
try {
   ⋮
   // obtain transaction current
   Object_ptr obj = orb->resolve_initial_refernces("Current");
   Current current = Current.narrow( obj );
   if( current == NULL ) {
      cerr << "Couldn't resolve current" << endl;
      exit( 1 );
   }
// execute transaction
   try {
      current->begin();
      atmSession->transfer("checking", "saving", 100.00 );
      current->commit( 0 );
   }   catch( ... ) {
      current->rollback();
   }
}
catch( ... ) {
   ⋮
}
```

Mapping for security

Security aspects of the EJB specification focuses on controlling access to enterprise beans. CORBA defines a number of ways to define the identities, including the following cases:

- **Plain IIOP.** CORBA's principal interface was deprecated in early 1998. The principal interface was intended for determining the identity of a client. However, the authors of the CORBA security services implemented a different approach, GIOP.
- The GIOP specification contains a component called service context, which is an array of value pairs. The identifier is a CORBA long and the value is a sequence of octet. Among other purposes, entries in the service context can be used to identify a caller.
- **Secure IIOP.** The CORBA security specification defines an opaque data type for the identity. The real type of the identity is determined by the chosen security mechanism; for example, GSS Kerberos, SPKM, or CSI-ECMA.
- **IIOP over SSL.** SSL uses X.509 certificates to identify servers and, optionally, clients. When a server requests a client certificate, the server can use the certificate as a client identity.

Chapter

11

The VisiClient Container

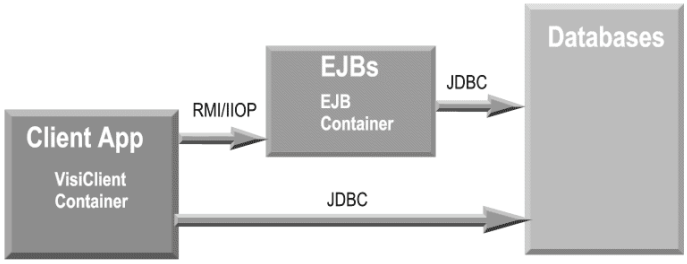
VisiClient is a container that provides a J2EE environment for services for application clients.

Containers are an integral part of J2EE applications. Most applications provide containers for each application type. Application clients depend on their containers to supply system services to all J2EE components.

Application Client architecture

J2EE application clients are first tier client programs that execute in their own Java virtual machines. Application clients obey the model for Java technology-based applications, in that they are invoked at their main method and run until the virtual machine is terminated. Like other J2EE application components, application clients depend on a container to provide system services; though in the case of application clients, these services are limited.

Figure 11.1 VisiClient architecture



Packaging and deployment

Deploying the application client components into a VisiClient container requires the specification of deployment descriptors using XML. (Refer to J2EE 1.3 Specification for more information about application clients, and their deployment into a J2EE 1.3 compliant container.)

Application clients are packaged in JAR files and include a deployment descriptor (similar to other J2EE application components). The deployment descriptor defines the EJB and the external resources referenced by the application. You can use the Borland AppServer (AppServer) Deployment Descriptor Editor for packaging and editing application client components. For more information, see “Using the Deployment Descriptor Editor” in the *Management Console User’s Guide*.

The deployment descriptor is necessary because there are a number of functions that must be configured at deployment time, such as assigning names to EJBs and their resources. The minimum requirements for deployment of an application client into a VisiClient container are:

- All the client-side classes are packaged into a JAR. See below section on required client JARs and files. A well-formed JAR should have the following:
  - Application specific classes including the class containing the application entry point (main class)
  - The JAR file must have a META-INF subdirectory with the following:
    - A manifest file
    - A standard XML file (application-client.xml), as required by J2EE 1.3 specifications
    - A vendor-specific XML file (application-client-borland.xml)
- RMI-IIOP stubs which can also be packaged separately. In this case, the file needs the classpath attribute of the manifest file set to the appropriate value. The JAR formed in this manner is deployable to a standalone container or to an EAR file. The following sections in this chapter describe this process in detail.

Benefits of the VisiClient Container

VisiClient offers users a range of benefits from the use of J2EE applications. These include:

- **Client code portability:** Applications can use logical names (as recommended in the J2EE specifications) to access resources such as database connections, remote EJBs and environment variables. The container, per the J2EE specification, exposes the resources as administered objects in the local JNDI namespace (java:comp/env).
- **JDBC Connection Pooling:** Client applications in Borland AppServer can use JDBC 2-based datasources (factories). VisiClient Container provides connection pooling to client applications in the Server that employ a JDBC 2-based datasource. For example, the VisiClient container's application uses java.net.URL, JMS, and Mail factories.

Datasource and URL factories are deployed in the in-process local JNDI subcontext that resides in the client container virtual machine on startup. Other res-ref-types (such as JMS and Mail) are configured and deployed using the relevant tools from the vendor of these products. Refer to the Deployment, Datasources and Transaction chapters of the Borland AppServer *Developer's Guide* for more information about configuration and deployment.

Document Type Definitions (DTDs)

There are two deployment descriptors for each J2EE compliant application client module. One is a J2EE standard deployment descriptor, and the other is a vendor specific file.

The XML grammar for a J2EE application client deployment descriptor is defined in the J2EE application-client Document Type Definition (DTD). The root element of the deployment descriptor for an application client is the application-client.

Note The content of XML elements are generally case sensitive. All valid application client deployment descriptors must contain the following DOCTYPE declaration:

```
<!DOCTYPE application-client PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE
Application Client
 1.3//EN';';http://java.sun.com/j2ee/dtds/application-client_1_3.dtd'>
```

The vendor-specific deployment descriptor for an application client must contain the following DOCTYPE declaration:

```
<!DOCTYPE application-client PUBLIC "-//Borland Corporation//DTD J2EE
Application Client
 1.3//EN""http://www.borland.com/devsupport/appserver/dtds/application-
client_1_3-borland.dtd">
```

The contents of the Borland-specific application client DTD are:

```
<!ELEMENT application-client (ejb-ref*, resource-ref*, property*)>
<!ELEMENT ejb-ref (ejb-ref-name, jndi-name)>
<!ELEMENT resource-ref (res-ref-name, jndi-name)>
<!ELEMENT property (prop-name, prop-type, prop-value)>
<!ELEMENT prop-name (#PCDATA)>
<!ELEMENT prop-type (#PCDATA)>
<!ELEMENT prop-value (#PCDATA)>
<!ELEMENT ejb-ref-name (#PCDATA)>
<!ELEMENT jndi-name (#PCDATA)>
<!ELEMENT res-ref-name (#PCDATA)>
```

Here ejb-ref-name and res-ref-names are the names of the corresponding elements in the J2EE XML file, and jndi-name is the absolute JNDI name with which the object is deployed in JNDI.

Example XML using the DTD

As discussed, every application client needs a pair of XML files; a standard file and a vendor-specific file.

Example of a standard file:

```
<?xml version="1.0" encoding="ISO8859_1"?>

<!DOCTYPE application-client PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE
Application Client 1.3//EN' 'http://java.sun.com/j2ee/dtds/application-
client_1_3.dtd'>
<application-client>
  <display-name>SimpleSort</display-name>
  <description>J2EE AppContainer spec compliant Sort client</description>
  <env-entry>
    <description>
      Testing environment entry
    </description>
    <env-entry-name>myStringEnv</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>MyStringEnvEntryValue</env-entry-value>
  </env-entry>
  <ejb-ref>
    <ejb-ref-name>ejb/Sort</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>SortHome</home>
    <remote>Sort</remote>
    <ejb-link>sort</ejb-link>
  </ejb-ref>
  <resource-ref>
    <description>
       reference to a jdbc datasource mentioned down in the DD section
    </description>
    <res-ref-name>jdbc/CheckingDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref></application-client>
```

Example of a vendor-specific file:

```
<?xml version="1.0"?>

<!DOCTYPE application-client PUBLIC "-//Borland Corporation//DTD J2EE
   Application Client 1.3//EN"
 "http://www.borland.com/devsupport/appserver/dtds/
   application-client_1_3-borland.dtd">
<application-client>
          <ejb-ref>
            <ejb-ref-name>ejb/Sort</ejb-ref-name>
            <jndi-name>sort</jndi-name>
          </ejb-ref>
          <resource-ref>
            <res-ref-name>jdbc/CheckingDataSource</res-ref-name>
<jndi-name>datasources/OracleDataSource</jndi-name>
          </resource-ref>
</application-client>
```

For more information about environment entries, ejb-refs, or resource-refs, see the relevant sections of Sun Microsystems' EJB 2.0 specifications at www.java.sun.com/j2ee.

Sample code

This example shows the usage of the logical local JNDI naming context. It shows how a client uses the deployment descriptors specified in the preceding section.

```
// get a JNDI context using the Naming service and create a remote object

    javax.naming.Context context = new javax.naming.InitialContext();
    Object ref = context.lookup("java:comp/env/ejb/Sort");
    SortHome home = (SortHome) javax.rmi.PortableRemoteObject.narrow(ref,
     SortHome.class);
    Sort sort = home.create();
    // get the value of an environment entry using JNDI
    Object envValue = context.lookup("java:comp/env/myStringEnv");
    System.out.println("Value of env entry = "+ (java.lang.String) envValue );
    // locate a  UserTransaction object
    javax.transaction.UserTransaction userTransaction =
      (javax.transaction.UserTransaction)  context.lookup("java:comp/
UserTransaction");

userTransaction.begin();
    // locate the datasource using resource-ref name
    Object resRef = context.lookup("java:comp/env/jdbc/CheckingDataSource");
    java.sql.Connection conn = ((javax.sql.DataSource)resRef).getConnection();
    //do some database work.
    userTransaction.commit();
⋮
```

Support of references and links

During application assembly and deployment you must verify that all EJB and resource references have been properly linked. For more information about EJB and resource references, consult Sun Microsystems' EJB 2.0 and J2EE 1.3 specifications.

The Borland AppServer client container supports the use of ejb-links. In the case of a standalone JAR file, the ejb-links have to be resolved before the JAR is deployed. There must be a JNDI name specified for the target bean in the vendor-specific section of the client deployment descriptor.

For a client JAR that is part of an Enterprise Application Archive (EAR), the JNDI name of the target EJB may live in a different ejb-jar. The client verify tool checks that the target EJB with the name specified in the ejb-link tag exists.

During runtime, the container resolves (locates) the target EJB corresponding to the ejb-link name in the EAR and uses the JNDI name of the target EJB. Note that application clients run in their own Java virtual machines. EJB-links are not optimized for application clients like they are for EJBs referring to another EJB located in the same container.

Keep the following rules in mind when working with EJB references and ejb-links in deployment descriptors for application client containers:

1 An ejb-ref that is not an ejb-link must have an entry in a Borland-specific file containing the JNDI name of the referenced (target) EJB.

2 An ejb-ref that has an ejb-link element must follow these rules:

- If the ejb-ref is in a client JAR and is a standalone JAR, rule 1 applies. That is, it should have a Borland-specific file with the JNDI name resolved in the deployment descriptor within the (same) JAR.
- If the ejb-ref is in a client-jar embedded in an application archive (an EAR), the JNDI name of the target EJB is not required to exist in the application-client-borland.xml file. In this case, the name in the ejb-link element is composed of a path name specifying the fully qualified path to the ejb-jar containing the referenced enterprise bean with the ejb-name of the target bean appended and separated from the path name by “#”. The path name is relative to the JAR file containing the application client that is referencing the enterprise bean. This allows multiple enterprise beans with the same ejb-name to be uniquely identified.

If the path is not specified, container picks first matching EJB-name that it finds from list of EJB JARs inside EAR and throws an exception if doesn't find a bean with same name in ejb-link element.>

Using the VisiClient Container

The following command line demonstrates the use of the VisiClient Container:

```
Prompt% appclient <client-archive> [-uri <uri>] [client-arg1 client-arg2 ..]
```

The following table describes VisiClient container command line elements and definitions

Table 11.1 Elements in a VisiClient container command

Element	Definition
<client-archive>	A standalone client JAR or EAR containing client JAR.
-uri	The relative location of the client JAR inside an EAR file. This is required for EAR contained JAR files.
<client-args>	Space separated list of arguments passed to the client's main class.

VisiClient Container usage example

The following command lines demonstrate usage of an application client. In the example, the `appclient` launcher sets the classpath required to launch VisiClient.

This example is also located in the Hello example in the `install_dir/examples/j2ee/hello` directory. When your server (EJB container) is up, to run a client embedded inside an EAR file, the command is:

```
appclient me install_dir\examples\j2ee\build\hello\hello.ear -uri
helloclient.jar
```

To run a client in a standalone JAR file, the command is:

```
appclient me install_dir\examples\j2ee\build\hello\client\helloclient.jar
```

Running a J2EE client application on machines not running AppServer

To run a J2EE application client on a client machine that does not have Borland AppServer installed on it, make sure to copy the following VisiClient files to your client machine and run the following processes.

1 Copy the following JAR files from `<install_dir>/lib` to client machine:

- `lm.jar`
- `xmlrt.jar`
- `asrt.jar`
- `vbjorb.jar`
- `vbsec.jar`
- `jsse.jar`
- `jaas.jar`
- `jcert.jar`
- `jnet.jar`
- `vbejb.jar`

2 Copy the following JAR file from `<install_dir>/jms/tibco/clients/java` to client machine:

- `tibjms.jar`

3 Copy `<install_dir>/bin/appclient.config` to client machine.

4 Copy `<install_dir>/BES/bin/appclient.exe` to client machine.

To run the J2EE client using the appclient:

1 Set the PATH to `appclient.exe` and JDK.

2 Edit the `appclient.config` to change `JAVA_HOME`, and `lib PATH`.

3 Run the J2EE client from `<client_application_folder>/client`.

Embedding VisiClient Container functionality into an existing application

As an alternative to deploying and running a client application in the VisiClient container, it is possible to use a programmatic approach to embed the client container's functionality into an existing application. In this case, the client application can be started in a common Java fashion by running a class implementing the `main()` method.

To embed the VisiClient container functionality into your application, you need to call the following method:

```
public static void com.borland.appclient.Container.init
  (java.io.InputStream   deploymentDescriptorSun,
  java.io.InputStream deploymentDescriptorBorland)
throws IllegalArgumentException;
```

This method will create and populate the “java:comp/env” naming context based on the information provided in the pair of Sun and Borland deployment descriptors. The `deploymentDescriptorSun` and `deploymentDescriptorBorland` parameters must represent text XML data corresponding to the deployment descriptors. An `IllegalArgumntException` exception is thrown if the data provided is not recognized as a valid deployment descriptor.

Sample code

This example shows usage of this method:

```
public static void main (String[] args) {
   ⋮
     // load deployment descriptor files
     java.io.FileInputStream ddSun = new
   java.io.FileInputStream("META-INF/application-client.xml");
     java.io.FileInputStream ddBorland = new
   java.io.FileInputStream("META-INF/application-client-borland.xml");
     // initialize client container
     com.borland.appclient.Container.init(ddSun, ddBorland);
     // lookup ejb in JNDI using an ejb-ref
   javax.naming.Context context = new javax.naming.InitialContext();
     Object ref = context.lookup ("java:comp/env/ejb/hello");
   ⋮
}
```

Note Only application client descriptors can be loaded using this method. This means that all ejb-refs must be resolved or located by specifying the jndi-name in the Borland descriptor. This cannot be done using the ejb-link in the Sun descriptor since using ejb-link requires complete knowledge of the whole application including application and EJB JAR deployment descriptors.

Use of Manifest files

VisiClient container relies on the presence of a manifest file to obtain information about launching an application. The manifest file should be saved in the META-INF subdirectory of the client archive. The relevant attributes in the manifest file for the VisiClient container are:

- The main class to be launched by the container on startup. This is an application entry point which must be present in the manifest file.
- The classpath of the dependencies of the main class. If the client-jar is self-contained, or if dependencies are specified using the system `CLASSPATH` during application launch, this attribute can be ignored.

Example of a Manifest file

An example of a Manifest file is shown below.

```
Manifest-Version: 1.0
Main-Class: SortClient
Class-Path:
```

This example shows the container will execute by loading the main method of the class specified in the `Main-Class` attribute of the Manifest file. In this example it is `SortClient`. The container expects to have a method with the following signature in this class:

```
public static void main(String[ ] args) throws Exception {...}
```

The container will report an error and exit if it doesn't find the main method. The client verify utility, which comes with VisiClient, tries to locate a main class and reports an error if it doesn't find one.

Exception handling

Application client code is responsible for taking care of any exceptions that are generated during the program execution. Any unhandled exceptions are caught by the container which will log them and terminate the Java virtual machine process.

Using resource-reference factory types

The client application deployed in a client container can use the VisiTransact JDBC connection pooling and Prepared Statement re-use facilities. Refer to the Deployment, and Transaction chapters of the Borland AppServer *Developer's Guide* for details about configuration and deployment. Client applications in AppServer can use JDBC 2-based datasources.

Note that just like `javax.sql.DataSource` (which is one of the possible res-ref-types) VisiClient allows the application to use URL, JMS, and Mail factories as the resource-ref types.

`java.net.url` and `java_mail.session` factories are deployed in the in-process local JNDI subcontext that resides in the client container virtual machine on startup. Other res-ref-types like JMS and Mail should be configured and deployed using the relevant vendor tools for these products.

Other features

The AppServer includes a number of extra features in the VisiClient container in addition to the requirements for the J2EE specification. These include:

- **User Transaction interface:** This is available in the java:comp/env name space and can be looked up using JNDI. It supports transaction demarcation, and propagation.
- **Client Verify Tool:** This runs on standalone client JARs or client JARs embedded in an EAR file. The verify tool enforces the following rules:
  - The manifest file in the client JAR has the main class specified.
  - The JAR/EAR is valid (it has the correct required manifest entries).
  - ejb-refs are valid (that is, a JNDI name for the target EJB is specified in the Borland-specific file).
  - If ejb-ref is an ejb-link, then the archive should be an EAR file. There must also be an EJB with the same name as the ejb-link value in the EAR file.
  - Resource references are valid.

Using the Client Verify tool

The following command line demonstrates the use of the Client Verify tool:

```
iastool -verify -src <srcjar> -role <DEVELOPER| ASSEMBLER| DEPLOYER>
```

Usage examples of Client Verify tool:

```
iastool -verify -src sort.jar -role DEVELOPER
iastool -verify -src sort.ear clients/sort_client.jar -role DEVELOPER
```

For more information see “verify” on page 336 on the available options.

Chapter

12

Caching of Stateful Session Beans

The EJB Container supports stateful session enterprise beans using a high-performance caching architecture based on the Java Session Service (JSS). There are two pools of objects: the ready pool and the passive pool. Enterprise beans transition from the ready pool to the passive pool after a configurable timeout. Transitioning an enterprise bean to the passive pool stores the enterprise bean's state in a database. Passivation of stateful sessions exists for two purposes:

1 Maximize memory resources

2 Implement failover

Configuring Borland's JSS implementation is discussed in Chapter 6, “Java Session Service (JSS) configuration.” This document explains the use of the properties that control the passivation and persistence of individual session objects.

Passivating Session Beans

At deployment time, the deployer uses the Borland AppServer's (AppServer) tools to set a passivation timeout for the EJB Container in a particular Partition. The container regularly polls active session beans to determine when they are last accessed. If a session bean has not been accessed during the timeout period, its state is sent to persistent storage and the bean instance is removed from memory.

Simple Passivation

Passivation timeouts are set at the container-level. You use the property `ejb.sfsd.passivation_timeout` to configure the length of time a session bean can go un-accessed before its state is persisted and its instance removed from memory. This length of time is specified in seconds. The default value is five seconds. This property can be set in the `partition.xml` properties file for the Partition you are configuring. This file is located in:

```
<install_dir>/var/domains/base/configurations/<configuration_name>
/ mos/<partition_name>/adm/properties
```

Edit this file to set the `ejb.sfsb.passivation_timeout` property.

If you set this property to a non-zero value, you can also set the integer property `ejb.sfsb.instance_max` for each deployed session bean in their deployment descriptors. This property defines the maximum number of instances of a particular stateful session bean that are allowed to exist in the EJB container's memory at the same time. If this number is reached and a new instance of a stateful session needs to be allocated, the EJB container throws an exception indicating lack of resources. 0 is a special value. It means no maximum set.

If the maximum number of stateful sessions defined by the `ejb.sfsb.instance_max property` is reached, the EJB container blocks a request for an allocation of a new bean for the time defined by the integer property `ejb.sfsb.instance_max_timeout`. The container will then wait for the number to drop below this value before throwing an exception indicating a lack of resources. This property is defined in ms (1/1000th of second). 0 is a special value. It means not to wait and throw an exception indicating lack of resources immediately.

Aggressive Passivation

One of the key advantages in the use of JSS is its ability to fail over. Several containers implementing JSS can be configured to use the same persistent store, allowing them to fail over to each other. Setting up the JSS for failover is discussed in Chapter 6, “Java Session Service (JSS) configuration.” To facilitate taking advantage of the JSS failover capability, Borland provides the option of using aggressive passivation.

Aggressive passivation is the storage of session state regardless of its timeout. A bean that is set to use aggressive passivation will have its session state persisted every time it is polled, although its instance will not be removed from memory unless it times out. In this way, if a container instance fails in a cluster, a recently-stored version of the bean is available to other containers using identical JSS instances communicating with the same backend. As in simple passivation, if the bean times out, it will still be removed from memory.

Again, aggressive passivation is set Partition-wide using the boolean property `ejb.sfsb.aggressive_passivation`. Setting the property to `true` (the default) stores the session's state regardless of whether it was accessed before the last passivation attempt. Setting the property to false allows the container to use only simple passivation. Again, this property is set in the container's properties file `partition.xml` located in:

```
<install_dir>/var/domains/base/configurations/<configuration_name>
/ mos/<partition_name>/adm/properties
```

Bear in mind that although using aggressive passivation aids in failover, it also results in a performance hit since the container accesses the database more often. If you configure the JSS to use a non-native database (that is, you choose not to use JDataStore), the loss of performance can be even greater. Be aware of the tradeoff between availability and performance before you elect to use aggressive passivation.

Sessions in secondary storage

Most sessions are not kept in persistent storage forever after they timeout. Borland provides a mechanism for removing stored sessions from the database after a discrete period of time known as the *keep alive timeout*. The keep alive timeout specifies the minimum amount of time in seconds to persist a passivated session in stateful storage. The actual amount of time it is kept in the database can vary, since it is not wise from a performance standpoint to constantly poll the database for unused sessions. The actual amount of time a session is persisted is at least the value of the keep alive timeout and not more than twice the value of the keep alive timeout.

Unlike the other passivation properties discussed above, the keep alive timeout can be specified either Partition-wide and/or on the individual session bean. If you set a keep alive timeout for a specific bean, its value will take precedence over any container-wide values. If you do not specify a keep alive timeout for a particular bean, it will use the Partition-wide value.

Setting the keep alive timeout in Containers

The Borland JSS implementation uses the property `ejb.sfsb.keep_alive_timeout` to specify the amount of time (in seconds) to maintain a passivated session in stateful storage. The default value is 86,400 seconds, or twenty-four hours. Like the other properties discussed above, you set the keep alive timeout in the container properties file:

```
<install_dir>/var/domains/base/configurations/<configuration_name>
/ mos/<partition_name>/adm/properties
```

Remember that any value you specify here can be overridden by setting a keep alive timeout for a specific session bean.

Setting the keep alive timeout for a particular session bean

You may wish to have certain session beans hosted in your container have their passivated states stored for greater or lesser periods of time than others. You can use the `<timeout>` element in the `ejb-borland.xml` file to set the keep alive timeout for a particular bean. The DTD element for a session bean provides this element:

```
<!ELEMENT session (ejb-name, bean-home-name?, bean-local-home-name?, timeout?,
ejb-ref*, ejb-local-ref*, resource-ref*, resource-env-ref*, property*)>
```

For example, let's say we have a simple stateful session bean called `personInfo` collecting a bit of personal information for simple message forum. We might be inclined to keep this session highly-available, without aggressive passivation, and have little need to store it in our database for more than a few minutes if it passivates. Since the rest of our session beans need to be kept in stateful storage a bit longer if they passivate, we'll use the Borland-specific deployment descriptor for the bean's JAR to set a shorter keep alive timeout, say 300 seconds (five minutes). In our `ejb-borland.xml` deployment descriptor, we'd have the following:

```
<ejb-jar>
 <enterprise-beans>
  <session>
   <ejb-name>personInfo</ejb-name>
   <timeout>300</timeout>
  </session>
 </enterprise-beans>
</ejb-jar>
```

This value will override any values we entered in the `ejbcontainer.properties` file while allowing other hosted sessions to use the default value found there.

Chapter

13

Entity Beans and CMP 1.1 in Borland AppServer

Here we'll examine how entity beans are deployed in the Borland AppServer (AppServer) and how persistence of entities can be managed. This is not, however, an introduction to entity beans and should not be treated as such. Rather, this document will explore the implications of using entity beans within Borland Partitions. We'll discuss descriptor information, persistence options, and other container-optimizations. Information on the Borland-specific deployment descriptors and implementations of Container-Managed Persistence (CMP) will be documented in favor of general EJB information that is generally available from the J2EE Specifications from Sun Microsystems.

Entity Beans

Entity beans represent a view of data stored in a database. Entity beans can be fine-grained entities mapping to a single table with a one-to-one correspondence between entity beans and table rows. Or, entity beans can span multiple tables and present data independent of the underlying database schema. Entity beans can have relationships with one another, can be queried for data by clients, and can be shared among different clients.

Deploying your Entity Bean to one of the AppServer Partitions requires that it be packaged as a part of a JAR. The JAR must include two descriptor files: `ejb-jar.xml` and the proprietary `ejb-borland.xml` file. The `ejb-jar.xml` descriptor is fully-documented at the Sun Java Center. The DTD for `ejb-borland.xml` is reproduced in this document and its usage documented here. The Borland proprietary descriptor contains a number of properties that can be set to optimize container performance and manage the persistence of your entity beans.

Container-managed persistence and Relationships

Borland's EJB container provides tools that generate the database access calls at the time that the entity bean is deployed; that is, when the entity bean is installed into a Partition. The tools use the deployment descriptors to determine the instance fields for which they must generate database access calls. Instead of coding the database access directly in the bean, the bean provider of a container-managed entity bean must specify in the deployment descriptor those instance fields for which the container tools must generate access calls. The container has sophisticated deployment tools capable of mapping the fields of an entity bean to its data source.

Container-managed persistence has many advantages over bean-managed persistence. It is simpler to code because bean provider does not have to code the database access calls. Handling of persistence can also be changed without having to modify and recompile the entity bean code. The Deployer or Application Assembler can do this by modifying the deployment descriptor when deploying the entity bean. Shifting the database access and persistence handling to the container not only reduces the complexity of code in the bean, it also reduces the scope of possible errors. The bean provider can focus on debugging the business logic of the bean rather than the underlying system issues.

The EJB 2.0 specification allows entity beans that use container-managed persistence to also have container-managed relationships among themselves. The container automatically manages bean relationships and maintain the referential integrity of these relationships. This differs from the EJB 1.1 specification, which only allowed you to expose a bean's instance state through its remote interface.

Just as you defined container-managed persistence fields in a bean's deployment descriptor, you can now define container-managed relationship fields in the deployment descriptor. The container supports relationships of various cardinalities, including one-to-one, one-to-many, and many-to-many.

Implementing an entity bean

Implementing an entity bean follows the rules defined in the EJB 1.1 and 2.0 specifications. You must implement a home interface, a remote interface or a local interface (if using the 2.0 container-managed persistence), and the entity bean implementation class. The entity bean class must implement the methods that correspond to those declared in the remote or local and home interfaces.

Packaging Requirements

Like session beans, entity beans can expose their methods with their interfaces. Each Entity Bean must also have corresponding entries in its JAR's deployment descriptors. The standard deployment descriptor, `ejb-jar.xml` contains essentially three different types of deployment information. These are:

1. **General Bean Information:** This corresponds to the `<enterprise-beans>` elements found in the descriptor file and is used for all three types of beans. This information also includes information on the bean's interfaces and class, security information, environmental information, and even query declarations.
2. **Relationships:** This corresponds to the `<relationships>` elements found in the descriptor file and applies to entity beans using CMP only. This is where container-managed relationships are spelled out.

3 **Assembly Information:** This corresponds to the `<assembly-descriptor>` element which explains how the beans interact with the application as a whole. Assembly information is broken down into four categories:

- Security Roles: simple definitions of security roles used by the application. Any security role references you defined for your beans must also be defined here.
- Method Permissions: each method of each bean can have certain rules about their execution. These are set here.
- Container-Transactions: this specifies the transaction attributes as per the EJB 2.0 specification for each method participating in a transaction
- Exclude List: methods to be uncalled by anyone

All of these can be accessed through the Deployment Descriptor Editor. You should refer to the EJB 2.0 specification for DTD information and the proper use of the descriptor files.

Entity Bean Primary Keys

Each Entity Bean must have a unique primary key that used to identify the bean instance. The primary key can be represented by a Java class that must be a legal value type in RMI-IIOP. Therefore, it extends the `java.io.Serializable` interface. It must also provide an implementation of the `Object.equals(Object other)` and `Object.hashCode()` methods.

Normally, the primary key fields of entity beans must be set in the `ejbCreate()` method. The fields are then used to insert a new record into the database. This can be a difficult procedure, however, bloating the method, and many databases now have built-in mechanisms for providing appropriate primary key values. A more elegant means of generating primary keys is for the user to implement a separate class that generates primary keys. This class can also implement database-specific programming logic for generating primary keys.

Generating primary keys from a user class

With enterprise beans, the primary key is represented by a Java class containing the unique data. This primary key class can be any class as long as that class is a legal value type in RMI-IIOP, meaning it extends the `java.io.Serializable` interface. It must also provide an implementation of the `Object.equals(Object other)` and `Object.hashCode()` methods, two methods which all Java classes inherit by definition.

The primary key class can be specific to an particular entity bean class. That is, each entity bean can define its own primary key class. Or, multiple entity beans can share the same primary key class.

The bank application uses two different entity beans to represent savings and checking accounts. Both types of accounts use the same field to uniquely identify a particular account record. In this case, they both use the same primary key class, AccountPK, to represent the unique identifier for either type of account. The following code shows the definition of the account primary key class:

```
public class AccountPK implements java.io.Serializable {
   public String name;
   public AccountPK() {}
   public AccountPK(String name) {
      this.name = name;
   }
}
```

Generating primary keys from a custom class

To generate primary keys from a custom class, you must write a class that implements the `com.borland.ejb.pm.PrimaryKeyGenerationListener` interface.

Support for composite keys

Primary keys are not restricted to a single column. Sometimes, a primary key is composed of more than one column. In a more realistic example, a course is not identified merely by its name. Instead, the primary key for each course record can be the department in which the course is offered and the course number itself. The department code and the course number are separate columns in the Course table. A select statement that retrieves a particular course, or all courses in which a student is enrolled, must use the entire primary key; that is, it must consider both columns of the primary key.

The Borland CMP engine supports composite primary keys. You can use keys with multiple columns in the where clause of a select statement. You can also select all fields of a compound key in the select clause portion of the statement.

For the where clause, specify multiple field names in the same manner that you specify single field names. Use “and” to separate each field. The format is

```
<column> = :<parameter>[ejb/<entity bean>]
```

Note that the equal (=) sign is one of several possible notations. You could also specify greater than (>), less than (<), greater than or equal (>=), or less than or equal (<=). The colon (:) notation indicates parameter substitution. The parameter field is specified with the bean name first, followed by a dot (.), then the bean attribute.

For example, to find all students taking Art 205, Renaissance Art where classes are identified by the department (Art) and the course number (205), you might have the following select statement defined for the finder method `findByCourse()`:

```
SELECT sname FROM Enrollment WHERE course_department = :c.department[ejb/
Course] AND
   course_number = :c.number[ejb/Course]
```

You can also have the select statement return multiple fields from a compound key. In the select clause of the select statement, list the fields, separated by commas. Note that you use the same dot notation as for parameters; that is, specify the entity bean name, followed by a dot (.), then the attribute name. For example, the finder method `findByStudent()` can have the following select statement:

```
SELECT c.department, c.number FROM Enrollment WHERE student_name = :s
```

Reentrancy

By default, entity beans are not reentrant. When a call within the same transaction context arrives at the entity bean, it causes the exception `java.rmi.RemoteException` to be thrown.

You can declare an entity bean reentrant in the deployment descriptor; however, take special care in this case. The critical issue is that a container can generally not distinguish between a (loopback) call within the same transaction and a concurrent invocation (in the same transaction context) on that same entity bean.

When the entity bean is marked reentrant, it is illegal to allow a concurrent invocation within the same transaction context on the bean instance. It is the programmer's responsibility to ensure this rule.

Container-Managed Persistence in AppServer

The AppServer's EJB Container is fully J2EE 1.3 compliant. The bean provider designs persistence schemas for their entity beans, determined the methods for accessing container-managed fields and relationships, and defines these in the beans' deployment descriptor. The deployer maps this persistence schema to the database and creates any other necessary classes for the beans' maintenance.

Information on J2EE 1.3 entity beans and CMP 2.0 is found in the Chapter 15, “Using Borland AppServer Properties for CMP 2.x.”

AppServer CMP engine's CMP 1.1 implementation

While you don't have to be an expert on all aspects of the Borland CMP engine to use it effectively, it is helpful to have some knowledge of certain areas. This section provides information on the areas that users of the CMP engine should understand. In particular, it focuses on the deployment descriptor file and the XML statements contained within the file.

Before continuing, there are some key things to note in the implementation of an entity bean that uses 1.1 container-managed persistence:

- The entity bean has no implementations for finder methods. The EJB Container provides the finder method implementations for entity beans with container-managed persistence. Rather than providing the implementation for finder methods in the bean's class, the deployment descriptor contains information that enables the container to implement these finder methods.
- The entity bean declares all fields `public` that are managed by the container for the bean. The CheckingAccount bean declares `name` and `balance` to be public fields.
- The entity bean class implements the seven methods declared in the `EntityBean` interface: `ejbActivate()`, `ebjPassivate()`, `ejbLoad()`, `ejbStore()`, `ejbRemove()`, `setEntityContext()`, and `unsetEntityContext()`. However, the entity bean is required to provide only skeletal implementations of these methods, though it is free to add application-specific code where appropriate. The CheckingAccount bean saves the context returned by `setEntityContext()` and releases the reference in `unsetEntityContext()`. Otherwise, it adds no additional code to the `EntityBean` interface methods.
- There is an implementation of the `ejbCreate()` method (because this entity bean allows callers of the bean to create new checking accounts), and the implementation initializes the instance's two variables, account name and balance amount, to the argument values. The `ejbCreate()` method returns a `null` value because, with container-managed persistence, the container creates the appropriate reference to return to the client.
- The entity bean provides the minimal implementation for the `ejbPostCreate()` method, though this method could have performed further initialization work if needed. For beans with container-managed persistence, it is sufficient to provide just the minimal implementation for this method because `ejbPostCreate()` serves as a notification callback. Note that the same rule applies to the methods inherited from the `EntityBean` interface as well.

Providing CMP metadata to the Container

According to the EJB Specification, the deployer must provide CMP metadata to the EJB container. The Borland Container captures the CMP-relevant metadata in the XML deployment descriptor. Specifically, the Borland Container uses the vendor-specific portion of the deployment descriptor for the CMP metadata.

This section illustrates some of the information that needs to be provided for container-managed finder methods, particularly if you are constructing container-managed finder methods at the command line level. Because it is not an exhaustive reference, you should refer to the DTD of the deployment descriptor for the detailed syntax. Look for the syntax for the finder methods and Object-Relation (OR) mapping metadata.

Constructing finder methods

When you construct a finder method, you are actually constructing an SQL select statement with a where clause. The select statement includes a clause that states what records or data are to be found and returned. For example, you might want to find and return checking accounts in a bank. The where clause of the select statement sets limits on the selection process; that is, you might want to find only those checking accounts with a balance greater than some specified amount, or accounts with a certain level of activity per month. When the Container uses container-managed persistence, you must specify the terms of the where clause in the deployment descriptor.

For example, suppose you have a finder method called `findAccountsLargerThan(int balance)` and you are using container-managed persistence. This finder method attempts to find all bank accounts with a balance greater than the specified value. When the Container executes this finder method, it actually executes a select statement whose where clause tests the account balances against the `int` value passed as a parameter to the method. Because we're using container-managed persistence, the deployment descriptor needs to specify the conditions of the where clause; otherwise, the Container does not know how to construct the complete select statement.

The value of the where clause for the `findAccountsLargerThan(int balance)` method is "`balance > :balance`". In English, this translates to: "the value of the balance column is greater than the value of the parameter named `balance`." (Note that there is only one argument to the finder method, an `int` value.)

The default container-managed persistence implementation supports this finder method by constructing the complete SQL select statement, as follows:

```
select * from Accounts where ? > balance
```

The CMP engine then substitutes "?" with the `int` parameter. Lastly, the engine converts the result set into either an `Enumeration` or `Collection` of primary keys, as required by the EJB Specification.

It is possible to inspect the various SQL statements that the CMP implementation constructs. To do this, enable the `EJBDebug` flag on the container. When that flag is enabled, it prints the exact statements constructed by the Container.

While other EJB Container products use code generation to support CMP, the Borland Container does not use code generation because it has serious limitations. For example, code generation makes it difficult to support a "tuned update" feature, because of the great number of different update statements to container-managed fields that are required.

Constructing the where clause

The `where` clause is a necessary part of select statements when you want to delimit the extent of the returned records. Because the `where` clause syntax can be fairly complex, you must follow certain rules in the XML deployment descriptor file so that the EJB Container can correctly construct this clause.

To begin with, you are not obligated to use the literal "where" in your `<where-clause>`. You can construct a where clause without this literal and rely on the Container to supply it. However, the Container only does this if the `<where-clause>` is not an empty string; it leaves empty strings empty. For example, you could define a where clause as either:

```
<where-clause> where a = b </where-clause>
```

or:

```
<where-clause> a = b </where-clause>
```

The Container converts `a = b` to the same `where` clause, `where a = b`. However, it leaves unmodified an empty string defined as `<where-clause>` "" `</where-clause>`.

Note The empty string makes it easy to specify the `findAll()` method. When you specify just an empty string, the Container construes that to mean the following:

```
select [values] from [table];
```

Such a select statement would return all values from a particular table.

Parameter substitution

Parameter substitution is an important part of the `where` clause. The Borland EJB Container does parameter substitution wherever it finds the standard SQL substitution prefix colon (`:`). Each parameter for substitution corresponds to a name of a parameter in the finder specification found in the XML descriptor.

For example, in the XML deployment descriptor, you might define the following finder method which takes a parameter `balance` (note that `balance` is preceded by a colon):

```
<finder>
   <method-signature>findAccountsLargerThan(float balance)</method-signature>
   <where-clause>balance > :balance</where-clause>
</finder>
```

The Container composes a SQL select statement whose `where` clause is:

```
balance > ?
```

Note that the `:balance` parameter in the deployment descriptor becomes a question mark (`?`) in the equivalent SQL statement. When invoked, the Container substitutes the value of the parameter `:balance` for the `?` in the `where` clause.

Compound parameters

The Container also supports compound parameters; that is, the name of a table followed by a column within the table. For this, it uses the standard dot (`.`) syntax, where the table name is separated from the column name by a dot. These parameters are also preceded by a colon.

For example, the following finder method has the compound parameters `:address.city` and `:address.state`:

```
<finder>
   <method-signature>findByCity(Address address)</method-signature>
   <where-clause>city = :address.city AND state = :address.state</where-clause>
</finder>
```

The `where` clause uses the `city` and `state` fields of the `address` compound object to select particular records. The underlying `Address` object could have Java Beans-style getter methods that correspond to the attributes `city` and `state`. Or, alternatively, it could have public fields that correspond to the attributes.

Entity beans as parameters

An entity bean can also serve as a parameter in a finder method. You can use an entity bean as a compound type. To do so, you must tell the CMP engine which field to use from the entity bean's passed reference to the SQL query. If you do not use the entity bean as a compound type, then the Container substitutes the bean's primary key in the `where` clause.

For example, suppose you have a set of `OrderItems` entity beans associated with an `Order` entity object. You might have the following finder method:

```
java.util.Collection OrderItemHome.findByOrder(Order order);
```

This method returns all `OrderItems` associated with a particular `Order`. The deployment descriptor entry for its `where` clause would be:

```
<finder>
   <method-signature>findByOrder(Order order)</method-signature>
   <where-clause>order_id = :order[ejb/orders]</where-clause>
</finder>
```

To produce this where clause, the Container substitutes the primary key of the `Order` object for the string `:order[ejb/orders]`. The string between the brackets (in this example, `ejb/orders`) must be the `<ejb-ref>` corresponding to the home of the parameter type. In this example, `ejb/orders` corresponds to an `<ejb-ref>` pointing to `OrderHome`.

When you use an EJBObject as a compound type (using the dot notation), you are actually accessing the underlying get method for the field in the `<finder>` definition. For example, the following in the `<finder>` definition:

```
order_id = :order.orderId
```

calls the `getOrderId()` method on the order `EJBObject` and uses the result of the call in the selection criterion.

Specifying relationships between entities

Relational databases (RDBMS) permit records in one table to be associated with records in another table. The RDBMS accomplishes this using foreign keys; that is, a record in one table maintains a field (or column) that is a foreign key or reference to (usually) the primary key of a related record in another table. You can map these same references among entity beans.

For the CMP engine to map references among entity beans, you use an `<ejb-link>` entry in the deployment descriptor. The `<ejb-link>` maps field names to their corresponding entities. The CMP engine uses this information in the deployment descriptor to locate the field's associated entity. (Refer to the pigs example for an illustration of the `<ejb-link>` entry.)

Any container-managed persistence field can correspond to a foreign key field in the corresponding table. When you look at the entity bean code, these foreign key CMP fields appear as object references.

For example, suppose you have two database tables, an address table and a country table. The address table contains a reference to the country table. The SQL create statements for these tables might look as shown below.

```
create table address (
   addr_id         number(10),
   addr_street1    varchar2(40),
   addr_street2    varchar(40),
   addr_city       varchar(30),
   addr_state       varchar(20),
   addr_zip         varchar(10),
   addr_co_id       number(4)         * foreign key *
);
create table country (
   co_id            number(4),
   co_name          varchar2(50),
   co_exchange      number(8, 2),
   co_currency      varchar2(10)
);
```

Note that the address table contains the field addr_co_id, which is a foreign key referencing the country table's primary key field, co_id.

There are two classes that represent the entities which correspond to these tables, the Address and Country classes. The Address class contains a direct pointer, country, to the Country entity. This direct pointer reference is an EJBObject reference; it is not a direct Java reference to the implementation bean.

Now examine the code for both classes:

```
//Address Class
public class Address extends EntityBean {
   public int id;
   public String street1;
   public String street1;
   public String city;
   public String state;
   public String zip;
   public Country country; // this is a direct pointer to the Country
}
//Country Class
public class Country extends EntityBean {
   public int id;
   public String name;
   public int exchange;
   public String currency;
}
```

In order for the Container to resolve the reference from the Address class to the Country class, you must specify information about the Country class in the deployment descriptor. Using the <ejb-link> entry in the deployment descriptor, you instruct the Container to link the reference to the field Address.country to the JNDI name for the home object, CountryHome. (Look at the pigs example for a more detailed explanation.) The container optimizes this cross-entity reference; because of the optimization, using the cross reference is as fast as storing the value of the foreign key.

However, there are two important differences between using a cross reference and storing the foreign key value:

- When you use a cross reference pointer to another entity, you do not have to call the other entity's home object `findByPrimaryKey()` method to retrieve the corresponding object entity. Using the above example as an illustration, the `Address.country` pointer to the `Country` object lets you retrieve the country object directly. You do not have to call `CountryHome.findByPrimaryKey(address.country)` to get the `Country` object that corresponds to the country `id`.
- When you use a cross reference pointer, the state of the referenced entity is only loaded when you actually use it. It is not automatically loaded when the entity containing the pointer is loaded. That is, merely loading in an `Address` object does not actually load in a `Country` object. You can think of the `Address.country` field as a "lazy" reference, though when the underlying object is actually used does a "lazy" reference load in its corresponding state. (Note that this "lazy" behavior is a part of the EJB model.) This facet of the EJB model results in the decoupling of the life cycle of `Address.country` from the life cycle of the `Address` bean instance itself. According to the model, `Address.country` is a normal entity EJBObject reference; thus, the state of `Address.country` is only loaded when and if it is used. The Container follows the EJB model and controls the state of `AddressBean.country` as it does with any other EJBObject.

Container-managed field names

The Borland Container has changed the container-managed persistent field names so that they are more Java friendly. SQL column names often prepend a shortened form of the table name, followed by an underscore, to each column name. For example, in the `address` table, there is a column for the city called `addr_city`. The full reference to this column is `address.addr_city`. With the Borland Container, this maps to the Java field `Address.city`, rather than the more redundant and more awkward `Address.addr_city`.

You can achieve this Java-friendly column-to-field-name mapping using the deployment descriptor. While this section shows you how to manually edit the deployment descriptor, it is best to use the Deployment Descriptor Editor GUI to accomplish this. See "Using the Deployment Descriptor Editor" in the *Management Console User's Guide* for instructions on using the GUI screens.

Should you choose to manually edit the deployment descriptor, use the `<env-entry-name>`, `<env-entry-type>`, and `<env-entry-value>` subtags within the `<env-entry>` tag. Place the more friendly Java field name in the `<env-entry-name>` tag, noting that it is referencing a JDBC column. Put the type of the field in the `<env-entry-type>` tag. Lastly, place the actual SQL column name in the `<env-entry-value>` tag. The following deployment descriptor code segment illustrates this:

```
<env-entry>
   <env-entry-name>ejb.cmp.jdbc.column:city</env-entry-name>
   <env-entry-type>String</env-entry-type>
   <env-entry-value>addr_city</env-entry-value>
</env-entry>
```

Setting Properties

Most properties for Enterprise JavaBeans can be set in their deployment descriptors. The Borland Deployment Descriptor Editor (DDEditor) also allows you to set properties and edit descriptor files. Use of the Deployment Descriptor Editor is described in the Borland AppServer *Management Console User's Guide*. Use properties in the deployment descriptor to specify information about the entity bean's interfaces, transaction attributes, and so forth, plus information that is unique to an entity bean. In addition to the general descriptor information for entity beans, here are also three sets of properties that can be set to customize CMP implementations, entity properties, table properties, and column properties. Entity properties can be set either by using the EJB Designer or in the XML directly.

Using the Deployment Descriptor Editor

You can use the Deployment Descriptor Editor, which is part of the Borland AppServer to set up all of the container-managed persistence information. You should refer to the *Management Console User's Guide* for complete information on the use of the Deployment Descriptor Editor and other related tools.

J2EE 1.2 Entity Bean using BMP or CMP 1.1

Descriptor Element	Navigation Tree Node/Panel Name	DDEditor Tab
Entity Bean name	Bean	General
Entity Bean class	Bean	General
Home Interface	Bean	General
Remote Interface	Bean	General
Home JNDI Name	Bean	General
Persistence Type (CMP or BMP)	Bean	General
Primary Key Class	Bean	General
Reentrancy	Bean	General
Icons	Bean	General
Environment Entries	Bean	Environment
EJB References to other Beans	Bean	EJB References
EJB Links	Bean	EJB References
Resource References to data objects/connection factories	Bean	Resource References
Resource Reference type	Bean	Resource References
Resource Reference Authentication Type	Bean	Resource References
Security Role References	Bean	Security Role References
Entity Properties	Bean	Properties
Container Transactions	Bean:Container Transactions	Container Transactions
Transactional Method	Bean:Container Transactions	Container Transactions
Transactional Method Interface	Bean:Container Transactions	Container Transactions
Transactional Attribute	Bean:Container Transactions	Container Transactions
Method Permissions	Bean:Method Permissions	Method Permissions
CMP Description	Bean:CMP1.1	CMP 1.1
CMP Tables	Bean:CMP1.1	CMP 1.1

Descriptor Element	Navigation Tree Node/Panel Name	DDEditor Tab
Container-Managed Fields Description	Bean:CMP1.1	CMP 1.1
Finders	Bean:CMP1.1	Finders
Finder Method	Bean:CMP1.1	Finders
Finder WHERE Clause	Bean:CMP1.1	Finders
Finder Load State option	Bean:CMP1.1	Finders

Container-managed data access support

For container-managed persistence, the Borland EJB Container supports all data types supported by the JDBC specification, plus some other types beyond those supported by JDBC.

The following shows the basic and complex types supported by the Borland EJB Container:

- Basic types:
  - `boolean Boolean`
  - `double Double`
  - `long Long`
  - `BigDecimal java.util.Date`
  - `byte Byte`
  - `float Float`
  - `short Short`
  - `byte[]`
  - `char Character`
  - `int Integer`
  - `String java.sql.Date`
  - `java.sql.Time java.sql.TimeStamp`
- Complex types
  - Any class implementing `java.io.Serializable`, such as Vector and Hashtable
  - Other entity bean references

Keep in mind that the Borland Container supports classes implementing the `java.io.Serializable` interface, such as Hashtable and Vector. The container supports other data types, such as Java collections or third party collections, because they also implement `java.io.Serializable`. For classes and data types that implement the `Serializable` interface, the Container merely serializes their state and stores the result into a `BLOB`. The Container does not do any “smart” mapping on these classes or types; it just stores the state in binary format. The Container's CMP engine observes the following rule: the engine serializes as a `BLOB` all types that are not one of the explicitly supported types.

In this context, the Container follows the JDBC specification: a `BLOB` is the type to which `LONGVARBINARY` maps. (For Oracle, this is `LONG RAW`.)

Using SQL keywords

The CMP engine for the Borland Container can handle all SQL keywords that comply with the SQL92 standard. However, you should keep in mind that vendors frequently add their own keywords. For example, Oracle uses the keyword `VARCHAR2`. If you want to ensure that the CMP engine can handle vendor keywords that may differ from the SQL standard, set up an environment property in the deployment descriptor that maps the CMP field name to the column name. By using this sort of environment property, you do not have to modify your code.

For example, suppose you have a CMP field called “select”. You can use the following environment property to map “select” to a column called “SLCT”, as shown below.

```
<cmp-info>
         <database-map>
            <table>Data</table>
            <column-map>
               <field-name>select</field-name>
               <column-name>SLCT</column-name>
               </column-map>
            </database-map>
</cmp-info>
```

Using null values

It is possible that your database values can contain SQL null values. If so, you must map them to fields whose Java data types are permitted to contain Java null values. Typically, you do this by using Java types instead of primitive types. Thus, you use a Java `Integer` type rather than a primitive `int` type, or a Java `Float` type rather than a primitive `float` type.

Establishing a database connection

You must specify a DataSource so that the CMP engine can open a database connection. The DataSource defines the information necessary for establishing a database connection, such as username and password. Define a DataSource and then use a `resource-ref` to refer to the DataSource in the XML deployment descriptor for the bean. The CMP engine can then use the DataSource to access the database via JDBC.

At the point in the vendor-specific XML file where you provide the `jndi` binding for the `resource-ref`, add the element

```
<cmp-resource>True</cmp-resource>
```

For cases where the entity bean declares only one `resource-ref`, you do not need to provide the above XML element. When the entity bean has only one `resource-ref`, the Borland Container knows to automatically choose that one resource as the `cmp-resource`.

Container-created tables

You can instruct the Borland EJB Container to automatically create tables for container-managed entities based on the entity's container-managed fields. Because table creation and data type mappings vary among vendors, you must specify the JDBC database dialect in the deployment descriptor to the Container. For all databases (except for JDataStore) if you specify the dialect, then the Container automatically creates tables for container-managed entities for you. The Container will not create these tables unless you specify the dialect.

However, for the JDataStore database, the Container can detect the dialect from the URL for the JDataStore database. Thus, for JDataStore, the Container will create these tables regardless of whether you explicitly specify the dialect.

The following table shows the names or values for the different dialects (case is ignored for these values):

Database Name	Dialect Value
JDataStore	jdatastore
Oracle	oracle
Sybase	sybase
MSSQLServer	mssqlserver
DB2	db2
Interbase	interbase
Informix	informix
No database	none

Mapping Java types to SQL types

When you develop an enterprise bean for an existing database, you must map the SQL data types specified in the database schema to Java programming language data types.

The Borland EJB Container follows the JDBC rules for mapping Java programming language types to SQL types. JDBC defines a set of generic SQL type identifiers that represent the most commonly used SQL types. You must use these default JDBC mapping rules when you develop an enterprise bean to model an existing database table. (These types are defined in the class `java.sql.Types`.)

The following table shows the default SQL to Java type mapping as defined by the JDBC specification.

Java type	JDBC SQL type
`boolean/Boolean`	`BIT`
`byte/Byte`	`TINYINT`
`char/Character`	`CHAR(1)`
`double/Double`	`DOUBLE`
`float/Float`	`REAL`
`int/Integer`	`INTEGER`
`long/Long`	`BIGINT`
`short/Short`	`SMALLINT`
`String`	`VARCHAR`
`java.math.BigDecimal`	`NUMERIC`
`byte[]`	`VARBINARY`
`java.sql.Date`	`DATE`
`java.sql.Time`	`TIME`
`java.sql.Timestamp`	`TIMESTAMP`
`java.util.Date`	`TIMESTAMP`
`java.io.Serializable`	`VARBINARY`

Automatic table mapping

The Borland EJB container has the capability to automatically map Java types defined in the enterprise bean code to database table types. However, while it may create these tables automatically, it does not necessarily use the most optimal mapping approach. In fact, automatically generating these mappings and tables is more of a convenience for developers.

The Borland-generated tables are not optimized for performance. Often, they overuse database resources. For example, the container maps a Java `String` field to the corresponding SQL `VARCHAR` type. However, the mapping is not sensitive to the actual length of the Java field, and so it maps all string fields to the maximum `VARCHAR` length. Thus, it might map a two-character Java `String` to a `VARCHAR(2000)` column.

In a production situation, it is preferable for database administrators (DBA) to create the tables and do the type mapping. The DBA can override the default mappings and produce a table optimized for performance and use of database resources.

While all relational databases implement SQL types, there may be significant variations in how they implement these types. Even when they support SQL types with the same semantics, they may use different names to identify these types. For example, Oracle implements a Java `boolean` as a`NUMBER(1,0)`, while Sybase implements it as a `BIT` and DB2 implements it as a `SMALLINT`.

When the Borland EJB Container creates the database tables for your enterprise beans, it automatically maps entity bean fields and database table columns. The container must know how to properly specify the SQL types so that it can correctly create the tables in each supported database. As a result, the EJB Container maps some Java types differently, depending on the database in use. The following table shows the mapping for Oracle, Sybase/MSSQL, and DB2:

Java types	Oracle	Sybase/MSSQL	DB2
boolean/Boolean	NUMBER(1,0)	BIT	SMALLINT
byte/Byte	NUMBER(3,0)	TINYINT	SMALLINT
char/Character	CHAR(1)	CHAR(1)	CHAR(1)
double/Double	NUMBER	FLOAT	FLOAT
float/Float	NUMBER	REAL	REAL
int/Integer	NUMBER(10,0)	INT	INTEGER
long/Long	NUMBER(19,0)	NUMERIC(19,0)	BIGINT
short/Short	NUMBER(5,0)	SMALLINT	SMALLINT
String	VARCHAR(2000)	TEXT	VARCHAR(2000)
java.math.BigDecimal	NUMBER(38)	DECIMAL(28,28)	DECIMAL
byte[]	LONG RAW	IMAGE	BLOB
java.sql.Date	DATE	DATETIME	DATE
java.sql.Time	DATE	DATETIME	TIME
java.sql.Timestamp	DATE	DATETIME	TIMESTAMP
java.util.Date	DATE	DATETIME	TIMESTAMP
java.io.Serializable	RAW(2000)	IMAGE	BLOB

The following table shows the Java to SQL type mapping for JDatastore, Informix, and Interbase:

Java types	JDatastore	Informix	Interbase
boolean/Boolean	BOOLEAN	SMALLINT	SMALLINT
byte/Byte	SMALLINT	SMALLINT	SMALLINT
char/Character	CHAR(1)	CHAR(1)	CHAR(1)
double/Double	DOUBLE	FLOAT	DOUBLE PRECISION
float/Float	FLOAT	SMALLFLOAT	FLOAT
int/Integer	INTEGER	INTEGER	INTEGER
long/Long	LONG	DECIMAL(19,0)	NUMBER(15,0)
short/Short	SMALLINT	SMALLINT	SMALLINT
String	VARCHAR	VARCHAR(2000)	VARCHAR(2000)
java.math.BigDecimal	NUMERIC	DECIMAL(32)	NUMBER(15,15)
byte[]	OBJECT	BYTE	BLOB
java.sql.Date	DATE	DATE	DATE
java.sql.Time	TIME	DATE	DATE
java.sql.Timestamp	TIMESTAMP	DATE	DATE
java.util.Date	TIMESTAMP	DATE	DATE
java.io.Serializable	OBJECT	BYTE	BLOB

Chapter

14

Entity Beans and Table Mapping for CMP 2.x

Here we'll examine how entity beans are deployed in the Borland AppServer (AppServer) and how persistence of entities can be managed. This is not, however, an introduction to entity beans and should not be treated as such. Rather, this document will explore the implications of using entity beans within Borland Partitions. We'll discuss descriptor information, persistence options, and other container-optimizations. Information on the Borland-specific deployment descriptors and implementations of Container-Managed Persistence (CMP) will be documented in favor of general EJB information that is generally available from the J2EE Specifications from Sun Microsystems.

Entity Beans

Entity beans represent a view of data stored in a database. Entity beans can be fine-grained entities mapping to a single table with a one-to-one correspondence between entity beans and table rows. Or, entity beans can span multiple tables and present data independent of the underlying database schema. Entity beans can have relationships with one another, can be queried for data by clients, and can be shared among different clients.

Deploying your Entity Bean to one of the AppServer Partitions requires that it be packaged as a part of a JAR. The JAR must include two descriptor files: `ejb-jar.xml` and the proprietary `ejb-borland.xml` file. The `ejb-jar.xml` descriptor is fully-documented in the J2EE 1.3 Specification. The DTD for `ejb-borland.xml` is reproduced in this document and aspects of its usage documented here. The Borland proprietary descriptor allows for the configuration of a number of properties that can be set to optimize container performance and manage the persistence of your entity beans.

Container-managed persistence and Relationships

Borland's EJB container provides tools that generate the persistence calls at the time that the entity bean is deployed; that is, when the entity bean is installed into a Partition. The tools use the deployment descriptors to determine the instance fields which must be persisted. Instead of coding the database access directly in the bean, the bean provider of a container-managed entity bean must specify in the deployment descriptor those instance fields for which the container tools must generate access calls. The container has sophisticated deployment tools capable of mapping the fields of an entity bean to its data source.

Container-managed persistence has many advantages over bean-managed persistence. It is simpler to code because the bean provider does not have to code the database access calls. Handling of persistence can also be changed without having to modify and recompile the entity bean code. The Deployer or Application Assembler can do this by modifying the deployment descriptor when deploying the entity bean. Shifting the database access and persistence handling to the container not only reduces the complexity of code in the bean, it also reduces the scope of possible errors. The bean provider can focus on debugging the business logic of the bean rather than the underlying system issues.

Borland's Persistence Manager (PM) not only persists CMP fields but also CMP relationships. The container manages bean relationships and maintains the referential integrity of these relationships. Just as you defined container-managed persistence fields in a bean's deployment descriptor, you can now define container-managed relationship fields in the deployment descriptor. The container supports relationships of various cardinalities, including one-to-one, one-to-many, and many-to-many.

Packaging Requirements

Like session beans, entity beans can expose their methods with a remote interface or with a local interface. The remote interface exposes the bean's methods across the network to other, remote components. The local interface exposes the bean's methods only to local clients; that is, clients located on the same EJB container.

Entity beans that use EJB 2.0 container-managed persistence should use the local model. That is, the entity bean's local interface extends the `EJBLocalObject` interface. The bean's local home interface extends the `EJBLocalHome` interface. You must deploy these interfaces as well as an implementation of your bean's class.

Each Entity Bean must also have corresponding entries in its JAR's deployment descriptors. The standard deployment descriptor, `ejb-jar.xml` contains essentially three different types of deployment information. These are:

1. **General Bean Information:** This corresponds to the `<enterprise-beans>` elements found in the descriptor file and is used for all three types of beans. This information also includes information on the bean's interfaces and class, security information, environmental information, and even query declarations.
2. **Relationships:** This corresponds to the `<relationships>` elements found in the descriptor file and applies to entity beans using CMP only. This is where container-managed relationships are spelled out.
3. **Assembly Information:** This corresponds to the `<assembly-descriptor>` element which explains how the beans interact with the application as a whole. Assembly information is broken down into four categories:
   - **Security Roles:** simple definitions of security roles used by the application. Any security role references you defined for your beans must also be defined here.
   - **Method Permissions:** each method of each bean can have certain rules about their execution. These are set here.

- **Container-Transactions:** this specifies the transaction attributes as per the EJB 2.0 specification for each method participating in a transaction.
- **Exclude List:** methods not to be called by anyone.

In addition, each Entity Bean also provides persistence information in the Borland-specific descriptor file, `ejb-borland.xml`. In this descriptor file, you specify information used by the Borland CMP engine and PM to persist entities in a backing store. This information includes:

- **General Bean Information:** Information about deployed Enterprise JavaBeans, including interface locations.
- **Table and Column Properties:** Information about database tables and columns used by entity beans in the JAR.
- **Security Roles:** Authorization information for the deployed Enterprise JavaBeans.

All of these can be accessed from the Deployment Descriptor Editor. You should refer to the EJB 2.0 specification for DTD information and the proper use of the descriptor files.

A note on reentrancy

By default, entity beans are not reentrant. When a call within the same transaction context arrives at the entity bean, it causes the exception `java.rmi.RemoteException` to be thrown.

You can declare an entity bean reentrant in the deployment descriptor; however, take special care in this case. The critical issue is that a Container can generally not distinguish between a (loopback) call within the same transaction and a concurrent invocation (in the same transaction context) on that same entity bean.

When the entity bean is marked reentrant, it is illegal to allow a concurrent invocation within the same transaction context on the bean instance. It is the programmer's responsibility to ensure this rule.

Container-Managed Persistence in AppServer

The AppServer's EJB Container is fully J2EE 1.3 compliant. It implements both container-managed persistence (CMP) for Enterprise JavaBeans implementing either the EJB 1.1 and/or EJB 2.0 specifications. The bean provider designs persistence schemas for their entity beans, determines the methods for accessing container-managed fields and relationships, and defines these in beans' deployment descriptors. The deployer maps this persistence schema to the database and creates any other necessary classes for the beans' maintenance.

The EJB 2.0 Specification from Sun Microsystems details the specifics for the bean and container contracts in Chapters 10 and 11. Creating the persistence schema is not in the scope of this document, but is well discussed in both the Sun specification and in the Borland JBuilder documentation at `http://info.borland.com/techpubs/jbuilder/`, the relevant parts of which are the *Developing Applications with Enterprise JavaBeans* and the *Distributed Application Developer's Guide*.

About the Persistence Manager

The Persistence Manager (PM) provides a data-access layer for reading and writing entity beans. It also provides navigation and maintenance support for relationships between entities and extensions to EJB-QL. Currently, the PM only supports data access to relational database by means of JDBC. The PM uses an optimistic concurrency approach to data access. Conflicts in resource state are resolved before transaction commit or rollback by use of verified SQL update and delete statements.

Although the PM does not manage transactions (this is the Container's responsibility), it is aware of transaction start and completion and can therefore manage entity state. The PM uses the `TxContext` class to represent the root of managed entities during transaction lifecycles. When the container manages a transaction it asks the PM for the associated `TxContext` instance. If none exists, as is the case when a new transaction has started, one is created by the PM. When a transaction is completing, the container calls the method `TxContext.beforeCompletion()` to alert the PM to verify entity state.

The PM has complete responsibility for entity data storage and the maintenance of the state of relationships between entities. Relationship editing is also managed by the PM. This simplifies interactions with the container and allows the PM to optimize its read and write operations. This approach also suppresses duplicate `find` requests by tracking returned primary keys for requested entities. Data from duplicate `find` operations can then be returned from the first load of the entity's data.

Borland CMP engine's CMP 2.x implementation

In CMP 2.x, the details of constructing finder and select methods have been pushed into the EJB 2.0 specification. Users should thoroughly inspect the specification for details on implementing their database SQL. The Borland EJB Container is fully-compliant with the EJB 2.0 specification and supports all of its features.

The implementation class for an entity bean using 2.0 container-managed persistence is different from that of a bean using 1.1 container-managed persistence. The major differences are as follows:

- The class is declared as an abstract class.
- There are no public declarations for the fields that are container-managed fields. Instead, there are abstract `get` and `set` methods for container-managed fields. These methods are abstract because the container provides their implementation. For example, rather than declaring the fields `balance` and `name`, the `CheckingAccount` class might include these get and set methods:

```
public abstract float getBalance();
public abstract void setBalance(float bal);
public abstract String getName();
public abstract void setName(String n);
```

- Container-managed relationship fields are likewise not declared as instance variables. The class instead provides abstract get and set methods for these fields, and the container provides the implementation for these methods.

Table Mapping for CMP 2.x is accomplished using the vendor-specific `ejb-borland.xml` deployment descriptor. The descriptor is a companion to the ejb-jar.xml descriptor described in the EJB 2.0 specification. Borland uses the XML tag `<cmp2-info>` as an enclosure for table mapping data as needed. Then you use the `<table-properties>` and its associated `<column-properties>` elements to specify particular information about the entity bean's implementation. Use the DTD for syntax of the XML grammar.

Optimistic Concurrency Behavior

The container uses optimistic or pessimistic concurrency to control the behavior of multiple transactions accessing the same data. The AppServer has four optimistic concurrency behaviors which are specified as Table Properties. These behaviors are:

- `SelectForUpdate`
- `SelectForUpdateNoWAIT`
- `UpdateAllFields`
- `UpdateModifiedFields`
- `VerifyModifiedFields`
- `VerifyAllFields`

The behavior exhibited by the container corresponds to the value of the `optimisticConcurrencyBehavior` Table Property.

Pessimistic Behavior

In this mode, the container will allow only one transaction at a time to access the data held by the entity bean. Other transactions seeking the same data will block until the first transaction has committed or rolled back. This is achieved by setting the `SelectForUpdate` table property and issuing a tuned SQL statement with the `FOR UPDATE` statement included. You can issue this SQL by overriding the SQL generated by the CMP engine. Other selects on the row are blocked until then. The tuned SQL generated looks like this:

```
SELECT ID, NAME FROM EMP_TABLE WHERE ID=? FOR UPDATE
```

You can also specify the `SelectForUpdateNoWAIT` table property. Doing so instructs the database again to lock the row until the current transaction is committed or rolled back. However, other selects on the row will fail (rather than blocking). The following SQL illustrates a SELECT statement for the above:

```
SELECT ID, NAME FROM EMP_TABLE WHERE ID=? FOR UPDATE NOWAIT
```

These options should be used with caution. Although it does ensure the integrity of the data, your application's performance could suffer considerably. This option will also not function if you are using the Option A cache, since the entity bean remains in memory in this mode and calls to `ejbLoad()` are not made between transactions.

Optimistic Concurrency

This mode permits the container to allow multiple transactions to operate on the same data at the same time. While this mode is superior in performance, there is the possibility that data integrity could be compromised.

The AppServer has four optimistic concurrency behaviors which are specified as Table Properties. These behaviors are:

- `SelectForUpdate`
- `SelectForUpdateNoWAIT`
- `UpdateAllFields`
- `UpdateModifiedFields`
- `VerifyModifiedFields`
- `VerifyAllFields`

SelectForUpdate Use this option for pessimistic concurrency. With this option specified, the database locks the row until the current transaction is committed or rolled back. Other selects on the row are blocked until then.

SelectForUpdate NoWAIT Use this option for pessimistic concurrency. With this option specified, the database locks the row until the current transaction is committed or rolled back. Other selects on the row will fail.

UpdateAllFields With this option specified, the container issues an update on all fields, regardless of whether or not they were modified. For example, consider a CMP entity bean with three fields, KEY, VALUE1, and VALUE2. The following update will be issued at the terminus of every transaction, regardless of whether or not the bean was modified:

```
UPDATE MyTable SET (VALUE1 = value1, VALUE2 = value2) WHERE KEY = key
```

UpdateModifiedFields This option is the default optimistic concurrency behavior. The container issues an update only on the fields that were modified in the transaction, or suppresses the update altogether if the bean was not modified. Consider the same bean from the previous example, and assume that only VALUE1 was modified in the transaction. Using `UpdateModifiedFields`, the container would issue the following update:

```
UPDATE MyTable SET (VALUE1 = value1) WHERE KEY = key
```

This option can provide a significant performance boost to your application. Very often data access is read-only. In such cases, not sending an update to the database upon every transaction saves quite a bit of processing time. Suppressing these updates also prevents your database implementation from logging them, also enhancing performance. The JDBC driver is also taxed far less, especially in large-scale EJB applications. Even for well-tuned drivers, the less work they have to perform, the better.

VerifyModifiedFields This option, when enabled, orders the CMP engine to issue a tuned update while verifying that the updated fields are consistent with their previous values. If the value has changed in between the time the transaction originally read it and the time the transaction is ready to update, the transaction will roll back. (You will need to handle these rollbacks appropriately.) Otherwise, the transaction commits. Again using the same table, the CMP engine generates the following SQL using the `VerifyModifiedFields` behavior if only VALUE1 was updated:

```
UPDATE MyTable SET (VALUE1 = value1) WHERE KEY = key AND VALUE1 = old-VALUE1
```

VerifyAllFields

This option is very similar to `VerifyModifiedFields`, except that all fields are verified. Again using the same table, the CMP engine generates the following SQL using this option:

```
UPDATE MyTable SET (VALUE1 = value1) WHERE KEY = key AND VALUE1 = old-VALUE1
AND VALUE2 =  old-VALUE2
```

Note The two verify settings can be used to replicate the `SERIALIZABLE` isolation level in the Container. Often your applications require serializable isolation semantics. However, asking the database to implement this can have a significant performance impact. Using the verify settings allows the CMP engine to implement optimistic concurrency using field-level locking. The smaller the granularity of the locking, the better the concurrency.

Persistence Schema

The Borland CMP 2.x engine can create the underlying database schema based on the structure of your entity beans and the information provided in the entity bean deployment descriptors. You don't need to provide any CMP mapping information in such cases. Simply follow the instructions for “Specifying tables and datasources,” below. Or, the CMP engine can adapt to an existing underlying database schema. Doing so, however, requires you to provide information to the CMP engine about your database schema. In such cases, see “Basic Mapping of CMP fields to columns” on page 130 as well as CASE 2 in “Specifying tables and datasources.”

Specifying tables and datasources

The minimum information required in `ejb-borland.xml` is an entity bean name and an associated datasource. A datasource is used to obtain connections to a database. Information on datasource configuration is given in Chapter 20, "Connecting to Resources with Borland AppServer: using the Definitions Archive (DAR)." There are two means of providing this information.

CASE 1: A development environment without existing database tables using either JDataStore or Cloudscape databases.

In this case, the Borland CMP engine creates tables automatically, assuming that the entity bean name is the same as the desired table name. You need only provide the bean's name and its associated datasource as a property:

```
<entity>
 <ejb-name>CustomerEJB</ejb-name>
 <property>
  <prop-name>ejb.datasource</property>
  <prop-value>serial://ds/myDatasource</prop-value>
 </property>
</entity>
```

The Borland CMP engine will automatically create tables in this datasource based on the bean's name and fields.

CASE 2: A deployment environment with (or without) existing database tables using supported databases.

In this case, you need to supply information on the tables to which the entities map. You'll provide a table name in the `<entity>` portion of the descriptor, and some properties in the `<table-properties>` portion:

```
<entity>
 <ejb-name>CustomerEJB</ejb-name>
 <cmp2-info>
  <table-name>CUSTOMER</table-name>
 </cmp2-info>
</entity>
⋮
<table-properties>
 <table-name>CUSTOMER</table-name>
  <property>
   <prop-name>datasource</prop-name>
   <prop-value>serial://ds/myDatasource</prop-value>
  </property>
</table-properties>
```

Note that the datasource property is called `datasource` when specified in the `<table-properties>` element and `ejb.datasource` when in the `<entity>` element. If you are using a database other than JDataStore or Cloudscape and would like to have the Borland CMP engine automatically create this table, add the following XML to the `<table-properties>` element:

```
⋮
<table-properties>
 <table-name>CUSTOMER</table-name>
  <property>
   <prop-name>create-tables</prop-name>
   <prop-value>True</prop-value>
  </property>
</table-properties>
```

Basic Mapping of CMP fields to columns

Basic field mapping is accomplished using the `<cmp-field>` element in the ejb-borland.xml deployment descriptor. In this element, you specify a field name and a corresponding column to which it maps. Consider the following XML for an entity bean called `LineItem`, which maps two fields, `orderNumber` and `line`, to two columns, `ORDER_NUMBER` and `LINE`:

```
<entity>
 <ejb-name>LineItem</ejb-name>
 <cmp2-info>
  <cmp-field>
   <field-name>orderNumber</field-name>
   <column-name>ORDER_NUMBER</column-name>
  </cmp-field>
  <cmp-field>
   <field-name>line</field-name>
   <column-name>LINE</column-name>
  </cmp-field>
 </cmp2-info>
</entity>
```

Mapping one field to multiple columns

Many users may employ coarse-grained entity beans that implement a Java class to represent more fine-grained data. For example, an entity bean might use an `Address` class as a field, but may need to map elements of the class (like `AddressLine1`, `AddressCity`, and so forth) to an underlying database. To do this, you use the `<cmp-field-map>` element, which defines a field map between your fine-grained class and its underlying database representation. Note that such classes must implement `java.io.Serializable` and all their data members must be public.

Consider an entity bean called `Customer` that uses the class `Address` to represent a customer's address. The `Address` class has fields for `AddressLine`, `AddressCity`, `AddressState`, and `AddressZip`. Using the following XML, we can map the class to its representation in a database with corresponding columns:

```
<entity>
 <ejb-name>Customer</ejb-name>
 ⋮
 <cmp2-info>
  <cmp-field>
   <field-name>Address</field-name>
   <cmp-field-map>
    <field-name>Address.AddressLine</field-name>
    <column-name>STREET</column-name>
   </cmp-field-map>
   <cmp-field-map>
    <field-name>Address.AddressCity</field-name>
    <column-name>CITY</column-name>
   </cmp-field-map>
   <cmp-field-map>
    <field-name>Address.AddressState</field-name>
    <column-name>STATE</column-name>
   </cmp-field-map>
   <cmp-field-map>
    <field-name>Address.AddressZip</field-name>
    <column-name>ZIP</column-name>
   </cmp-field-map>
  </cmp-field>
 </cmp2-info>
 ⋮
</entity>
```

Note that we use one `<cmp-field-map>` element per database column.

Mapping CMP fields to multiple tables

You may have an entity that contains information persisted in multiple tables. These tables must be linked by at least one column representing a foreign key in the linked table. For example, you might have a `LineItem` entity bean mapping to a table `LINE_ITEM` with a primary key `LINE` that is a foreign key in a table called `QUANTITY`. The `LineItem` entity also contains some fields from the `QUANTITY` table that correspond to `LINE` entries in `LINE_ITEM`. Here's what our `LINE_ITEM` table might look like:

LINE	ORDER_NO	ITEM	QUANTITY	COLOR	SIZE
001	XXXXXXX01	Kitty Sweater	2	red	XL

QUANTITY, COLOR, and SIZE are all values that are also stored in the QUANTITY table, shown here. Note the identical values for some of the fields. This is because the LINE_ITEM table itself stores information in the QUANTITY table, using the LineItem entity to provide composite information.

LINE	QUANTITY	COLOR	SIZE
001	2	red	XL

Again, we can describe these relationships using a combination of `<cmp-field>` elements and a `<table-ref>` element. The `<cmp-field>` elements define the fields found in `LineItem`. Since there are some fields that require information from `QUANTITY`, we'll specify that generically by using a `TABLE_NAME.COLUMN_NAME` syntax. For instance, we'd define `LINE_ITEM`'s `COLOR` column as `QUANTITY.COLOR`. Finally, we'll specify the linking column, `LINE`, that makes up our primary key/foreign key relationship. We'll do this using the `<table-ref>` element.

Now let's look at the XML. First we define the CMP fields for the `LineItem` entity bean:

```
<entity>
 <ejb-name>LineItem</ejb-name>
 ⋮
 <cmp2-info>
  <cmp-field>
   <field-name>orderNumber</field-name>
   <column-name>ORDER_NO</column-name>
  </cmp-field>
  <cmp-field>
   <field-name>line</field-name>
   <column-name>LINE</column-name>
  </cmp-field>
  <cmp-field>
   <field-name>item</field-name>
   <column-name>ITEM</column-name>
  </cmp-field>
  <cmp-field>
   <field-name>quantity</field-name>
   <column-name>QUANTITY.QUANTITY</column-name>
  </cmp-field>
  <cmp-field>
   <field-name>color</field-name>
   <column-name>QUANTITY.COLOR</column-name>
  </cmp-field>
  <cmp-field>
   <field-name>size</field-name>
   <column-name>QUANTITY.SIZE</column-name>
  </cmp-field>
```

Next, we specify the linking column between LINE_ITEM and QUANTITY by using a <table-ref> element.

```
    <table-ref>
     <left-table>
      <table-name>LINE_ITEM</table-name>
      <column-list>
       <column-name>LINE</column-name>
      </column-list>
     </left-table>
     <right-table>
      <table-name>QUANTITY</table-name>
      <column-list>
       <column-name>LINE</column-name>
      </column-list>
     </right-table>
    </table-ref>
   </cmp2-info>
  </entity>
```

Specifying relationships between tables

To specify relationships between tables, you use the <relationships> element in ejb-borland.xml. Within the <relationships> element, you define an <ejb-relationship-role> containing the role's source (an entity bean) and a <cmr-field> element containing the relationship. The descriptor then uses <table-ref> elements to specify relationships between two tables, a <left-table> and a <right-table>. You must observe the following cardinalities:

- One <ejb-relationship-role> must be defined per direction; if you have a bi-directional relationship, you must define an <ejb-relationship-role> for each bean with each referencing the other.
- Only one <table-ref> element is permitted per relationship.

Within the <left-table> and <right-table> elements, you specify a column list that contains the column names to be linked together. The column list corresponds to the <column-list> element in the descriptor. The XML is:

```
<!ELEMENT column-list (column-name+)>
```

Let's look at some relationships to see how this XML is put into practice:

CASE 1: a unidirectional one-to-one relationship.

Here, we have a Customer entity bean with a primary key, CUSTOMER_NO, that is also used as a primary key for an entity called SpecialInfo, which contains special customer information stored in a separate table. We need to specify a relationship between these two entities. The Customer entity uses a field called specialInformation to map to the SpecialInfo bean. We specify two relationship roles, one for each bean and assign either to left- and/or right-table. Then we specify the name of their related column for both.

```
<relationships>
<ejb-relation>
 <ejb-relationship-role>
  <relationship-role-source>
   <ejb-name>Customer</ejb-name>
  </relationship-role-source>
  <cmr-field>
   <cmr-field-name>specialInformation</cmr-field-name>
   <table-ref>
    <left-table>
     <table-name>CUSTOMER</table-name>
```

```
      <column-list>CUSTOMER_NO</column-list>
     </left-table>
     <right-table>
      <table-name>SPECIAL_INFO</table-name>
      <column-list>CUSTOMER_NO</column-list>
     </right-table>
    </table-ref>
   </cmr-field>
  </ejb-relationship-role>
```

Next, we finish the `<ejb-relation>` entry by providing its other half, the `SpecialInfo` bean. Since this is a mono-directional relationship, we don't need to specify any table elements. We only need add the following, defining the other half of the relationship and its source:

```
   <ejb-relationship-role>
    <relationship-role-source>
     <ejb-name>SpecialInfo</ejb-name>
    </relationship-role-source>
   </ejb-relationship-role>
  </ejb-relation>
 </relationships>
```

CASE 2: a bidirectional one-to-many relationship.

Here, we have a `Customer` entity bean with a primary key, `CUSTOMER_NO`, that is also a foreign key in an `Order` entity bean. We want the Borland EJB Container to manage this relationship. The `Customer` bean uses a field called “orders” that links a customer to his orders. The `Order` bean uses a field called “customers” for linking in the reverse direction. First, we define the relationship and its source for the first direction: setting up the mapping for a `Customer`'s orders.

```
<relationships>
 <ejb-relation>
  <ejb-relationship-role>
   <relationship-role-source>
    <ejb-name>Customer</ejb-name>
   </relationship-role-source>
   <cmr-field>
    <cmr-field-name>orders</cmr-field-name>
```

Then, we add the table references to specify the relationship between the tables. We're basing this relationship on the `CUSTOMER_NO` column, which is a primary key for `Customer` and a foreign key for `Orders`:

```
    <table-ref>
     <left-table>
      <table-name>CUSTOMER</table-name>
      <column-list>
       <column-name>CUSTOMER_NO</column-name>
      </column-list>
     </left-table>
     <right-table>
      <table-name>ORDER</table-name>
      <column-list>
       <column-name>CUSTOMER_NO</column-name>
      </column-list>
     </right-table>
     </table-ref>
    </cmr-field>
   </ejb-relationship-role>
```

We're not quite done with our relationship, though. Now, we need to complete it by specifying the relationship role for the other direction:

```
<ejb-relationship-role>
 <relationship-role-source>
  <ejb-name>Customer</ejb-name>
 </relationship-role-source>
 <cmr-field>
  <cmr-field-name>customers</cmr-field-name>
  <table-ref>
   <left-table>
    <table-name>ORDER</table-name>
    <column-list>
     <column-name>CUSTOMER_NO</column-name>
    </column-list>
   </left-table>
   <right-table>
    <table-name>CUSTOMER</table-name>
    <column-list>
     <column-name>CUSTOMER_NO</column-name>
    </column-list>
   </right-table>
  </table-ref>
 </cmr-field>
</ejb-relationship-role>
</ejb-relation>
⋮
</relationships>
```

CASE 3: a many-to-many relationship.

If you define a many-to-many relationship, you must also have the CMP engine create a cross-table which models a relationship between the left table and the right table. Do this using the `<cross-table>` element, whose XML is:

```
<!ELEMENT cross-table (table-name, column-list, column-list)>
```

You may name this cross-table whatever you like using the `<table-name>` element. The two `<column-list>` elements correspond to columns in the left and right tables whose relationship you wish to model. For example, consider two tables, `EMPLOYEE` and `PROJECT`, which have a many-to-many relationship. An employee can be a part of multiple projects, and projects have multiple employees. The `EMPLOYEE` table has three elements, an employee number (`EMP_NO`), a last name (`LAST_NAME`), and a project ID number (`PROJ_ID`). The `PROJECT` table contains columns for the project ID number (`PROJ_ID`), the project name (`PROJ_NAME`), and assigned employees by number (`EMP_NO`).

To model the relationship between these two tables, a cross-table must be created.. For example, to create a cross-table that shows employee names and the names of the projects on which they are working, the <table-ref> element would look like the following:

```
<table-ref>
    <left-table>
       <table-name>EMPLOYEE</table-name>
          <column-list>
                <column-name>EMP_NO</column-name>
           <column-name>LAST_NAME</column-name>
               <column-name>PROJ_ID</column-name>
           </column-list>
    </left-table>
    <cross-table>
        <table-name>EMPLOYEE_PROJECTS</table-name>
```

```
            <column-list>
                 <column-name>EMP_NAME</column-name>
               <column-name>PROJ_ID</column-name>
            </column-list>
            <column-list>
             <column-name>PROJ_ID</column-name>
                <column-name>PROJ_NAME</column-name>
            </column-list>
        </cross-table>
        <right-table>
            <table-name>PROJECT</table-name>
               <column-list>
                   <column-name>PROJ_ID</column-name>
                <column-name>PROJ_NAME</column-name>
                <column-name>EMP_NO</column-name>
               </column-list>
        </right-table>
    </table-ref>
```

Since these are “secondary tables” and therefore have no primary keys, the `PROJ_ID` column appears in both column lists. This could also be the common column `EMP_NO`, depending upon how you wish to model the data.

Using cascade delete and database cascade delete

Use `<cascade-delete>` when you want to remove entity bean objects. When cascade delete is specified for an object, the container automatically deletes all of that object's dependent objects. For example you may have a Customer bean which has a one-to-many, uni-directional relationship to an Address bean. Because an address instance must be associated to a customer, the container automatically deletes all addresses related to the customer when you delete the customer.

To specify cascade delete, use the <cascade-delete> element in the `ejb-jar.xml` file as follows:

```
<ejb-relation>
 <ejb-relation-name>Customer-Account</ejb-relation-name>
 <ejb-relationship-role>
  <ejb-relationship-role-name>Account-Has-Customer
  </ejb-relationship-role-name>
  <multiplicity>one</multiplicity>
  <cascade-delete/>
 </ejb-relationship-role>
</ejb-relation>
```

Database cascade delete support

AppServer supports the database cascade delete feature, which allows an application to take advantage of a database's built in cascade delete functionality. This reduces the number of SQL operations sent to the database by the container, therefore improving performance.

To use database cascade delete, the tables corresponding to the entity beans have to be created with the appropriate table constraints on the respective database. For example, if you are using cascade delete in EJB 2.0 entity beans on `Order` and `LineItem` entity beans, the tables have to be created as follows:

```
create table ORDER_TABLE (ORDER_NUMBER integer, LAST_NAME varchar(20),
FIRST_NAME varchar(20), ADDRESS varchar(48));
create table LINE_ITEM_TABLE (LINE integer, ITEM varchar(100), QUANTITY
numeric, ORDER_NUMBER integer CONSTRAINT fk_order_number REFERENCES
ORDER_TABLE(ORDER_NUMBER) ON DELETE CASCADE);
```

The <cascade-delete-db> element in the ejb-borland.xml file specifies that a cascade delete operation will use the cascade delete functionality of the database. By default this feature is turned off.

Note If you specify the <cascade-delete-db> element in the ejb-borland.xml file, you must specify <cascade-delete> in ejb-jar.xml.

The XML for <cascade-delete-db> in the ejb-borland.xml is shown in the following relationship:

```
<relationships>
        <!--
        ONE-TO-MANY: Order LineItem
        -->
        <ejb-relation>
            <ejb-relationship-role>
                <relationship-role-source>
                    <ejb-name>OrderEJB</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>lineItems</cmr-field-name>
                    <table-ref>
                        <left-table>
                            <table-name>ORDER_TABLE</table-name>
                            <column-list>
                                <column-name>ORDER_NUMBER</column-name>
                            </column-list>
                        </left-table>
                        <right-table>
                            <table-name>LINE_ITEM_TABLE</table-name>
                            <column-list>
                                <column-name>ORDER_NUMBER</column-name>
                            </column-list>
                        </right-table>
                    </table-ref>
                </cmr-field>
            </ejb-relationship-role>
            <ejb-relationship-role>
                <relationship-role-source>
                    <ejb-name>LineItemEJB</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>order</cmr-field-name>
                    <table-ref>
                        <left-table>
                            <table-name>LINE_ITEM_TABLE</table-name>
                            <column-list>
                                <column-name>ORDER_NUMBER</column-name>
                            </column-list>
                        </left-table>
                        <right-table>
                            <table-name>ORDER_TABLE</table-name>
                            <column-list>
                                <column-name>ORDER_NUMBER</column-name>
                            </column-list>
                        <right-table>
                    </table-ref>
                </cmr-field>
            </ejb-relationship-role>
      <cascade-delete-db />
        </ejb-relation>
    </relationships>
```

Chapter 15

Using Borland AppServer Properties for CMP 2.x

Setting Properties

Most properties for Enterprise JavaBeans can be set in their deployment descriptors. The Borland Deployment Descriptor Editor (DDEditor) also allows you to set properties and edit descriptor files. Use of the Deployment Descriptor Editor is described in the Borland Management Console *User's Guide*. For more information, see “Using the Deployment Descriptor Editor” on page 137. Use properties in the deployment descriptor to specify information about the entity bean's interfaces, transaction attributes, and so forth, plus information that is unique to an entity bean. In addition to the general descriptor information for entity beans, here are also three sets of properties that can be set to customize CMP implementations, entity properties, table properties, and column properties. Entity properties can be set either by using the Deployment Descriptor Editor or in the XML directly.

Using the Deployment Descriptor Editor

You can use the Deployment Descriptor Editor, which is part of the Borland AppServer (AppServer), to set up all of the container-managed persistence information. The following table shows descriptor information and where in the Deployment Descriptor Editor that information can be entered.

For complete information on the use of the Deployment Descriptor Editor and other related tools, see “Using the Deployment Descriptor Editor” in *Management Console User's Guide*.

The EJB Designer

CMP 2.x properties are set using the EJB Designer. For more information about the EJB Designer, see “The EJB Designer” in the *Borland Management Console User's Guide*.

J2EE 1.3 and 1.4 Entity Bean

Descriptor Element	Navigation Tree Node/Panel Name	DDEditor Tab
Entity Bean name	Bean	General
Entity Bean class	Bean	General
Home Interface	Bean	General
Remote Interface	Bean	General
Local Home Interface	Bean	General
Local Interface	Bean	General
Home JNDI Name	Bean	General
Local Home JNDI Name	Bean	General
Persistence Type (CMP or BMP)	Bean	General
CMP Version	Bean	General
Primary Key Class	Bean	General
Reentrancy	Bean	General
Icons	Bean	General
Environment Entries	Bean	Environment
EJB References to other Beans	Bean	EJB References
EJB Links	Bean	EJB References
Resource References to data objects/connection factories	Bean	Resource References
Resource Reference type	Bean	Resource References
Resource Reference Authentication Type	Bean	Resource References
Security Role References	Bean	Security Role References
Entity Properties	Bean	Properties
Security Identity	Bean	Security Identity
EJB Local References to beans in the name JAR	Bean	EJB Local References
EJB Local Links	Bean	EJB Local References
Resource Environmental References for JMS	Bean	Resource Env Refs
Container Transactions	Bean:Container Transactions	Container Transactions
Transactional Method	Bean:Container Transactions	Container Transactions
Transactional Method Interface	Bean:Container Transactions	Container Transactions
Transactional Attribute	Bean:Container Transactions	Container Transactions
Method Permissions	Bean:Method Permissions	Method Permissions
Entity, Table, and Column Properties	JAR	EJB Designer (see below)

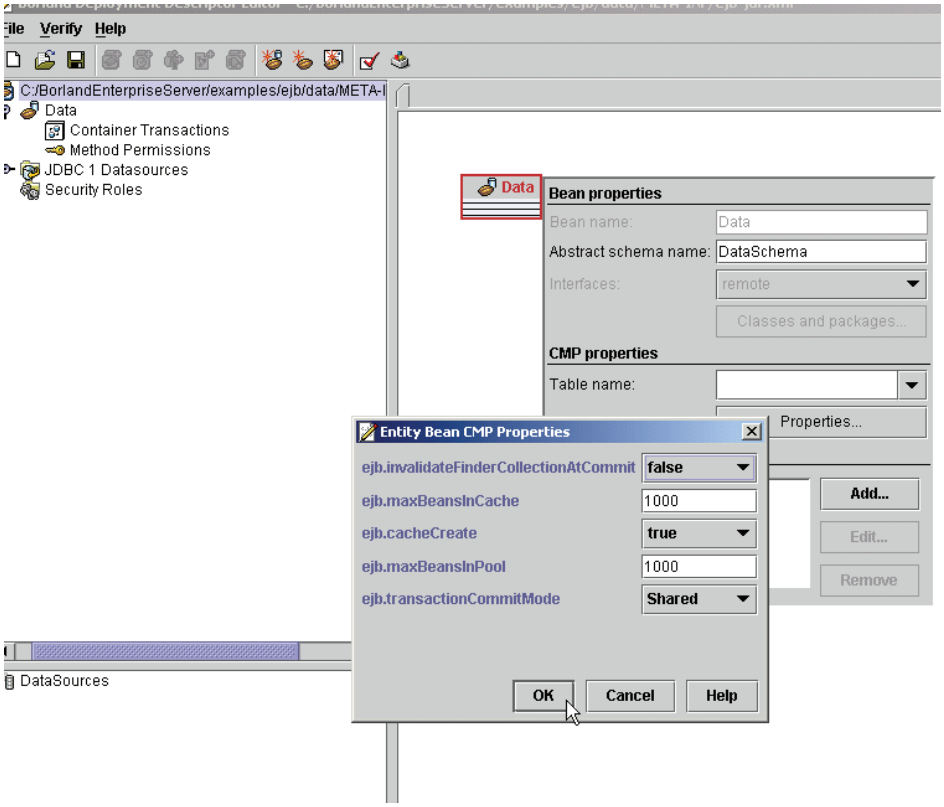
Setting CMP 2.x Properties

The AppServer uses the EJB Designer, a component of the Deployment Descriptor Editor, to set CMP 2.x properties. The EJB Designer is fully-documented in “The EJB Designer” in the *Borland Management Console User's Guide*..

Editing Entity properties

To edit Entity properties using the EJB Designer:

1. Start the DDEditor and open the deployment descriptor for the JAR containing your entity beans.
2. Select the top-level object in the DDEditor's Navigation Pane. In the Properties Pane you will see three tabs—General, XML, and EJB Designer.
3. Choose the EJB Designer Tab and left-click on any of the bean representations that appear. Click the Properties button. The Entity Beans Properties window appears.
4. Edit the properties you desire and click OK. The properties themselves are discussed below.



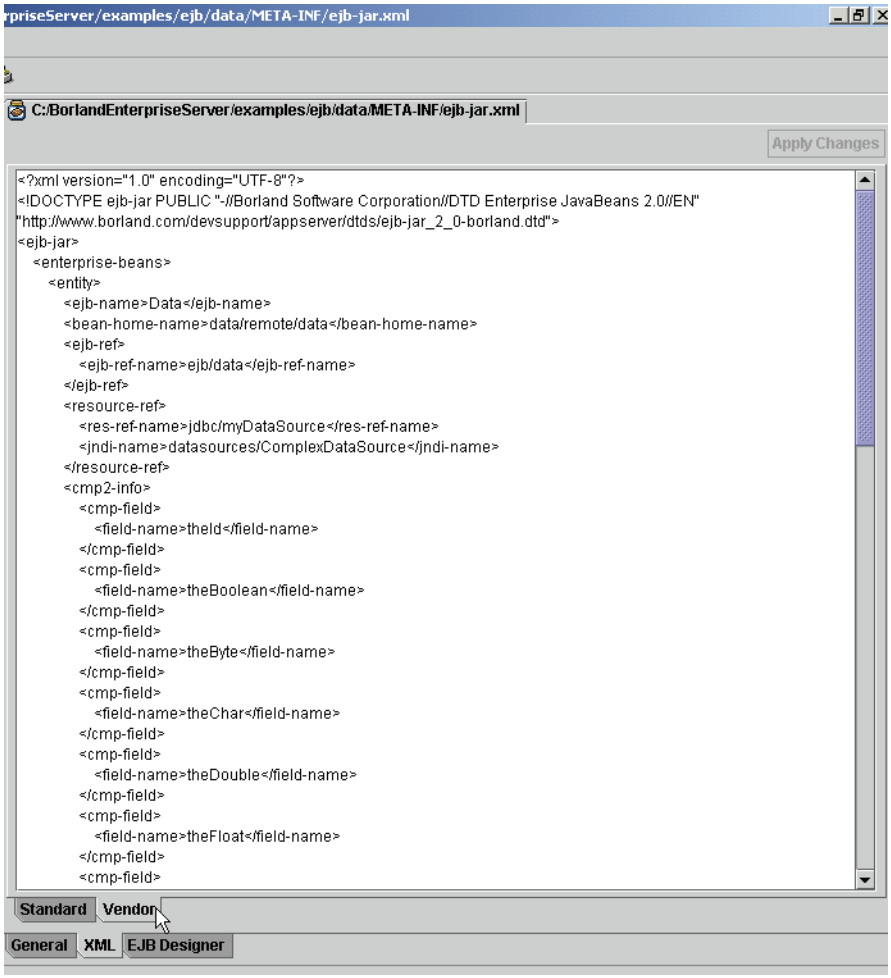
Editing Table and Column properties

Table and Column properties can only be set by editing the `ejb-borland.xml` descriptor file from the DDEditor's Vendor XML Tab, or by using the EJB Designer. To edit or add Table and Column properties:

1 Start the DDEditor and open the deployment descriptor for the JAR containing your entity beans.

2 Select the top-level object in the DDEditor's Navigation Pane. In the Properties Pane you will see three tabs: General, XML, and EJB Designer.

3 Select the XML Tab. Two additional Tabs are now available in the Properties Pane; Standard and Vendor. Choose Vendor.



4 Locate or add either the `<column-properties>` or `<table-properties>` elements and add property definitions in accordance with the borland-specific DTD (see the ejb-borland.xml). Germane entries are in bold. Descriptions of the entity, table, and column properties follow, including their data type, default values, and a property description.

Entity Properties

These properties are for CMP 1.1 and above implementations:

Property	Type	Default	Description
ejb.maxBeansInCache	lava.lang.Integer	1000	This option specifies the maximum number of beans in the cache that holds on to beans associated with primary keys, but not transactions. This is relevant for Option “A” and “B” (see ejb.transactionCommitMode below). If the cache exceeds this limit, entities will be moved to the ready pool by calling ejbPassivate.
ejb.maxBeansInPool	java.lang.Integer	1000	The maximum number of beans in the ready pool. If the ready pool exceeds this limit, entities will be removed from the container by calling unsetEntityContext().
ejb.maxBeansInTransactions	lava.lang.Integer	500* (see Description)	A transaction can access any/large number of entities. This property sets an upper limit on the number of physical bean instances that EJB container will create. Irrespective of the number of database entities/rows accessed, the container will manage to complete the transaction with a smaller number of entity objects (dispatchers). The default for this is calculated as ejb.maxBeansInCache/2. If the ejb.maxBeansInCache property is not set, this translates to 500.
ejb.TransactionCommitMode	Enumerated	Shared	Indicates the disposition of an entity bean with respect to a transaction. Acceptable values are: ■ **Exclusive**: This entity has exclusive access to the particular table in the database. The state of the bean at the end of the last committed transaction can be assumed to be the state of the bean at the beginning of the next transaction. ■ **Shared**: This entity shares access to the particular table in the database. However, for performance reasons, a particular bean remains associated with a particular primary key between transactions to avoid extraneous calls to ejbActivate() and ejbPassivate() between transactions. The bean stays in the active pool. ■ **None**: This entity shares access to the particular table in the database. A particular bean does not remain associated with a particular primary key between transactions, but goes back to the ready pool after every transaction.

These properties are for CMP 2.x implementations only:

Property	Type	Default	Description
ejb.invalidateFinderCollectionAtCommit	java.lang.Boolean	False	Whether or not to optimize transaction commit by invalidating finder collections. CMP 2.x only.
ejb.cacheCreate	java.lang.Boolean	True	Whether or not to attempt to cache the insert of the entity bean until the ejbPostCreate is processed.
ejb.datasource	java.lang.String	N/A	Default JDBC datasource to use in case no table-properties have been set. CMP 2.x only.

Property	Type	Default	Description
ejb.truncateTableName	java.lang.Boolean	False	If no table name is specified, CMP2.x engine will use the EJB name as the table name. EJB names can be more than 30 characters in length. Moreover, certain databases have a restriction on the table length to be 30 characters or less. This property is used to force the table name to be truncated to be 30 characters or less. CMP 2.x only.
ejb.eagerLoad	java.lang.Boolean	False	eager-loads the entire row and keeps the data in the transactional cache. After loading, all database resources are released. Subsequent getters could get data in cache and not having to require any more database resources. CMP 2.x only.

Table Properties

The following properties apply to CMP 2.x only. If you are migrating from CMP 1.1 to CMP 2.x, you must update your CMP properties. CMP 1.1 properties were formerly of the format ejb.<property-name>, and were all specified in the <entity> portion of the deployment descriptor. With CMP 2.x, the AppServer adds Table and Column Properties, which manage persistence. Refer to these properties below to see where migration issues may appear.

Property	Type	Default	Description
datasource	java.lang.String	None	JNDI datasource name of the database for this table.
optimisticConcurrencyBehavior	java.lang.String	UpdateModifiedFields	The container uses optimistic or pessimistic concurrency to control multiple transactions (updates) that access shared tables. Acceptable values are: ■ SelectForUpdate: database locks the row until the current transaction is committed or rolled back. Other selects on the row are blocked (wait) until then. ■ SelectForUpdateNoWAIT: database locks the row until the current transaction is committed or rolled back. Other selects on the row will fail. ■ UpdateAllFields: perform an update on all of an entity's fields, regardless if they were modified or not. ■ UpdateModifiedFields: perform an update only on fields known to have been modified prior to the update being issued. ■ VerifyModifiedFields: verify the entity's modified fields against the database prior to update. ■ VerifyAllFields: verify all the entity's fields against the database prior to update regardless if they were modified or not. Pessimistic concurrency specifies the container to allow only one transaction at a time to access the entity bean. Other transactions that try to access the same data will block (wait) until the first transaction completes. This is achieved by issuing a tuned SQL with FOR UPDATE when the entity bean is loaded. To achieve pessimistic concurrency set SelectForUpdate or SelectForUpdateNoWAIT.

Property	Type	Default	Description
useGetGeneratedKeys	java.lang.Boolean	False	Whether to use the JDBC3 java.sql.Statement.getGeneratedKeys() method to populate the primary key from autoincrement/sequence SQL fields. Currently, only Borland JDataStore supports this statement.
primaryKeyGenerationListener	java.lang.String	None	Specifies a class, written by the user, that implements com.borland.ejb.pm.PrimaryKeyGenerationListener interface and generates primary keys..
dbcAccesserFactory	java.lang.String	None	A factory class that can provide accessor class implementations to get values from a java.sql.ResultSet, and set values for a java.sql.PreparedStatement.
getPrimaryKeyBeforeInsertSql	java.lang.String	None	SQL statement to execute before inserting a row to provide primary key column names.
getPrimaryKeyAfterInsertSql	java.lang.String	None	SQL statement to execute after inserting a row to provide primary key column names.
useAlterTable	java.lang.Boolean	false	Whether or not to use the SQL ALTER statement to alter an entity's table to add columns for fields that do not have a matching column.
createTableSql	java.lang.String	None	SQL statement used to create the table if it needs to be created automatically.
create-tables	java.lang.Boolean	false	The Borland CMP engine automatically creates tables for Cloudscape and JDataStore databases—that is, in the development environment. To enable automatic table creation in other databases, you must set this flag to true.

Column Properties

Property	Type	Default	Description
ignoreOnInsert	java.lang.String	false	Specifies the column that must not be set during the execution of an INSERT statement. This property is used in conjunction with the getPrimaryKeyAfterInsertSql property.
createColumnSql	java.lang.String	None	Use this property to override the standard data-type lookup and specify the data type manually, use this property. ■ Local transactions support the javax.ejb.EJBContext methods setRollbackOnly() and getRollbackOnly(). ■ Local transactions support time-outs for database connections and transactions. ■ Local transactions are lightweight from a performance standpoint.

Property	Type	Default	Description
columnJavaType	java.lang.String	None	Java type used to create this column if the table needs to be created automatically. The acceptable values are: ■ java.lang.Boolean ■ java.lang.Byte ■ java.lang.Character ■ java.lang.Short ■ java.lang.Integer ■ java.lang.Long ■ java.lang.Float ■ java.math.BigDecimal ■ java.lang.String ■ java.sql.Time ■ java.sql.Date ■ java.sql.TimeStamp ■ java.io.Serializable This property is ignored if createColumnSql is set.

Security Properties

These security properties are specified in the <entity> portion of the deployment descriptor.

Property	Type	Default	Description
ejb.security.transportType	Enumerated	SECURE_ONLY	This property configures the Quality of Protection of a particular EJB. If set to CLEAR_ONLY, only non-secure connections are accepted from the client to this EJB. This is the default setting, if the EJB does not have any method permissions. If set to SECURE_ONLY, only secure connections are accepted form the client to this EJB. This is the default setting, if the EJB has at least one method permission set. If set to ALL, both secure and non-secure connections are accepted from the client. Setting this property controls a transport value of the ServerQoPConfig policy. See the “Security API” chapter from the Programmer's Reference for details.
ejb.security.trustInClient	java.lang.Boolean	False	This property configures the Quality of Protection of a particular EJB. If set to true, the EJB container requires the client to provide an authenticated identity. By default, the property is set to false, if there is at least one method with no method permissions set. Otherwise, it is set to true. Setting this property controls a transport value of the ServerQoPConfig policy. See the “Security API” chapter from the Programmer's Reference for details.

Chapter

16

EJB-QL and Data Access Support

EJB-QL allows you to specify queries in an object oriented query language, EJB-QL. The Borland CMP engine translates these queries into SQL queries. The Borland AppServer (AppServer) provides some extensions to the EJB-QL functionality described in the Sun Microsystems EJB 2.x Specification.

Selecting a CMP Field or Collection of CMP Fields

When only one cmp-field of an otherwise large EJB is required, you can use EJB-QL to select a single instance of collection of that cmp-field. Using EJB-QL in this way improves application performance by eliminating the need to load an entire EJB. For example, this query method selects only the balance field from the Account table:

```
<query>
   <query-method>
     <method-name>ejbSelectBalanceOfAccountLineItem</method-name>
      <method-params>
       <method-param>java.lang.Long</method-param>
   </method-params>
   </query-method>
   <result-type-mapping>Local</result-type-mapping>
   <ejb-ql>SELECT l.balance FROM Account a, IN (a.accountLineItem)
     l WHERE l.lineItemId=?1</ejb-ql>
</query>
```

The return types of the EJB-QL query method are:

- If the Java type of the cmp-field is an object type, and the query method is a single-object query method, the return type is an instance of that object type.
- If the Java type of the cmp-field is an object type and the query method returns multiple objects, a collection of instances of the object type is returned.
- If the Java type of the cmp-field is a primitive Java type, and the SELECT method is a single-object method, the return type is that primitive type.
- If the Java type of the cmp-field is a primitive Java type, and the SELECT method is for multiple objects, a collection of the wrappered Java type is returned.

Selecting a ResultSet

When more than one cmp-field is to be returned by a single query method, the return type must be of type ResultSet. This allows you to select multiple cmp-fields from the same or multiple EJBs in the same query method. You then write code to extract the desired data from the ResultSet. This feature is a Borland extension of the CMP 2.x specification.

Aggregate Functions in EJB-QL

Aggregate functions are MIN, MAX, SUM, AVG, and COUNT. For the aggregate functions MIN, MAX, SUM, and AVG, the path expression that forms the argument for the function must terminate in a cmp-field. Also, database queries for MAX, MIN, SUM, and AVG will return a null value if there are no rows corresponding to the argument to the aggregate function. If the return type is an object-type, then null is returned. If the return type is a primitive type, then the container will throw a `ObjectNotFoundException` (a sub-class of `FinderException`) if there is no value in the query result.

The path expression to the COUNT functions may terminate in either a cmp-field or cmr-field, or may be an identification variable.

For example, the following EJB-QL aggregate function terminates in a CMP field:

```
<query>
   <query-method>
     <method-name>ejbSelectMaxLineItemId</method-name>
      <method-params>
       <method-param>java.lang.String</method-param>
      </method-params>
   </query-method>
   <result-type-mapping>Local</result-type-mapping>
   <ejb-ql>SELECT MAX(l.lineItemId) FROM Account AS a, IN (a.accountLineItem) l
WHERE l.accountId=?1</ejb-ql>
</query>
```

The following restrictions must be observed for aggregate functions:

- Arguments to the SUM and AVG functions must be numeric (Integer, Byte, Long, Short, Double, Float, and BigDecimal).
- Arguments to the MAX and MIN functions must correspond to orderable cmp-field types (numeric, string, character, and dates).
- The path expression that forms the argument for the COUNT function can terminate in either a cmp-field or a cmr-field. Application performance is greatly enhanced when the COUNT function is used to determine the size of a collection of cmr-fields.

Data Type Returns for Aggregate Functions

The following table shows the data types that can be arguments for the various aggregate functions in EJB-QL selecting a single object, and what data types will be returned.

An aggregate function that selects multiple objects returns a collection of the wrappered Java data type that is returned.

Aggregate Function	Argument data type	Expected return type
MIN, MAX, SUM	java.lang.Integer	java.lang.Integer
AVG	java.lang.Integer	java.lang.Double
COUNT	java.lang.Integer	java.lang.Long

Aggregate Function	Argument data type	Expected return type
MIN, MAX, SUM	java.lang.Integer	java.lang.Integer
AVG	java.lang.Integer	java.lang.Double
COUNT	java.lang.Integer	java.lang.Long
MIN, MAX, SUM	java.lang.Byte	java.lang.Byte
AVG	java.lang.Byte	java.lang.Double
COUNT	java.lang.Byte	java.lang.Long
MIN, MAX, SUM	java.lang.Byte	java.lang.Byte
AVG	java.lang.Byte	java.lang.Double
COUNT	java.lang.Byte	java.lang.Long
MIN, MAX, SUM	java.lang.Long	java.lang.Long
AVG	java.lang.Long	java.lang.Double
COUNT	java.lang.Long	java.lang.Long
MIN, MAX, SUM	java.lang.Long	lonjava.lang.Long
AVG	java.lang.Long	java.lang.Double
COUNT	java.lang.Long	java.lang.Long
MIN, MAX, SUM	java.lang.Short	java.lang.Short
AVG	java.lang.Short	java.lang.Double
COUNT	java.lang.Short	java.lang.Long
MIN, MAX, SUM	java.lang.Short	java.lang.Short
AVG	java.lang.Short	java.lang.Double
COUNT	java.lang.Short	java.lang.Long
MIN, MAX, SUM	java.lang.Double	java.lang.Double
AVG	java.lang.Double	java.lang.Double
COUNT	java.lang.Double	java.lang.Long
MIN, MAX, SUM	java.lang.Double	java.lang.Double
AVG	java.lang.Double	java.lang.Double
COUNT	java.lang.Double	java.lang.Long
MIN, MAX, SUM	java.lang.Float	java.lang.Float
AVG	java.lang.Float	java.lang.Double
COUNT	java.lang.Float	java.lang.Long
MIN, MAX, SUM	java.lang.Float	java.lang.Float
AVG	java.lang.Float	java.lang.Double
COUNT	java.lang.Float	java.lang.Long
MIN, MAX, SUM	java.math.BigDecimal	java.math.BigDecimal
AVG	java.math.BigDecimal	java.lang.Double
COUNT	java.math.BigDecimal	java.lang.Long
MIN, MAX	java.lang.String	java.lang.String
COUNT	java.lang.String	java.lang.Long
MIN, MAX	java.util.Date	java.util.Date
COUNT	java.util.Date	java.lang.Long
MIN, MAX	java.sql.Date	java.sql.Date
COUNT	java.sql.Date	java.lang.Long
MIN, MAX	java.sql.Time	java.sql.Time
COUNT	java.sql.Time	java.lang.Long
MIN, MAX	java.sql.Timestamp	java.sql.Timestamp
COUNT	java.sql.Timestamp	java.lang.Long

Support for ORDER BY

The EJB 2.0 Specification supports three SQL clauses in EJB-QL: `SELECT`, `FROM`, and `WHERE`.

The Borland CMP engine also supports the SQL clause `ORDER BY` in the same EJB-QL statement, provided it is placed **after** the `WHERE` clause. This is done in the standard `ejb-jar.xml` deployment descriptor in the `<ejb-ql>` entity. For example, the following EJB-QL statement selects distinct objects from a Customer Bean and orders them by the LNAME field:

```
<query>
 <description></description>
 <query-method>
  <method-name>findCustomerByNumber</method-name>
  <method-params />
 <ejb-ql>SELECT Distinct Object(c) from CustomerBean c WHERE c.no >
     1000 ORDER BY c.LNAME</eql-ql>
 </query-method>
<query>
```

You can specify either ASC (ascending) or (DESC) descending in your EJB-QL as well. If you do not specify either, the results will be ordered ascending by default.

For example, consider the following table:

NAME	DEPARTMENT	SALARY	HIRE DATE
Timmy Twitfuller	Mail Room	1000	1/1/01
Sam Mackey	The Closet with the Light Out	800	1/2/02
Ralph Ossum	Coffee Room	900	1/4/01

The query:

```
SELECT OBJECT(e) FROM EMPLOYEE e ORDER BY e.HIRE_DATE
```

will produce the following result:

NAME	DEPARTMENT	SALARY	HIRE DATE
Timmy Twitfuller	Mail Room	1000	1/1/01
Ralph Ossum	Coffee Room	900	1/4/01
Sam Mackey	The Closet with the Light Out	800	1/2/02

Support for GROUP BY

The GROUP BY clause is used to group rows in the result table prior to the SELECT operation being performed. Consider the following table:

NAME	DEPARTMENT	SALARY	HIRE DATE
Mike Miller	Mail Room	1200	11/18/99
Timmy Twitfuller	Mail Room	1000	1/1/01
Buddy	Coffee Room	1000	4/13/97
Sam Mackey	The Closet with the Light Out	800	1/2/02
Todd Whitmore	The Closet with the Light Out	900	4/12/01
Ralph Ossum	Coffee Room	900	1/4/01

We can get the average salary of each department using a single query method:

```
SELECT e.DEPARTMENT, AVG(e.SALARY) FROM EMPLOYEE e GROUP BY e.DEPARTMENT
```

The results are:

DEPARTMENT	AVG(SALARY)
Coffee Room	950
Mail Room	1100
The Closet with the Light Out	850

Sub-Queries

Sub-queries are permitted as deep as the database implementation being queried allows. For example, you could use the following sub-query (in bold) specified in ejb-jar.xml. Note that the sub-query includes ORDER BY as well, and the results are to be returned in descending (DESC) order.

```
<query>
   <query-method>
     <method-name>findApStatisticsWithGreaterThanAverageValue</method-name>
     <method-params />
   </query-method>
     <ejb-ql>SELECT Object(s1) FROM ApStatistics s1 WHERE s1.averageValue >
SELECT AVG(s2.averageValue) FROM ApStatistics s2 ORDER BY s1.averageValue
DESC</ejb-ql>
</query>
```

See your database implementation documentation for details on the appropriate use of sub-queries.

Dynamic Queries

There are situations where you may need to search dynamically for data, based on variable criteria. Unfortunately EJB-QL queries do not support this scenario. Since EJB-QL queries are specified in the deployment descriptor, any changes to the queries require re-deployment of the bean. The AppServer offers a Dynamic Query feature which allows you to construct and execute EJB-QL queries dynamically and programmatically in the bean code.

Dynamic queries offer these benefits:

- allow you to create and execute new queries without having to update and deploy an EJB.
- reduce the size of the EJB's deployment descriptor file because finder queries can be dynamically created instead of statically defined in the deployment descriptors.

Dynamic queries don't need to be added to the deployment descriptor. They are declared in the bean class for dynamic `ejbSelects`, or in the local or remote home interfaces for dynamic finders.

A finder method for a dynamic query is:

```
public java.util.Collection findDynamic(java.lang.String ejbql,
    Class[] types, Object[] args)
  throws javax.ejb.FinderException

public java.util.Collection findDynamic(java.lang.String ejbql,
    Class[] types, Object[] args, java.lang.String sql)
  throws javax.ejb.FinderException
```

The `ejbSelect`s for dynamic queries are:

```
  public java.util.Collection selectDynamicLocal(java.lang.String ejbql,
Class[] types, Object[] params)
      throws javax.ejb.FinderException

public java.util.Collection selectDynamicLocal(java.lang.String ejbql, Class[]
types, Object[] params, java.lang.String sql)
      throws javax.ejb.FinderException

  public java.util.Collection selectDynamicRemote(java.lang.String ejbql,
Class[] types, Object[] params)
      throws javax.ejb.FinderException

public java.util.Collection selectDynamicRemote(java.lang.String ejbql, Class[]
types, Object[] params, java.lang.String sql)
      throws javax.ejb.FinderException

  public java.sql.ResultSet selectDynamicResultSet(java.lang.String ejbql,
Class[] types, Object[] params)
      throws javax.ejb.FinderException

public java.sql.ResultSet selectDynamicResultSet(java.lang.String ejbql,
Class[] types, Object[] params, java.lang.String sql)
      throws javax.ejb.FinderException
```

where the following applies:

- **`java.lang.String ejbql`**: this represents the actual EJB-QL syntax.
- **`Class[] types`**: this array gives the class types of the parameters to the select or finder method (it can be an empty array if there are no parameters).

- `Object[] params`: this array gives the actual values of the parameters. This is the same as the parameters argument of the regular select or finder method.

  The return type of a dynamic select or finder is always `java.util.Collection`, with the exception of the `selectDynamicResultSet`. If there is a single instance of the object or value type returned from the query, it is the first member of the collection. Dynamic queries follow the same rules as regular queries.

- `java.lang.String sql`: User specified sql. If specified, this will override the sql generated by EJB-QL.

Note There should not be any trace of the eight methods associated with dynamic queries in your deployment descriptor.

Overriding SQL generated from EJB-QL by the CMP engine

Important This feature is for advanced users only!

The Borland CMP engine generates SQL calls to your database based on the EJB-QL you enter in your deployment descriptors. Depending on your database implementation, the generated SQL may be less than optimal. You can capture the generated SQL using tools supplied by your backing-store implementation or another development tool. If the generated SQL is not optimal, you can replace it with your own. However, we offer no validation on the user SQL.

Note A problem with your SQL may generate an exception which can potentially crash the system.

You specify your own optimized SQL in the Borland proprietary deployment descriptor, `ejb-borland.xml`. The XML grammar is identical to that found in `ejb-jar.xml`, except that the `<ejb-ql>` element is replaced with a `<user-sql>` element. This proprietary element contains a SQL-92 statement (**not** an EJB-QL statement) that is used to access the database instead of the CMP engine-generated SQL.

Important The `SELECT` clause for this statement must be identical to the `SELECT` clause generated by the Borland CMP engine.

Subsequent clauses are user-optimized. The ordering of the fields in the `SELECT` clause is proprietary to the CMP engine and therefore must be preserved.

For example:

```
<entity>
 <ejb-name>EmployeeBean</ejb-name>
 ⋮
 <query>
  <query-method>
   <method-name>findWealthyEmployees</method-name>
   <method-params />
  </query-method>
  <user-sql>SELECT E.DEPT_NO, E.EMP_NO, E.FIRST_NAME, E.FULL_NAME,
          E.HIRE_DATE, E.JOB_CODE, E.JOB_COUNTRY,
          E.JOB_GRADE, E.LAST_NAME, E.PHONE_EXT, E.SALARY
       FROM EMPLOYEE E WHERE E.SALARY > 200000
  </user-sql>
 </query>
 ⋮
</entity>
```

Note The extensive `SELECT` statement reflects the type of SQL generated by the CMP engine.

When the CMP engine encounters an EJB-QL statement in the `ejb-jar.xml` deployment descriptor, it checks `ejb-borland.xml` to see if there is any user SQL provided in the same bean's descriptor.

If none is present, the CMP engine generates its own SQL and executes it.

If the ejb-borland.xml descriptor does contain a query element, it uses the SQL within the <user-sql> tags instead.

Important The <query> element in ejb-borland.xml **does not** replace the <query> element in the standard ejb-jar.xml deployment descriptor. If you want to override the CMP engine's SQL, you must provide the elements in **both** descriptors.

Container-managed data access support

For CMP, the Borland EJB Container supports all data types supported by the JDBC specification, including types beyond those supported by JDBC.

The following shows the basic and complex types supported by the Borland EJB Container:

- Basic types:
  - boolean Boolean
  - double Double
  - long Long
  - BigDecimal java.util.Date
  - byte Byte
  - float Float
  - short Short
  - byte[]
  - char Character
  - int Integer
  - String java.sql.Date
  - java.sql.Time java.sql.TimeStamp
- Complex types
  - Any class implementing java.io.Serializable, such as Vector and Hashtable
  - Other entity bean references

Note The Borland CMP engine now supports using the Long value type for dates, as well as java.sql.Date for java.util.Date.

Keep in mind that the Borland Container supports classes implementing the java.io.Serializable interface, such as Hashtable and Vector. The container supports other data types, such as Java collections or third party collections, because they also implement java.io.Serializable. For classes and data types that implement the Serializable interface, the Container merely serializes their state and stores the result into a BLOB. The Container does not do any “smart” mapping on these classes or types; it just stores the state in binary format. The Container's CMP engine observes the following rule: the engine serializes as a BLOB all types that are not one of the explicitly supported types.

Depending on your database implementation, the following data types require fetching based on column index:

Database	Data Types
Oracle	■ LONG RAW
Sybase	■ NTEXT ■ IMAGE
MS SQL	■ NTEXT ■ IMAGE

Note If you use either of the two data types BINARY (MS SQL) or RAW (Oracle) as primary keys, you must explicitly specify their size.

Support for Oracle Large Objects (LOBs)

There are two types of Large Objects (LOBs), Binary Large Objects (BLOBs) and Character Large Objects (CLOBs).

BLOBs are mapped to CMP fields with the following data types:

- byte[]
- java.io.Serializable
- java.io.InputStream

CLOBs, by virtue of being *Character* Large Objects, can only be mapped to cmp-fields with the java.lang.String data type.

By default, the Borland CMP engine does not automatically map cmp-field to LOBs. If you intend to use LOB data types, you must inform the CMP engine explicitly in the `ejb-borland.xml` deployment descriptor. You do this by setting the Column Property `createColumnSql`. For example:

```
<column-properties>
 <column-name>CLOB-column</column-name>
 <property>
  <prop-name>createColumnSql</prop-name>
  <prop-type>String></prop-type>
  <prop-value>CLOB</prop-value>
 </property>
</column-properties>

<column-properties>
 <column-name>BLOB-column</column-name>
 <property>
  <prop-name>createColumnSql</prop-name>
  <prop-type>String></prop-type>
  <prop-value>BLOB</prop-value>
 </property>
</column-properties>
```

Note The default BLOB size limit for adding and finding BLOB data from AppServer using CMP EJBs is 10,000 bytes. The default can be changed by setting the system property below:

```
-DEJBCmpMaxBlobSize=xxxxxxxx
```

This limit does not actually control the size of the BLOB. The database and its driver can also limit this size. For example, Oracle treats BLOBs that are greater than 4GB size differently than BLOB less than 4GB.

Container-created tables

You can instruct the Borland EJB Container to automatically create tables for container-managed entities based on the entity's container-managed fields by enabling the `create-tables` property. Because table creation and data type mappings vary among vendors, you must specify the JDBC database dialect in the deployment descriptor to the Container. For all databases (except for JDataStore) if you specify the dialect, then the Container automatically creates tables for container-managed entities for you if the `create-tables` property is set to `true`. The Container will not create these tables unless you specify the dialect.

The following table shows the names or values for the different dialects (case is ignored for these values):

Database Name	Dialect Value
JDataStore	jdatastore
Oracle	oracle
Sybase	sybase
MSSQLServer	mssqlserver
DB2	db2
Interbase	interbase
Informix	informix

Chapter

17

Generating Entity Bean Primary Keys

Each entity bean must have a unique primary key that is used to identify the bean instance. The primary key can be represented by a Java class, which must be a legal value type in RMI-IIOP. Therefore, it extends the `java.io.Serializable` interface. It must also provide an implementation of the `Object.equals(Object other)` and `Object.hashCode()` methods.

Normally, the primary key fields of entity beans must be set in the `ejbCreate()` method. The fields are then used to insert a new record into the database. This can be a difficult procedure, however, bloating the method, and many databases now have built-in mechanisms for providing appropriate primary key values. A more elegant means of generating primary keys is for the user to implement a separate class that generates primary keys. This class can also implement database-specific programming logic for generating primary keys.

You may either generate primary keys by hand, use a custom class, or allow the container to use the database tools to perform this for you. If you use a custom class, implement the `com.borland.ejb.pm.PrimaryKeyGenerationListener` interface, discussed below. To use the database tools, you can set properties for the CMP engine to generate primary keys depending upon the database vendor.

Generating primary keys from a user class

With enterprise beans, the primary key is represented by a Java class containing the unique data. This primary key class can be any class as long as that class is a legal value type in RMI-IIOP, meaning it extends the `java.io.Serializable` interface. It must also provide an implementation of the `Object.equals(Object other)` and `Object.hashCode()` methods, two methods which all Java classes inherit by definition.

Generating primary keys from a custom class

To generate primary keys from a custom class, you must write a class that implements the `com.borland.ejb.pm.PrimaryKeyGenerationListener` interface.

Note this is a new interface for generating primary keys. In previous versions of Borland AppServer, this class was `com.inprise.ejb.cmp.PrimaryKeyGenerator`. This interface is still supported, but Borland recommends using the newer interface when possible.

Next, you must inform the container of your intention to use your custom class to generate primary keys for your entity beans. To do this, you set a table property `primaryKeyGenerationListener` to the class name of your primary key generator.

Implementing primary key generation by the CMP engine

Primary key generation can also be implemented by the CMP engine. Borland provides four properties to support primary key generation using database specific features. These properties are:

- `getPrimaryKeyBeforeInsertSql`
- `getPrimaryKeyAfterInsertSql`
- `ignoreOnInsert`
- `useGetGeneratedKeys`

All of these properties are table properties except `ignoreOnInsert`, which is a column property.

Oracle Sequences: using getPrimaryKeyBeforeInsertSql

The property `getPrimaryKeyBeforeInsertSql` is typically used in conjunction with Oracle Sequences. The value of this property is a SQL statement used to select a primary key generated from a sequence. For example, the property could be set to:

```
SELECT MySequence.NEXTVAL FROM DUAL
```

The CMP engine would execute this SQL and then extract the appropriate value from the ResultSet. This value will then be used as the primary key when performing the subsequent `INSERT`. The extraction from the ResultSet is based on the primary key's type

SQL Server: using getPrimaryKeyAfterInsertSql and ignoreOnInsert

Two properties need to be specified for cases involving SQL Server. The `getPrimaryKeyAfterInsertSql` property specified the SQL to execute after the `INSERT` has been performed. As above, the CMP engine extracts the primary key from the ResultSet based on the primary key's type. The property `ignoreOnInsert` must also be set to the name of the identity column. The CMP engine will then know not to set that column in the `INSERT`.

JDataStore JDBC3: using useGetGeneratedKeys

Borland's JDataStore supports the new JDBC3 method `java.sql.Statement.getGeneratedKeys()`. This method is used to obtain primary key values from newly inserted rows. No additional coding is necessary, but note that this method is unsupported in other databases and is recommended for use only with Borland JDataStore. To use this method, set the boolean property `useGetGeneratedKeys` to `True`.

Automatic primary key generation using named sequence tables

A named sequence table is used to support auto primary key generation when the underlying database (such as Oracle `SEQUENCE`) and the JDBC driver (`AUTOINCREMENT` in JDBC 3.0) do not support key generation . The named sequence table allows you to specify a table that holds a key to use for primary key generation. The container uses this table to generate the keys.

The table must contain a single row with a single column

To use the name sequence table your table must have a single row with a single column that is an integer (for the sequence values). You must create a table with one column named "`SEQUENCE`" with any initial value. For example:

```
CREATE TABLE TAB_A_SEQ (SEQUENCE int);
INSERT into TAB_A_SEQ values (10);
```

In this example key generation starts from value 10.

To enable this feature, set it in `<column-properties>` in `ejb-borland.xml`:

```
<table-properties>
        <table-name>TABLE_A</table-name>
        <column-properties>
       <column-name>ID</column-name>
            <property>
                <prop-name>autoPkGenerator</prop-name>
                <prop-type>java.lang.String</prop-type>
                <prop-value>NAMEDSEQUENCETABLE</prop-value>
            </property>
       <property>
                <prop-name>namedSequenceTableName</prop-name>
                <prop-type>java.lang.String</prop-type>
                <prop-value>TAB_A_SEQ</prop-value>
            </property>
            <property>
                <prop-name>keyCacheSize</prop-name>
                <prop-type>java.lang.Integer</prop-type>
                <prop-value>2</prop-value>
            </property>
            </column-properties>
⋮
    </table-properties>
```

Note that "ID" is the primary key column, which is marked for auto Pk Generation using `NAMEDSEQUENCETABLE`. The table used is `TAB_A_SEQ`.

Note Set the `ejb.CacheCreate` property to `false` while using `getPrimaryKeyAfterInsert` or `useGetGeneratedKeys`. The container needs to know the primary key to dispatch calls to the bean instance. Therefore, it needs to know the primary key at the same time the `Create` method returns.

Key cache size

When generating the primary key, the container fetches the key from the table in the database. You can improve performance by reducing trips to the database by specifying a key cache size. To use this feature, in the `ejb-borland.xml` file, you set the `<key-cache-size>` element to specify how many primary key values the database will fetch. The container will cache the number of keys used for primary key generation when the value of the cache size is > 1.

The default value for key cache size, if not specified, is 1. Although key cache size is optional, it is recommended you specify a value > 1 to utilize performance optimization.

Note There may be gaps in the keys generated if the container is rebooted or used in a clustered mode.

Chapter 18

Transaction management

This chapter describes how to handle transactions.

Understanding transactions

Application programmers benefit from developing their applications on platforms such as Java 2 Enterprise Edition (J2EE) that support transactions. A transaction-based system simplifies application development because it frees the developer from the complex issues of failure recovery and multi-user programming. Transactions are not limited to single databases or single sites. Distributed transactions can simultaneously update multiple databases across multiple sites.

A programmer typically divides the total work of an application into a series of units. Each unit of work is a separate transaction. As the application progresses, the underlying system ensures that each unit of work, each transaction, fully completes without interference from other processes. If not, it rolls back the transaction and completely undoes whatever work the transaction had performed.

Characteristics of transactions

Typically, transactions refer to operations that access a shared resource like a database. All access to a database is performed in the context of a transaction. All transactions share the following characteristics:

- Atomicity
- Consistency
- Isolation
- Durability

These characteristics are denoted by the acronym ACID.

A transaction often consists of more than a single operation. Atomicity requires that either all or none of the operations of a transaction are performed for the transaction to be considered complete. If any of a transaction's operations cannot be performed, then none of them can be performed.

Consistency refers to resource consistency. A transaction must transition the database from one consistent state to another. The transaction must preserve the database's semantic and physical integrity.

Isolation requires that each transaction appear to be the only transaction currently manipulating the database. Other transactions can run concurrently. However, a transaction must not see the intermediate data manipulations of other transactions until and unless they successfully complete and commit their work. Because of interdependencies among updates, a transaction can get an inconsistent view of the database were it to see just a subset of another transaction's updates. Isolation protects a transaction from this sort of data inconsistency.

Transaction isolation is qualified by varying levels of concurrency permitted by the database. The higher the isolation level, the more limited the concurrency extent. The highest level of isolation occurs when all transactions can be serialized. That is, the database contents look as if each transaction ran by itself to completion before the next transaction started. However, some applications can tolerate a reduced level of isolation for a higher degree of concurrency. Typically, these applications run a greater number of concurrent transactions even if transactions are reading data that may be partially updated and perhaps inconsistent.

Lastly, durability means that updates made by committed transactions persist in the database regardless of failure conditions. Durability guarantees that committed updates remain in the database despite failures that occur after the commit operation and that databases can be recovered after a system or media failure.

Transaction support

The Borland AppServer (AppServer) supports flat transactions, but not nested transactions. Transactions are implicitly propagated. This means that the user does not have to explicitly pass the transaction context as a parameter, because the J2EE container transparently handles this for the client.

Transaction management can be performed programmatically by calling the standard JTS or JTA APIs. An alternative, and more recommended approach, when writing J2EE components such as Enterprise JavaBeans (EJBs) is to use declarative transactions where the J2EE Container transparently starts and stops transactions.

Transaction manager services

There are two transaction managers, or engines, available in AppServer:

- Transaction Manager (formerly known as Partition Transaction Service)
- OTS (formerly known as 2PC Transaction Service)

A Transaction Manager exists in each AppServer Partition. It is a Java implementation of the CORBA Transaction Service Specification. The Transaction Manager supports transaction timeouts, one-phase commit protocol and can be used in a two-phase commit protocol under special circumstances.

Use the Transaction Manager under the following conditions:

- When using one-phase commit protocol.
- When you need faster performance. Currently, only the Transaction Manager can be configured to be in-process. The transaction management APIs and other transaction components are in-process JVM calls, so it is much faster than the OTS engine.

- When using a two-phase commit protocol but do not care about transaction recovery. For example, when checking business logic during development of an Enterprise JavaBean there is no need for transaction recovery. If you use the Transaction Manager for two-phase commit, you must set the “Allow unrecoverable completion” property to true in “Properties” for the Transaction Manager as displayed under the Partition in the AppServer Management Console. Alternatively, you can set system property `EJBAllowUnrecoverableCompletion` for the partition.

The OTS engine exists in a separate address space. It provides a complete solution for distributed transactional CORBA applications. Implemented on top of the VisiBroker ORB, the OTS engine simplifies the complexity of distributed transactions by providing an essential set of services—including a transaction service, recovery and logging, integration with databases, and administration facilities—within one, integrated architecture.

Distributed transactions and two-phase commit

The Borland EJB Container supports distributed transactions. Distributed transactions are those transactions that cross systems, platforms, and Java Virtual Machines (JVMs).

Transactions that manipulate data across multiple resources use a two-phase commit process. This process ensures that the transaction correctly updates all resources involved in the transaction. If it cannot update all resources, then it updates none of the resources.

Note Although support is provided by AppServer for two-phase commit transactions, they are inherently expensive due to number of remote procedure calls (RPCs) and should be used only when needed. See “When to use two-phase commit transactions” on page 162.

There are two steps to a two-phase commit. The first step is the preparation phase. In this phase the transaction service requests that each resource involved in the transaction readies its updates and signal to the transaction service whether it can commit the updates. The second step is the commit phase. The transaction service initiates the actual resource updates only when all resources have signaled that they can complete the update process. Should any resource signal they cannot perform updates, the transaction service instructs all other resources to rollback all updates involved in the transaction.

The Transaction Manager and OTS engine support both heterogeneous distributed (two-phase commit) transactions and two-phase commit for homogeneous resources.

By default, the Transaction Manager does not allow multiple resources to participate in a global transaction, but it can be configured to allow multiple resource participation through its support for unrecoverable transaction completion. This can be enabled on the Transaction Manager by setting either “Allow unrecoverable completion” option from the Management Console (right-click the Transaction Manager and select “Properties”), or the Partition system property `EJBAllowUnrecoverableCompletion`. When unrecoverable transaction completion is enabled, the container makes a one-phase commit call on each participating resource during the transaction commit process. Care must taken when enabling unrecoverable transaction completion; as the name suggests, no recovery is available when a failure occurs prior to transaction completion, which may lead to inconsistent states in participating resources.

To support heterogeneous two-phase commit transactions, the OTS engine must integrate with XA support in the underlying resources. With availability of XA-enabled JDBC drivers from DBMS vendors and JMS support provided by message service providers, the EJB container and OTS engine allow multiple resources to participate in a single transaction.

Two-phase commit for homogeneous databases requires some configuration of the DBMS servers. While the container controls the commit to the first database, the DBMS server controls the commits to the subsequent databases using the DBMS's built-in transaction coordinator. For more information, see your vendor's manual for the DBMS server.

When to use two-phase commit transactions

One of the basic principals of building high performance distributed applications is to limit the number of remote procedure calls (RPCs). The following explains typical situations; when and when not to use two-phase commit transactions. Avoiding a two-phase commit transaction when it is not needed, therefore avoiding unnecessary RPCs involving JTA XAResource objects and the OTS engine, greatly improves your application's performance.

Using multiple JDBC connections for access to multiple database resources from a single vendor in the same transaction

In scenarios involving multiple databases from a single vendor, it is often possible to avoid using two-phase commit. You can access one database and use it to access the second database by tunneling access through the connection to the first database. Oracle and other DBMSs provide this capability. In this case the AppServer Partition can be configured with only one JDBC connection to the “fronting” database. Access to the “backing” database is tunneled through the first JDBC connection.

Using multiple JDBC connections to the same database resource in the same transaction

When multiple JDBC connections to the same database are obtained and used by distributed participants within a single transaction, a two-phase commit can be avoided. The JDBC connections, as expected, need to be obtained from a XA datasource. But, rather than performing a two-phase commit, a one-phase commit can be used to complete the transaction since only a single resource is involved. This is achieved by using the Transaction Manager rather than the OTS engine. An alternative is to collocate all EJBs involved in the transaction, rather than having them deployed in distributed Partitions. In this case, a non-XA datasource is used and no two-phase commit is required.

Using multiple disparate resources in a single transaction

In this case there is a need for a two-phase commit transaction. This situation arises when, for example, you are running a single transaction against both Oracle and Sybase, or if you have a transaction that includes access to an Oracle database and a JMS provider, such as MQSeries. In the latter case, the transaction is coordinated using JTA XAResource object, obtained from Oracle via JDBC and MQSeries via JMS, and enables both resources to participate in the two-phase commit transaction completion. It is worth noting that two-phase commit capabilities (provided by the OTS engine), are only needed when a single transaction involves access to multiple incompatible resources.

Note In order to utilize the OTS engine as the default transaction service, the Transaction Manager must be stopped first.

EJBs and 2PC transactions

With the introduction of messaging in the J2EE platform, a number of common scenarios now exist involving access to multiple resources from EJBs in a single transaction . As we know, when more than one resource is involved in a transaction, the OTS engine is needed to reliably complete the transaction using the two-phase commit protocol. Sample scenarios include:

- A session bean accesses two types of entity beans in a transaction where each are persisted in a different database.
- A session bean accesses an entity bean and in the same transaction does some messaging work, such as sending a message to a JMS queue.
- In the `onMessage` method of a message-driven bean, access entity beans on message delivery.

In each of the above examples, two heterogeneous resources need to be accessed from within a session bean or a message-driven bean as part of a single transaction. These EJBs have the `REQUIRED` transaction attribute defined and need access to the OTS engine. However, if the OTS engine is running, then all modules deployed to that Partition are able to discover it and can attempt to use it. The OTS engine will perform a one-phase commit when only one resource is registered in a transaction, but suffers the extra RMI overhead since it is an external process. Ideally, the in-process Transaction Manager should be used for EJBs not involved in a two-phase commit transaction. To better utilize the transaction services available in AppServer, a bean-level property, `ejb.transactionManagerInstanceName` may be specified for EJBs that require 2PC transaction completion. This property provides the name of the OTS engine to be used by the EJB container doing transaction demarcation on any of the methods for the relevant bean. Both the Transaction Manager and the OTS engine may be available for all EJBs but only those that do not have `ejb.transactionManagerInstanceName` specified will discover the Transaction Manager.

This property can be commonly used for session or message-driven beans since transactions are usually demarcated in a session bean facade or the `onMessage` method of a message-driven bean.

To set the `ejb.transactionManagerInstanceName` property use the Management Console. Navigate to your deployed EJB module, right-click on it and select “DDEditor”. In the DDEditor select the required bean from the Navigation Pane. Select the “Properties” tab and add the `ejb.transactionManagerInstanceName` property. Define the property as a `String` and specify a unique name value such as “MyTwoPhaseEngine”.

Next, you must modify the OTS engine factory name with the `ejb.transactionManagerInstanceName` value. In the Management Console, select the OTS engine from the “corbaSample” configuration, identified as the “OTS engine” managed object type. Right-click and select “Properties” from the drop-down menu. In the Properties dialog choose the Settings tab and modify the value for “Factory Name”. Click OK, and restart the service. The OTS engine may also be started from the command line, independent of an AppServer server. The factory name can be provided using property `vbroker.ots.name` as follows:

```
prompt> ots -Dvbroker.ots.name=<MyTwoPhaseEngine>
```

The EJB will now use the OTS engine named “MyTwoPhaseEngine”. As mentioned, the Partition may be hosting several J2EE modules, but only those beans that have `ejb.transactionManagerInstanceName` set go to the (non-default) OTS engine. Other beans in the Partition that require method invocation in a transaction, but do not require 2PC, always find the Transaction Manager due to local service affinity.

Following is a deployment configuration usage example. Displayed below is an extract from deployment descriptor `ejb-borland.xml` packaged with the deployed EJB module and viewable in the DDEditor. The `ejb.transactionManagerInstanceName` property is set for Session bean "`OrderSesEJB`" where **`OrderSesEJB`** takes orders from customers, creates an order in the database and sends messages to the manufacturers for making parts.

```
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>OrderSesEJB</ejb-name>
            <bean-home-name>OrderSes</bean-home-name>
            <bean-local-home-name />
            <ejb-local-ref>
                <ejb-ref-name>ejb/OrderEntLocal</ejb-ref-name>
                <jndi-name>OrderEntLocal</jndi-name>
            </ejb-local-ref>
            <ejb-local-ref>
                <ejb-ref-name>ejb/ItemEntLocal</ejb-ref-name>
            </ejb-local-ref>
            <resource-ref>
                <res-ref-name>jms/QueueConnectionFactory</res-ref-name>
                <jndi-name>QueueConnectionFactory</jndi-name>
            </resource-ref>
            <resource-env-ref>
                <resource-env-ref-name>jms/OrderQueue</resource-env-ref-name>
                <jndi-name>OrderQueue</jndi-name>
            </resource-env-ref>
            <property>
                <prop-name>ejb.transactionManagerInstanceName</prop-name>
                <prop-type>String</prop-type>
                <prop-value>TwoPhaseEngine</prop-value>
            </property>
        </session>

<ejb-jar>
```

Example runtime scenarios

The following diagrams show configurations where the standard Transaction Manager and the OTS engine co-exist. The deployment configuration is done in a manner in which the beans participating in 2PC transactions have their transaction management done by the OTS engine, named "`TwoPhaseEngine`", and those that don't need 2PC transactions use the default in- process Transaction Manager.

The example archive used is `complex.ear`, in an AppServer Partition. It has three beans:

- **`OrderSesEJB`:** takes orders from customers, creates an order in the database, and sends messages to the manufacturers for making parts.
- **`UserSesEJB`:** creates new users in the company database. Only accesses a single database, therefore only needs to access a 1PC engine (Transaction Manager).
- **`OrderCompletionMDB`:** receives a notification from the manufacturer about the part delivery, and also updates the database using entity beans.

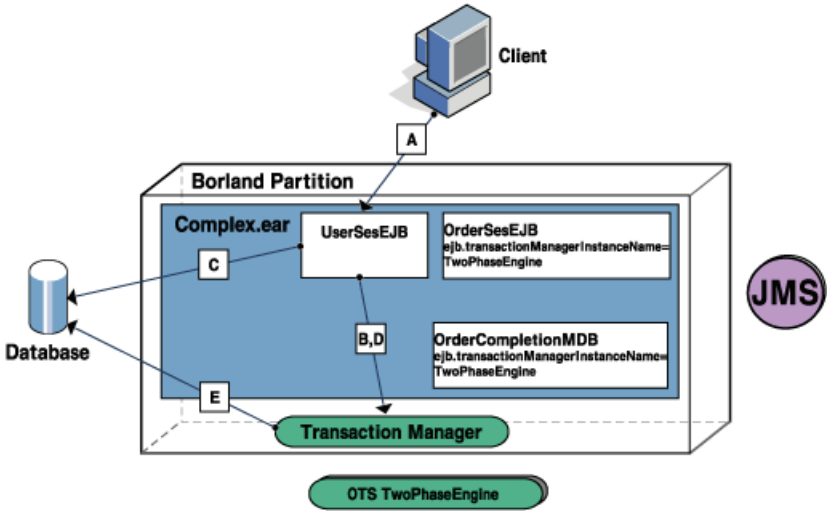
To configure this example deployment scenario:

1. Using the DDEditor, add the `ejb.transactionManagerInstance` property to the beans `OrderSesEJB` and `OrderCompletionMDB`. Refer to the above XML extract for this example.
2. Next, using the Management Console, start the OTS engine with factory name set as "TwoPhaseEngine".
3. Keep the local Transaction Manager enabled.

The following diagrams show example interactions between the client, the AppServer Partition, and how the AppServer Partition locates the right transaction service based on the above configuration. All of the beans are assumed to have container-managed transactions.

Example 1PC usage

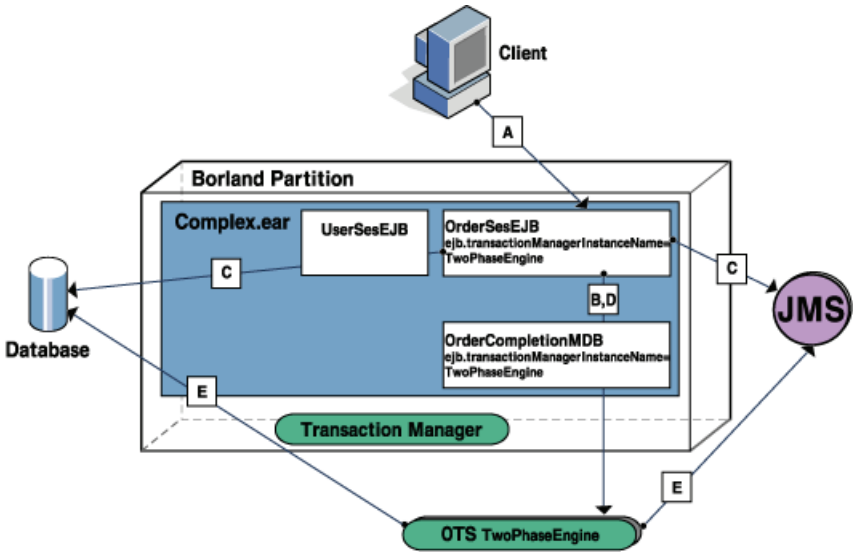
1. The client calls a method of `UserSesEJB`. This is an implementation of the method that creates users in the database.
2. Before the call is actually invoked, as shown below, the Partition uses its in-process Transaction Manager to begin the transaction.
3. The session bean does some database work.
4. When the call is over, the Partition issues commit.
5. The Transaction Manager calls `commit_one_phase()` on the database resource.



Example 2PC usage

1. The client calls `OrderSesEJB.create()` method to create a new order.
2. Since the bean is configured to use the OTS engine named **TwoPhaseEngine**, the container locates the right transaction service named TwoPhaseEngine, and uses it for beginning the transaction.
3. The session bean does some database work, and sends a message to a JMS queue.
4. When the call is over, the Partition issues commit.

5 The OTS engine coordinates the transaction completion with the database and the JMS resources.



Example 2PC usage with MDBs

At some point in time, an asynchronous message is delivered to `OrderCompletionMDB` by invoking its `onMessage()` method, which has a `REQUIRED` transaction attribute. The container starts a transaction using ITS and then invokes the `onMessage()` method. In the body of the method, the bean updates the database to indicate order delivery. It is important to note that there are 2 resources involved. The first one is the JMS resource, which is associated with the MDB instances that got the message, and the second is the database that the MDB instance updated. This scenario is similar to the example diagram above.

Note `ejb.transactionManagerInstanceName` is also supported for MDBs. See "MDBs and Transactions" on page 184 for more information.

Declarative transaction management in Enterprise JavaBeans

Transaction management for Enterprise JavaBeans (EJBs) is handled by the EJB Container and the EJBs. Enterprise JavaBeans make it possible for applications to update data in multiple databases within a single transaction.

EJBs utilize a declarative style of transaction management that differs from the traditional transaction management style. With declarative management, the EJB declares its transaction attributes at deployment time. The transaction attributes indicate whether the EJB container manages the bean's transactions or whether the bean itself manages its own transactions, and, if so, to what extent it does its own transaction management.

Traditionally, the application was responsible for managing all aspects of a transaction. This entailed such operations as:

- Creating the transaction object.
- Explicitly starting the transaction.
- Registering resources involved in the transaction.
- Keeping track of the transaction context.
- Committing the transaction when all updates completed.

It requires a developer with extensive transaction processing expertise to write an application that is responsible for managing a transaction from start to finish. The code for such an application is more complex and difficult to write, and it is easy for “pilot error” to occur.

With declarative transaction management, the EJB container manages most if not all aspects of the transaction for you. The EJB container handles starting and ending the transaction, plus maintains its context throughout the life of the transaction object. This greatly simplifies an application developer's responsibilities and tasks, especially for transactions in distributed environments.

Understanding bean-managed and container-managed transactions

When an EJB programmatically performs its own transaction demarcation as part of its business methods, that bean is considered to be using bean-managed transaction. On the other hand, when the bean defers all transaction demarcation to its EJB container, and the container performs the transaction demarcation based on the Application Assembler's deployment instructions, then the bean is referred to as using container-managed transaction.

EJB session beans, both stateful and stateless varieties, can use either container- or bean-managed transactions. However, a bean cannot use both types of transaction management at the same time. EJB entity beans can only use container-managed transaction. It is the bean provider who decides the type of transaction which an EJB can use.

An EJB can manage its own transaction if it wishes to start a transaction as part of one operation and then finish the transaction as part of another operation. However, such a design might be problematic if one operation calls the transaction starting method, but no operation calls the transaction ending method.

Whenever possible, enterprise beans should use container-managed transactions as opposed to bean-managed transactions. Container-managed transactions require less programming work and are less prone to programming error. In addition, a container-managed transaction bean is easier to customize and compose with other beans.

Local and Global transactions

A transaction involves an atomic unit of work performed against data maintained by one or more resource managers. Examples of resource managers are database managements systems and JMS message providers. A local transaction involves work performed against a single resource manager independent of an external transaction manager. For instance, a JDBC connection obtained from a database can have SQL operations performed on it to update the database and then have the work committed in a local transaction using a `commit()` operation, assuming autoCommit mode for the connection is turned off, otherwise each operation is performed within a local transaction. A global transaction is coordinated by a transaction manager, such as the partition Transaction Manager or OTS engine, and it can involve work performed for one or more distributed resource managers. Transaction management for EJBs, both container-managed and bean-managed, implies use of global transactions. When a single resource manager participates in a global transaction, all work may be performed within a local transaction on behalf of the global transaction (refer to EJB Specification Version 2.0, Section 17.6.4 Local transaction optimization for more details).

Methods of an EJB defined with bean-managed transactions must obtain an implementation handle to JTA interface `javax.transaction.UserTransaction` and invoke operations on it to explicitly participate in a global transaction.

With container-managed transactions, the EJB Container interposes each EJB method call and follows certain rules to determine whether or not work should be processed as part of a global transaction. The decision taken by the Container depends on the transaction attribute value set for the method, by the Application Assembler in the components deployment descriptor, and whether a global transaction context exists upon invocation of the method (refer to Table 14 in EJB Specification Version 2.0, Section 17.6.2.7 Transaction attribute summary). Should the method be processed without the presence of a global transaction context, work performed against an external resource manager from within the method is completed using local transaction(s). The following are specific examples of when local transactions are used for EJB methods of an EJB with container-managed transaction demarcation:

- If the transaction attribute is set to `NotSupported` and the container detects that resources were accessed.
- If the transaction attribute is set to `Supports` and the container detects that a) the method was not invoked from within a global transaction, and b) resources were accessed.
- If the transaction attribute is set to `Never` and the container detects that resources were accessed.

Transaction attributes

EJBs that use bean-managed transaction have transaction attributes associated with each method of the bean. The attribute value tells the container how it must manage the transactions that involve this bean. There are six different transaction attributes that can be associated with each method of a bean. This association is done at deployment time by the Application Assembler or Deployer.

These attributes are:

- `Required`: This attribute guarantees that the work performed by the associated method is within a global transaction context. If the caller already has a transaction context, then the container uses the same context. If not, the container begins a new transaction automatically. This attribute permits easy composition of multiple beans and co-ordination of the work of all the beans using the same global transaction.
- `RequiresNew`: This attribute is used when the method does not want to be associated with an existing transaction. It ensures that the container begins a new transaction.
- `Supports`: This attribute permits the method to avoid using a global transaction. This must only be used when a bean's method only accesses one transaction resource, or no transaction resources, and does not invoke another enterprise bean. It is used solely for optimization, because it avoids the cost associated with global transactions. When this attribute is set and there is already a global transaction, the EJB Container invokes the method and have it join the existing global transaction. However, if this attribute is set, but there is no existing global transaction, the Container starts a local transaction for the method, and that local transaction completes at the end of the method.
- `NotSupported`: This attribute also permits the bean to avoid using a global transaction. When this attribute is set, the method must not be in a global transaction. Instead, the EJB Container suspends any existing global transaction and starts a local transaction for the method, and the local transaction completes at the conclusion of the method.

- `Mandatory`: It is recommended that this attribute not be used. Its behavior is similar to Requires, but the caller must already have an associated transaction. If not, the container throws a `javax.transaction.TransactionRequiredException`. This attribute makes the bean less flexible for composition because it makes assumptions about the caller's transaction.
- `Never`: It is recommended that this attribute not be used. However, if used, the EJB Container starts a local transaction for the method. The local transaction completes at the conclusion of the method.

Under normal circumstances only two attributes, `Required` and `RequiresNew`, must be used. The attributes `Supports` and `NotSupported` are strictly for optimization. The use of `Never` and `Mandatory` are not recommended because they affect the composibility of the bean. In addition, if a bean is concerned about transaction synchronization and implements the `javax.ejb.SessionSynchronization` interface, then the Assembler/ Deployer can specify only the attributes `Required`, `RequiresNew`, or `Mandatory`. These attributes ensure that the container invokes the bean only within a global transaction, because transaction synchronization can only occur within a global transaction.

Note When a client calls an EJB which in turn calls another EJB, and both EJBs access the same database, only one JDBC connection will be used if the Transaction attribute of the methods involved is set to Required. The reason is that work done in each of the beans becomes part of the single transaction.

Programmatic transaction management using JTA APIs

All transactions use the Java Transaction API (JTA). When transactions are container managed, the platform handles the demarcation of transaction boundaries and the container uses the JTA API; you do not need to use this API in your bean code.

A bean that manages its own transactions (bean-managed transaction), however, must use the JTA `javax.transaction.UserTransaction` interface. This interface allows a client or component to demarcate transaction boundaries. Enterprise JavaBeans that use bean-managed transactions use the method `EJBContext.getUserTransaction()`.

In addition, all transactional clients use JNDI to look up the `UserTransaction` interface. This simply involves constructing a JNDI `InitialContext` using the JNDI naming service, as shown in the following line of code:

```
javax.naming.Context context = new javax.naming.InitialContext();
```

Once the bean has obtained the `InitialContext` object, it can then use the JNDI `lookup()` operation to obtain the `UserTransaction` interface, as shown in the following code sample.

```
javax.transaction.UserTransaction utx = (javax.transaction.UserTransaction)
      context.lookup("java:comp/UserTransaction");
```

Note that an EJB can obtain a reference to the `UserTransaction` interface from the `EJBContext` object. This is because an enterprise bean by default inherits a reference to the `EJBContext` object. Thus, the bean can simply use the `EJBContext.getUserTransaction()` method rather than having to obtain an `InitialContext` object and then using the JNDI `lookup()` method. However, a transactional client that is not an enterprise bean must use the JNDI lookup approach.

When the bean or client has the reference to the `UserTransaction` interface, it can then initiate its own transactions and manage these transactions. That is, you can use the `UserTransaction` interface methods to begin and commit (or rollback) transactions. You use the `begin()` method to start the transaction, then the `commit()` method to commit the changes to the database. Or, you use the `rollback()` method to abort all changes made within the transaction and restore the database to the state it was in prior to the start of the transaction. Between the `begin()` and `commit()` methods, you include code to carry out the transaction's business.

JDBC API Modifications

The standard Java Database Connectivity (JDBC) API is used by the AppServer to access databases that support JDBC through vendor provided drivers. Requests for access to a database is centralized through the AppServer JDBC Connection Pool. This section describes modifications the AppServer JDBC pool makes to JDBC behavior for transactions.

The JDBC pool is a pseudo JDBC driver that allows a transactional application to obtain a JDBC connection to a database. The JDBC pool associates JDBC connections with the Transaction Manager's transactions, and delegates connection requests to JDBC drivers that factory the JDBC connections. Once a connection is obtained using the JDBC pool, the transaction is coordinated automatically by the transaction service.

The JDBC pool and its associated resources provide complete transactional access to the DBMS. The JDBC pool registers resources transparently with the transaction coordinator. Because of limitations of the 1.x version of the JDBC API, the JDBC pool can only provide one-phase commit. Version 2.0 of the JDBC API supports full two-phase commit.

Modifications to the behavior of the JDBC API

To enable JDBC access for transactional applications written in Java, you use the JDBC API. The JDBC API is fully documented at the following web site:

```
http://www.javasoft.com/products/jdk/1.2/docs/guide/jdbc/spec/
jdbc-spec.frame.html
```

However, the behavior of some JDBC methods is overridden by the partition's transaction service when they are invoked within the context of a transaction managed by the partition. The following methods are affected:

- `Java.sql.Connection.commit()`
- `Java.sql.Connection.rollback()`
- `Java.sql.Connection.close()`
- `Java.sql.setAutoCommit(boolean)`

The rest of this section explains the changes to the semantics of these methods for partition-managed transactions.

Note If a thread is not associated with a transaction, all of these methods will use the standard JDBC transaction semantics.

Overridden JDBC methods

Java.sql.Connection.commit()

As defined in the JDBC API, this method commits all work that was performed on a JDBC connection since the previous `commit()` or `rollback()`, and releases all database locks.

If a global transaction is associated with the current thread of execution do not use this method. If the global transaction is not a container-managed transaction, that is the application manages its own transactions, and a commit is required use the JTA API to perform the commit rather than invoking `commit()` directly on the JDBC connection.

Java.sql.Connection.rollback()

As defined in the JDBC API, this method rolls back all work that was performed on a JDBC connection since the previous `commit()` or `rollback()`, and releases all database locks.

If a global transaction is associated with the current thread of execution do not use this method. If the global transaction is not a container-managed transaction, that is the application manages its own transactions, and a rollback is required use the JTA API to perform the rollback rather than invoking `rollback()` directly on the JDBC connection.

Java.sql.Connection.close()

As defined in the JDBC API, this method closes the database connection and all JDBC resources associated with the connection.

If the thread is associated with a transaction this call simply notifies the JDBC pool that work on the connection is complete. The JDBC pool releases the connection back to the connection pool once the transaction has completed. JDBC connections opened by the JDBC pool cannot be closed explicitly by an application.

Java.sql.Connection.setAutoCommit(boolean)

As defined in the JDBC API, this method is used to set the auto commit mode of a transaction. The `setAutoCommit()` method allows Java applications to either:

- Execute and commit all SQL statements as individual transactions (when set to true). This is the default mode, or
- Explicitly invoke `commit()` or `rollback()` on the connection (when set to false).

If the thread is associated with a transaction, the JDBC pool turns off the auto-commit mode for all connections factoried in the scope of a partition's transaction service transaction. This is because the transaction service must control transaction completion. If an application is involved with a transaction, and it attempts to set the auto commit mode to true, the `java.sql.SQLException()` will be raised.

Handling of EJB exceptions

Enterprise JavaBeans can throw application and/or system level exceptions if they encounter errors while handling transactions. Application-level exceptions pertain to errors in the business logic and are intended to be handled by the calling application. System-level exceptions, such as runtime errors, transcend the application itself and can be handled by the application, the bean, or the bean container.

The EJB declares application-level exceptions and system-level exceptions in the `throws` clauses of its `Home` and `Remote` interfaces. You must check for checked exceptions in your program `try`/`catch` block when calling bean methods.

System-level exceptions

An EJB throws a system-level exception, which is a `java.ejb.EJBException` (but may also be a `java.rmi.RemoteException`), to indicate an unexpected system-level failure. For example, it throws this exception if it cannot open a database connection. The `java.ejb.EJBException` is a runtime exception and does not have to be listed in the `throws` clause of the bean's business methods.

System-level exceptions usually require the transaction to be rolled back. Often, the container managing the bean does the rollback. Other times, especially with bean-managed transactions, the client must rollback the transaction.

Application-level exceptions

An EJB throws an application-level exception to indicate application-specific error conditions, that is, business logic errors and not system problems. These application-level exceptions are exceptions other than `java.ejb.EJBException`. Application-level exceptions are checked exceptions, which means you must check for them when you call a method that potentially can throw this exception.

The EJB's business methods use application exceptions to report abnormal application conditions, such as unacceptable input values or amounts beyond acceptable limits. For example, a bean method that debits an account balance can throw an application exception to report that the account balance is not sufficient to permit a particular debit operation. A client can often recover from these application-level errors without having to rollback the entire transaction.

The application or calling program gets back the same exception that was thrown and this allows the calling program to know the precise nature of the problem. When an application-level exception occurs, the EJB instance does not automatically rollback the client's transaction. The client now has the knowledge and the opportunity to evaluate the error message, take the necessary steps to correct the situation, and recover the transaction. Otherwise, the client can abort the transaction.

Handling application exceptions

Because application-level exceptions report business logic errors, the client is expected to handle these exceptions. While these exceptions can require transaction rollback, they do not automatically mark the transaction for rollback. You often have the option to retry the transaction, though there are times when you must abort and rollback the transaction.

The bean Provider is responsible for ensuring that the state of the bean is such that, if the client continues with the transaction, there is no loss of data integrity. If the Provider cannot ensure this degree of integrity, then the bean marks the transaction for rollback.

Transaction rollback

When your client program gets an application exception, you must first check if the current transaction has been marked for “rollback only”. For example, a client can receive a `javax.transaction.TransactionRolledbackException`. This exception indicates that the helper enterprise bean failed and the transaction has been aborted or marked “rollback only”. In general, the client does not know the transaction context within which the called enterprise bean operated. The called bean may have operated in its own transaction context separate from the calling program's transaction context, or it may have operated in the calling program's context.

If the EJB operated in the same transaction context as the calling program, then the bean itself (or its container) may have already marked the transaction for rollback. When the EJB container has marked a transaction for rollback, the client should stop all work on the transaction. Normally, a client using declarative transactions will get an appropriate exception, such as `javax.transaction.TransactionRolledbackException`. Note that declarative transactions are those transactions where the container manages the transaction details.

A client that is itself an EJB calls the `javax.ejb.EJBContext.getRollbackOnly` method to determine if its own transaction has been marked for rollback or not.

For bean-managed transactions—those transactions managed explicitly by the client—the client should rollback the transaction by calling the `rollback` method from the `java.transaction.UserTransaction` interface.

Options for continuing a transaction

When a transaction is not marked for rollback, then the client has three options:

- Rollback the transaction.
- Pass the responsibility by throwing a checked exception or re-throwing the original exception.
- Retry and continue the transaction. This can entail retrying portions of the transaction.

When a client receives a checked exception for a transaction not marked for rollback, its safest course is to rollback the transaction. The client does this by either marking the transaction as “rollback only” or, if the client has actually started the transaction, calling the `rollback` method to actually rollback the transaction.

The client can also throw its own checked exception or re-throw the original exception. By throwing an exception, the client lets other programs further up the transaction chain decide whether or not to abort the transaction. However, in general it is preferable for the code or program closest to the occurrence of the problem to make the decision about saving the transaction.

Lastly, the client can continue with the transaction. The client can evaluate the exception message and decide if invoking the method again with different parameters is likely to succeed. However, you need to keep in mind that retrying a transaction is potentially dangerous. You have no knowledge of nor guarantee that the enterprise bean properly cleaned up its state.

Clients that are calling stateless session beans, on the other hand, can retry the transaction with more confidence if they can determine the problem from the thrown exception. Because the called bean is stateless, the client does not have the problem of not knowing the state in which the bean left the transaction.

Chapter 19

Message-Driven Beans and JMS

JMS and EJB

According to the specification, there are no limitations on a bean acting as a JMS message producer or synchronous consumer. It can use the regular JMS APIs to send a message to a queue or publish to a topic. As long as you perform synchronous style consumption of messages (that is, not based on `javax.jms.MessageListener`), then there are no problems on the consumption side either. The complexity lies in the need for a JMS message send or receive request to participate in a transaction context shared by other work in an application. We already know how to solve this problem using JMS and JTA in a non-EJB application. The EJBs demand no special treatment.

Since EJB method invocations are synchronous, some calls will have to wait until the bean has completed its processing. This may include calling other beans, databases, and so forth. This RMI behavior can be undesirable in many situations. For example, you may just want to call the method and have it return before doing any heavy processing, allowing the caller to proceed with other tasks in the meantime. Threading in the client is an obvious way to achieve this, but it suffers from two problems:

- the client's programming model is not a true asynchronous style
- if the client is an EJB, threading is prohibited in its method implementations

The most desirable scenario is for an appclient, servlet, EJB, or other component to have the capability to fire a message and then have an EJB be driven asynchronously by that message. In turn, that EJB can send a message to another EJB or perform direct data access or other business logic. The caller does not wait beyond the time the message is successfully queued. On the other side, the EJB can process the message at its convenience. This EJB's processing typically involves a unit of work made up of three operations:

1. dequeueing the message,
2. activating an instance and performing whatever work the business logic demands, and
3. optionally queuing a reply message back

Enterprise systems require that it be possible to have transactional and other container-managed guarantees for this unit of work.

EJB 2.0 Message-Driven Bean (MDB)

The EJB 2.0 specification formalizes the integration between JMS and asynchronous invocation of enterprise beans by pushing these responsibilities to the EJB Container. This eases the burden on the developer, who now simply provides a class that is a JMS listener and also an EJB. This is done by implementing `javax.jms.MessageListener` and `javax.ejb.MessageDrivenBean` in the class. This and an XML descriptor containing all the deployment settings is all that the application programmer needs to provide.

From a client's perspective, this EJB is nonexistent. The client simply publishes messages to the queue or topic. The EJB container associates the MDB with the published queue/topic and handles all lifecycle, pooling, concurrency, reentrance, security, transaction, message handling, and exception handling issues.

EJB 2.1 MDB

With integration of the J2EE Connector Architecture 1.5 (JCA 1.5) in EJB 2.1 the MDB can now process messages from non-JMS messaging servers in addition to JMS based providers. A JCA 1.5 compliant resource adapter implementation can be developed for any type of messaging server and deployed to an application server. When configured to pass inbound messages from the messaging server to the application server, the resource adapter can be selected as the source for messages driving 2.1 MDBs.

JCA 1.5 defines a Message Inflow contract which is a messaging contract between the EJB container and an asynchronous connector, so that MDBs automatically process incoming messages from an EIS or some other type of messaging provider. EJB 2.1 MDBs must implement the standard javax.ejb.MessageDrivenBean interface as well as a specific messaging interface defined by the connector. If the connector is a JMS-based provider, the MDB must implement javax.jms.MessageListener, but for non-JMS providers it must implement some other type of interface that is specific to the provider.

In Borland Application Server 6.6, EJB 2.1 MDBs can be configured to process messages from JMS providers either indirectly through a JCA resource adapter or directly without the need for a pre-deployed JCA resource adapter.

Client View of an MDB

Clients do not bind to an MDB like they do for session beans and entity beans. The client only needs to send a message to the destination to which the MDB is configured to listen. Typically clients also use the `<resource-ref>` and `<resource-env-ref>`(in EJB 2.0) or `<message-destination-ref>`(in EJB 2.1) for JMS destination specification in their deployment descriptor and then point to the same JNDI names as configured in the MDB deployment descriptor. See “Obtaining JMS Connection Factories and Destinations in J2EE Application Components” on page 209 for information on how to configure your client deployment descriptors to communicate with the JMS provider.

This being the case, there is no EJB metadata or handle of which the client needs to be aware. This is because there is no RMI client view of a Message Driven Bean.

MDB Configuration

Since MDBs do not expose EJB interfaces, they do not have JNDI names in the normal sense like `EJBHome` objects do. When an MDB is deployed, it communicates with a message provider in preparation for processing incoming messages.

EJB 2.0 MDBs are associated with two JMS resource objects that must pre-exist in JNDI before the MDB is deployed. These are:

- a JMS connection factory to use for connecting to the JMS provider and
- a JMS queue/topic on that provider to listen to for incoming messages

The JNDI names for these objects are specified in the MDB's `ejb-borland.xml` deployment descriptor. The `<connection-factory-name>` captures the resource connection factory used to connect to the JMS service provider. The `<message-driven-destination-name>` element captures the actual topic/queue on which the MDB is to listen. Once these elements are specified, the MDB has all the information it needs to connect to the JMS service provider, receive messages, and send replies.

EJB 2.1 MDBs can be configured in one of two ways. If the EJB 2.1 MDB implements `javax.jms.MessageListener`, indicating a JMS based MDB, it can be configured to communicate directly with the JMS provider rather than use a JCA 1.5 connector. In this case, JNDI names for JMS resource objects can be specified in the MDB's `ejb-borland.xml` deployment descriptor under `<jms-provider-ref>` element. An EJB 2.1 MDB can alternatively be configured to receive messages from a JCA 1.5 connector using element `<resource-adapter-ref>` in the Borland-specific deployment descriptor file `ejb-borland.xml`.

Connecting to a JMS Server from EJB 2.0 MDBs

EJB 2.0 MDBs provide a special case for connecting to the JMS server, the source for inbound messages. In the standard deployment descriptor file, `ejb-jar.xml`, the type of JMS destination from which the inbound messages are received, is defined using the `<message-driven-destination>` element in the MDB's declaration. For example:

```
<message-driven>
 <ejb-name>MyMDBTopic</ejb-name>
⋮
<message-driven-destination>
  <destination-type>javax.jms.Topic</destination-type>
  <subscription-durability>Durable</subscription-durability>
 </message-driven-destination>
⋮
</message-driven>
```

Consult the J2EE 1.3 Specification for the proper use of this element. The Borland-specific XML file, `ejb-borland.xml` binds the logical name of the JMS destination with the JNDI name using equivalent element, `<message-driven-destination>`. The JNDI name for JMS connection factory required to connect to JMS server must also be defined using `<connection-factory-name>`. For example,

```
<message-driven>
<ejb-name>MyMDBTopic</ejb-name>
⋮
<message-driven-destination>jms/resources/Topic</message-driven-destination>
<connection-factory-name>jms/resources/tcf</connection-factory-name>
⋮
</message-driven>
```

See the Configuring JMS Connection Factories and Destinations section in the *Using JMS* section for detailed information on configuring these JMS resource objects bound under JNDI.

Note You must use an XA connection factory when the MDB is deployed with the `REQUIRED` transaction attribute. The whole idea of this deployment is to enable the consumption of the message that drives the MDB to share the same transaction as any other work that is done from within the `MDB.onMessage()` method. To achieve this the container performs XA coordination with the JMS service provider and any other resources enlisted in the transaction.

Connecting to message source from EJB 2.1 MDBs

As a result of EJB 2.1 and JCA 1.5, there have been changes to both the standard deployment descriptor, `ejb-jar.xml`, and the Borland proprietary deployment descriptor, `ejb-borland.xml` for J2EE 1.4.

Changes to ejb-jar.xml

Each EJB 2.1 MDB is connected to its message source based on the information in the deployment descriptor. The standard deployment descriptor, `ejb-jar.xml`, in EJB 2.1 has changed to accommodate the connector-based MDB.

EJB 2.1 adds new elements, `<messaging-type>`, `<message-destination-type>`, and `<activation-config>`, in the `ejb-jar.xml` file:

The `<messaging-type>` element indicates which message type will be used, it does this by stating the fully-qualified interface name that the MDB implements. If no interface name is given, the container defaults to JMS message type, `javax.jms.MessageListener`.

The optional `<message-destination-type>` element designates a fully-qualified interface name that represents the type of destination from which the bean will get messages. For MDBS that represent JMS message type, `javax.jms.MessageListener`, the allowed values are `javax.jms.Topic` or `javax.jms.Queue`.

Since the connector-based MDBs no longer rely exclusively on JMS, the EJB 2.0 elements `<message-driven-destination><message-selector>` and `<acknowledge-mode>` elements are eliminated in EJB 2.1. Configuration properties required by EJB 2.1 MDB activation can be defined in a generic set of name-value pairs under the `<activation-config>` element. The property names and values used to describe the messaging service vary depending on the type of service used. These `<activation-config>` properties are examined when the message-driven bean is deployed. Each eliminated JMS-related element from EJB 2.0 can be represented by an `<activation-config-property>` element when `<messaging-type>` element specifies a JMS messaging type (`javax.jms.MessageListener`)

Here is an example of how a JMS-based MDB can be defined in EJB 2.1 `ejb-jar.xml` file:

```
<enterprise-beans>
 <message-driven>
   <ejb-name>EJB_SEC_MDB_TOPIC_CMT</ejb-name>
   <ejb-class>com.sun.ts.tests.ejb.ee.sec.mdb.MsgBean</ejb-class>
   <messaging-type>javax.jms.MessageListener</messaging-type>
   <transaction-type>Container<transaction-type>
   <message-destination-type>javax.jms.Topic</message-destination-type>
   <message-destination-link>StockTopic</message-destination-link>
   <activation-config>
    <activation-config-property>
     <activation-config-property-name>acknowledgeMode
        </activation-config-property-name>
     <activation-config-property-value>Auto-acknowledge
```

```
        <activation-config-property-value>
      </activation-config-property>
      <activation-config-property>
       <activation-config-property-name>destinationType
        </activation-config-property-name>
       <activation-config-property-value>javax.jms.Topic
        <activation-config-property-value>
      </activation-config-property>
      <activation-config-property>
       <activation-config-property-name>subscriptionDurability
        </activation-config-property-name>
       <activation-config-property-value>DURABLE
        <activation-config-property-value>
      </activation-config-property>
     <activation-config>
   </message-driven>
   ⋮
  </enterprise-beans>
```

The property names and values used in the <activation-config> to describe the messaging service vary depending on the type of message service used, but EJB 2.1 defines the following four fixed properties for JMS-based MDBs:

<activation-config-property-name>	Description	<activation-config-property-value>
acknowledgeMode	Allows an MDB container to notify the JMS provider that the MDB has received the message.	Auto-acknowledge (default) or Dups-ok-acknowledge
messageSelector	Allows an MDB to be selective about which messages it receives. It allows an MDB to set properties based on which messages will be received. These properties can be expressions or Boolean logic.	String selector
destinationType	Indicates the type of destination from which the MDB receives messages	javax.jms.Queue or javax.jms.Topic
subscriptionDurablity	Determines whether the JMS provider must store any messages that an MDB receives when the MDB container is disconnected from the provider.	NonDurable (default) or Durable

The Message Inflow contract in the JCA 1.5 specification is a contract between the messaging service provider and the application server for the delivery of messages to an MDB. As a part of this contract, the messaging provider implements a JavaBean component called ActivationSpec. The ActivationSpec defines the properties that are required by the messaging provider in order to deliver messages. The administrator may define default values for these properties, but when the application containing the MDB is deployed, these will be overridden by any `<activation-config-property>` elements defined in the MDB's deployment descriptor. A JMS provider that confirms to the Sun specification will therefore have the above mentioned properties on its ActivationSpec. You can leave a property out of the MDB's deployment descriptor and define it administratively instead. Conversely, there may be other, provider-specific properties that you previously had to define administratively that you can now include in the MDB's deployment descriptor.

Standard descriptor element `<message-destination-link>` is used to define a logical name for the message destination. It is used with a `<message-destination>` element to illustrate message flow within an application. For MDBs that specify a JMS provider message source, the JMS destination object is resolved from the target `<message-destination>` of `<message-destination-link>`, if present in the MDBs deployment descriptor.

In EJB 2.1 MDBs, standard descriptor element <message-destination-ref> can be used instead of <resource-env-ref> for definition of JMS destinations used within application logic of the MDB.

Changes to ejb-borland.xml

The Borland proprietary deployment descriptor has been modified to accommodate the new connector-based MDB. It now includes a new element, <message-source>. This element allows an application assembler to specify activation of the MDB through a JCA 1.5 resource adapter or in the case of a JMS messaging type MDB directly to a JMS provider. If you are using a JMS provider, you must use the <jms-provider-ref> element as follows:

```
<enterprise-beans>
   <message-driven>
      <ejb-name>EJB_SEC_MDB_TOPIC_CMT</ejb-name>
      <message-source>
         <jms-provider-ref>
            <message-driven-destination-name>
              Jms/MyTopic
            </message-driven-destination-name>
            <connection-factory-name>jms/myTCF</connection-factory-name>
            <pool>
               <max-size>120</max-size>
               <init-size>100</init-size>
               <wait-timeout>600</wait-timeout>
            </pool>
         </jms-provider-ref>
      </message-source>
          ⋮
   </message-driven>
</enterprise-beans>
```

If you are using a Connector-based non-JMS messaging provider, use the following <message-source>:

```
<enterprise-beans>
   <message-driven>
      <ejb-name>EJB_SEC_MDB_TOPIC_CMT</ejb-name>
      <message-source>
         <resource-adapter-ref>
            <instance-name>
              MyResourceApadter
            </instance-name>
         </resource-adapter-ref>
      </message-source>
   </message-driven>
</enterprise-beans>
```

The resource adapter may provide the Java class name and the interface type of an optional set of JavaBean classes representing various administered objects. Administered objects are specific to a messaging style or message provider and can be referenced using <resource-env-ref> from the application logic of an MDB. For example, some messaging styles may need applications to use special administered objects for sending and synchronously receiving messages via connection objects using messaging-style specific APIs. Borland deployment descriptor element <resource-env-ref> is extended to override property values of an administered object.

For example:

```
⋮
  <message-driven>
     <message-source>
        <resource-adapter-ref>
           <instance-name>ResourceAdapter1</instance-name>
        </resource-adapter-ref>
     </message-source>
     ⋮
     <resource-env-ref>
        <resource-env-ref-name>mdbRequiredConnFactory</resource-env-ref-name>
        <admin-object>
           <property>
               <prop-name>serverUrl</prop-name>
               <prop-type>String<prop-type>
               <prop-value>localhost:7222</prop-value>
           </property>
        </admin-object>
     </resource-env-ref>
     ⋮
  </message-driven>
  ⋮
```

Clustering of MDBs

The clustering of MDBs differs from the clustering of other enterprise beans. With MDBs, producers put messages into a destination. The messages will reside in the destination until a consumer takes the messages off the destination (or, if the messages are non-durable, when the server hosting the destination crashes). This is a *pull* model since the message will just reside on the destination until a consumer asks for it. The containers contend to get the next available message on the destination. MDBs provide an ideal load-balancing paradigm, one that is smoother than other enterprise bean implementations for distributing a load . The server that is the least burdened can ask for and obtain the message. The tradeoff for this optimal load-balancing is that messaging has extra container overhead by virtue of the destination's position between the producer and the consumer.

There is not, however, the same concept of failover with a messaging service as exists in VisiBroker. If the consumer disappears, the queue fills up with messages. As soon as the consumer is brought back online, the messages resume being consumed. Of course, the JMS server itself should be fault-tolerant. The client should never notice any “failure” with the exception of response delays if such messages are expected. This kind of fault tolerance demands only a way of detecting failed consumers and activating them after failure.

That said, it is possible to deploy MDBs in more than one Partition with the Messaging Server pushing messages to only one, switching to the other in case of failure. Most JMS products allow queues to behave in load-balancing or fault-tolerant modes. That is, MDB replicas can register to the same queue and the messages are distributed to them using a load-balancing algorithm. Alternately, messages may all go to one consumer until it fails, at which point delivery shifts to another. The connection established to the JMS service provider from the MDB can also provide a load-balancing and/or fault-tolerant node. JMS service providers may provide fault-tolerance features. For specific information on clustering and fault-tolerance features, see Chapter 23, “JMS provider pluggability.”

Keep in mind that only one MDB instance in a container that subscribes to a topic will consume any given message. This means that, for all parallel instances of an MDB to concurrently process messages, only one of the instances will actually receive any particular message. This frees up the other instances to process other messages that have been sent to the topic. Note that each container that binds to a particular topic will consume a message sent to that topic. The JMS subsystem will treat each message-driven bean in separate containers as a separate subscriber to that message! This means that if the same MDB is deployed to many containers in a cluster, then each deployment of the bean will consume a message from the topic to which it subscribes. If this is not the behavior you desire, and you require exactly one consumption of a message, then you should consider deploying a queue rather than a topic.

Error Recovery

The following section deals with JMS server connection failures and setting connection rebind attempt properties. It also covers the redelivery of messages when an MDB fails to consume a message.

Rebinding EJB 2.0 and EJB 2.1 MDBs configured with a JMS provider message source

A connection failure usually occurs after you deploy your bean, causing a need for rebind attempt. You also receive an error if you are trying to deploy your bean and a connection to the JMS server was never established. Whether a failure occurs post deployment or no connection was found during deployment, the container will transparently attempt to rebind the JMS service provider connection when you set the rebind attempt properties. This ensures even greater fault-tolerance from an MDB instance.

The two bean-level properties that control the number of rebind attempts made and the time interval between attempts are:

- `ejb.mdb.rebindAttemptCount`: this is the number of times the EJB Container tries to re-establish a failed JMS connection for this MDB. The default value is `5` (five).

  To make the container attempt to rebind infinitely you need to explicitly specify `ejb.mdb.rebindAttemptCount=0`.

- `ejb.mdb.rebindAttemptInterval`: the time in seconds between successive retry attempts. The default value is `60`.

Redelivered messages for EJB 2.0 and EJB 2.1 MDBs configured with a JMS provider message source

Should the MDB fail to consume a message for any reason, the message will be re-delivered by the JMS service. The message will only be re-delivered five times. After five attempts, the message will be delivered to a dead queue (if one is configured). There is one bean-level property that controls the re-deliver attempt count:

- `ejb.mdb.maxRedeliverAttemptCount`: the max number of times a message will be re-delivered by the JMS service provider if an MDB is unable to consume it. The default value is `5`.

There are two bean-level properties for delivering a message to a dead queue:

- `ejb.mdb.unDeliverableQueueConnectionFactory`: looks up JNDI name for the connection factory to create connection to the JMS service.
- `ejb.mdb.unDeliverableQueue`: looks up the JNDI name of the queue.

  The XML example for `unDeliverableQueueConnectionFactory` and `unDeliverableQueue` is shown here:

```
<ejb-jar>
   <enterprise-beans>
       <message-driven>
           <ejb-name>MyMDB</ejb-name>
          <message-driven-destination-name>serial://jms/q
            </message-driven-destination-name>
           <connection-factory-name>serial://jms/xaqcf
            </connection-factory-name>
           <pool>
               <max-size>20</max-size>
               <init-size>0</init-size>
           </pool>
           <resource-ref>
               <res-ref-name>jms/QueueConnectionFactory</res-ref-name>
               <jndi-name>jms/xaqcf</jndi-name>
           </resource-ref>
           <property>
               <prop-name>ejb.mdb.maxRedeliverAttemptCount</prop-name>
               <prop-type>String</prop-type>
               <prop-value>3</prop-value>
           </property>
           <property>
               <prop-name>ejb.mdb.unDeliverableQueueConnectionFactory
                  </prop-name>
               <prop-type>String</prop-type>
               <prop-value>serial://jms/qcf</prop-value>
           </property>
           <property>
               <prop-name>ejb.mdb.unDeliverableQueue</prop-name>
               <prop-type>String</prop-type>
               <prop-value>serial://jms/q2</prop-value>
           </property>
           <property>
               <prop-name>ejb-designer-id</prop-name>
               <prop-type>String</prop-type>
               <prop-value>MyMDB</prop-value>
           </property>
       </message-driven>
    </enterprise-beans>
   <assembly-descriptor />
</ejb-jar>
```

You can set these properties with the DDEditor. From the Console, navigate the tree on the left until you find the module containing your MDBs. Right-click the module and select DDEditor. When the DDEditor appears, select the bean node in the Navigation Pane to open the editor's panels for that bean. Select the “Properties” tab from the Content pane, and Add properties.

MDBs and Transactions

For information about using JMS within transactions, see “JMS and Transactions” on page 214. This section deals exclusively with using MDBs in transactions.

A common scenario for using the MDBs involves transactions requiring two-phase commit (2PC). Such an MDB has the `REQUIRED` transaction attribute. The MDB application method could be written to access and possibly update an external resource. Completion of the container managed transaction for the MDB method must include receipt of the message that triggered the method, and any work performed against the external resource from within the method. To achieve this, the transaction must be coordinated by a 2PC transaction service, such as the OTS engine. See “EJBs and 2PC transactions” on page 163 for more details on optimal ways to use the OTS engine with MDBs.

Chapter

20

Connecting to Resources with Borland AppServer: using the Definitions Archive (DAR)

J2EE specifies a uniform mechanism for establishing connections to resources using Java standard interfaces. A resource related object containing resource manager location details and connection attributes is bound under a JNDI service provider, and can be retrieved by your application as a resource connection factory in a JNDI lookup. Sample resource connection factories include JDBC datasources and JMS connection factories. Once a resource connection factory is obtained from JNDI, a connection to the desired resource manager can then be established. A connection to a relational database is obtained through a JDBC datasource, a connection to a message broker is obtained through a JMS connection factory, and general Enterprise Information Systems (EIS) connections are obtained through JCA resource adapters.

Use the Borland Management Console and Borland Deployment Descriptor Editor (DDEditor) to create, edit, and deploy resource connection factories and other resource related JNDI objects, such as JMS destinations. An XML descriptor file (`jndi-definitions.xml`), generally called the JNDI Definitions module, captures the properties representing resource related objects. This file is packaged in a Data ARchive (DAR) module.

In the Borland AppServer (AppServer), a partition hosted Naming Service represents the default JNDI service provider—its an implementation of a CosNaming service provider. Resource related objects are bound in the Naming Service of an AppServer partition through deployment of DAR or RAR module using standard AppServer deployment procedures. At that time, only properties required to create an instance of a resource connection factory, or JMS destination, are stored in the JNDI bound object. During JNDI lookup for a resource related object, an instance of the desired resource object is created using the stored property values from the object retrieved. The newly created instance is then passed back to the caller of JNDI `lookup()` method. In this way, DARs can be successfully deployed to an AppServer partition without having to load classes for vendor specific resource objects. Class libraries for resource vendors are only required by application processes that actually perform JNDI lookup of resource related objects.

Note In prior versions of AppServer, a file-system service provider called the Serial Provider was the default JNDI service provider for deployment of DARs and JNDI Definitions modules. Resource related objects bound to this provider involved creation of the resource objects during deployment, and hence required vendor class libraries to be deployed in advance. In addition, JNDI names for resource related objects required a serial URL prefix, that is "serial://". With the Naming Service being the default service provider, this prefix is no longer required in JNDI name specification. Deployment of existing DARs/JNDI Definitions modules that have JNDI names with this prefix are now automatically bound to the Naming Service.

A resource related object is obtained in J2EE through a resource reference. You can reference resource connection factories or JMS destinations from EJBs, servlets and other J2EE application components using resource-reference elements in the component's deployment descriptors. See Chapter 21, "Using JDBC" for more specific information on how to define JDBC datasource resource references, and Chapter 22, "Using JMS" for resource reference definition examples of JMS connection factories and destinations.

Each AppServer partition has a predeployed DAR module named `default-resources.dar` containing example definitions for JDBC datasources, JMS connection factories and JMS destinations. This module can be examined, updated and redeployed in the following steps:

1. Navigate to **default-resources.dar** under the Deployed Modules node of a partition in the left pane of the Borland Management console.
2. Right-click on **default-resources.dar** and select **Edit deployment descriptor** from the context menu. The Borland Deployment Descriptor Editor window will open. The available datasources and connection factories should be visible in the left pane.
3. Right-click on the root node in the navigation pane of the Borland Deployment Descriptor Editor and select the appropriate option to create a New object you want to add.

When a J2EE component attempts a JNDI lookup for a resource reference, vendor classes associated with the resource object must be available in the runtime environment. If the J2EE component is deployed to an AppServer partition, the vendor class libraries must be deployed to the partition as a library archive. Exceptions to this rule include JNDI lookups for resource objects whose dependent class libraries are bundled with AppServer. For example, a JDataStore datasource or any JMS resource object of the JMS message server installed with AppServer.

JNDI Definitions Module

Resource related objects are bound under the Naming Service through deployment of a DAR file containing the JNDI Definitions Module. A DAR has a special .dar file extension. It must be deployed to an AppServer partition either individually or packaged with other J2EE modules in an EAR file.

Note A DAR is *not* a part of the J2EE specification. It is a Borland-specific implementation designed to simplify deployment and management of resource connection factories and JMS destinations. You do *not* package connection factory or JMS destination vendor classes in this archive type. Those classes must be deployed as a library to individual Partitions.

The only contents of the DAR that you must provide is an XML descriptor file called `jndi-definitions.xml`. It contains definitions for resource related objects, each with a JNDI name identifying its location in JNDI. Like other descriptors, this is placed within the `META-INF` directory of the DAR. The contents of the DAR hence is as follows:

```
META-INF/jndi-definitions.xml
```

You deploy the DAR containing the descriptor file just as you would any other J2EE module using either the Console or command-line utilities, or as part of an EAR. You can deploy any number of distinctly named DARs in the same Partition or to an AppServer cluster. Should two or more deployed DARs have resource object definitions with identical JNDI names, the last deployed module overwrites any existing object binding on the same node.

Once deployed, resource objects defined in the DAR can be examined in the Naming Service namespace using the JNDI Browser.

Migrating to DARs from previous versions of Borland AppServer

Previous product versions, including IAS 4.1 and BAS 4.5, did not have a DAR module to contain the `jndi-definitions.xml` descriptor. If you have a customized `jndi-definitions.xml` file that needs to be transferred to AppServer, follow these migration steps:

1. If you want the entire contents of the default resources overridden, make a temporary directory called `META-INF` and place your existing `jndi-definitions.xml` file within it.
2. Open a command window and use the following `jar` command:

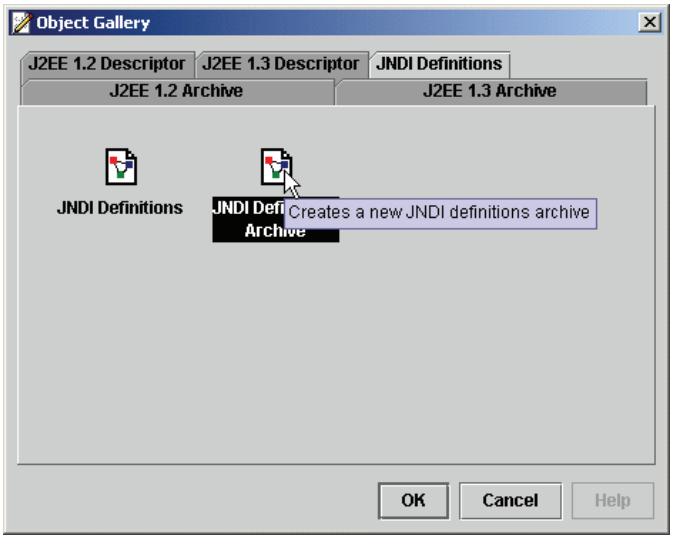
```
prompt>jar uvMf default-resources.dar META-INF/jndi-definitions.xml
```

3. Now deploy this module following the usual procedures.

If you have performed only a few customizations on your old `jndi-definitions.xml` file then it may be easier to simply move the appropriate XML stanzas from the old file into the one contained within the pre-deployed DAR.

Creating and Deploying a DAR

The DDEditor walks you through creating a new JNDI Definitions Module. Open the DDEditor and select “File|New...” The Object Gallery appears.



Select the JNDI Definitions tab and select JNDI Definitions Archive to create a new DAR. Click OK. You may now add JDBC datasources or add JMS resources. Or, you may do this later. When you are finished save the module by choosing “File|Save...” .

After saving the archive, use the J2EE Deployment Wizard to deploy the module. The wizard reads resource definitions from the DAR and binds them to the Naming Service of the target partition(s). To start the wizard, open the Console and select “Wizards| Deployment Wizard.” Follow the on-screen instructions.

Disabling and Enabling a Deployed DAR

Once a DAR module is deployed to a partition, it is enabled. This means that its resource object definitions are bound to the Naming Service of the partition for as long as the Naming Service remains active. Enabling a DAR module rebinds its resource object definitions to the Naming Service and in the process overwrites any pre-existing content for JNDI names specified. Disabling a DAR module has no immediate impact on the contents of an active Naming Service. Upon a subsequent restart of the partition, disabled DARs are not deployed to the partition, that is, resource object definitions are not bound to the Naming Service. By default, the Naming Service stores object bindings in memory. Each time the host partition is restarted, resource object bindings from previously deployed DARs are destroyed. If the Naming Service is configured with a JDBC backing store, resource object bindings for all DARs are retained, even those that were once deployed but now marked disabled. Use the JNDI Browser to locate and permanently remove these bindings.

To manipulate a deployed DAR module, use the Console to select it from the set of Deployed Modules for a partition, right-click and choose the appropriate action.

Packaging DAR Modules in an Application EAR

Sometimes it is useful to package all archives that make up a complete application into a single deployable unit. The common scenario is that you have some EJBs in an EJB Archive, some servlets and JSPs in a Web Archive and they both depend on some datasources or JMS administrative objects defined in a DAR. Using the Archive Tools in the Console it is easy to package a set of individual archives into a single EAR Module.

Note Because DARs are not a part of the J2EE specification, you must include at least one other valid J2EE module along with your DAR within the EAR. An EAR containing only a DAR file is not a valid J2EE archive.

Chapter

21

Using JDBC

Resource related objects such as JDBC datasources are obtained in a portable J2EE mandated way through JNDI. A JDBC datasource is resolved by performing a JNDI lookup of a J2EE Resource Reference defined in the deployment descriptors of an application component. Resource Reference definitions involve both standard J2EE and Borland's proprietary deployment descriptors. In the standard deployment descriptor, a Resource Reference specifies a logical name relative to the application's JNDI environment naming context, `java:comp/env/`. Borland's deployment descriptor complements the standard descriptor by associating the Resource Reference logical name with the actual JNDI location of the JDBC resource definition. For example, in an EJB Jar component, the standard J2EE deployment descriptor, `ejb-jar.xml`, specifies Resource References for an EJB using a <resource-ref> element for a JDBC datasource. In Borland AppServer (AppServer), a JNDI lookup of a Resource Reference involves retrieval of the JDBC datasource definition from which the desired datasource object is created and returned to caller of lookup. The property values present in the JDBC datasource definition determine the type and characteristics of datasource object created.

Before a Resource Reference lookup can be attempted, the required datasource definition must first be bound to its physical JNDI location. In AppServer, JDBC datasource definitions are bound into a JNDI service provider during deployment of a Definitions ARchive (DAR) module. By default, these objects are bound to the partitions Naming Service, the JNDI CosNaming service provider in AppServer. This chapter describes how to define JDBC datasource definitions in a DAR module and how to obtain a reference to a JDBC datasource from your J2EE application.

Configuring JDBC Datasources

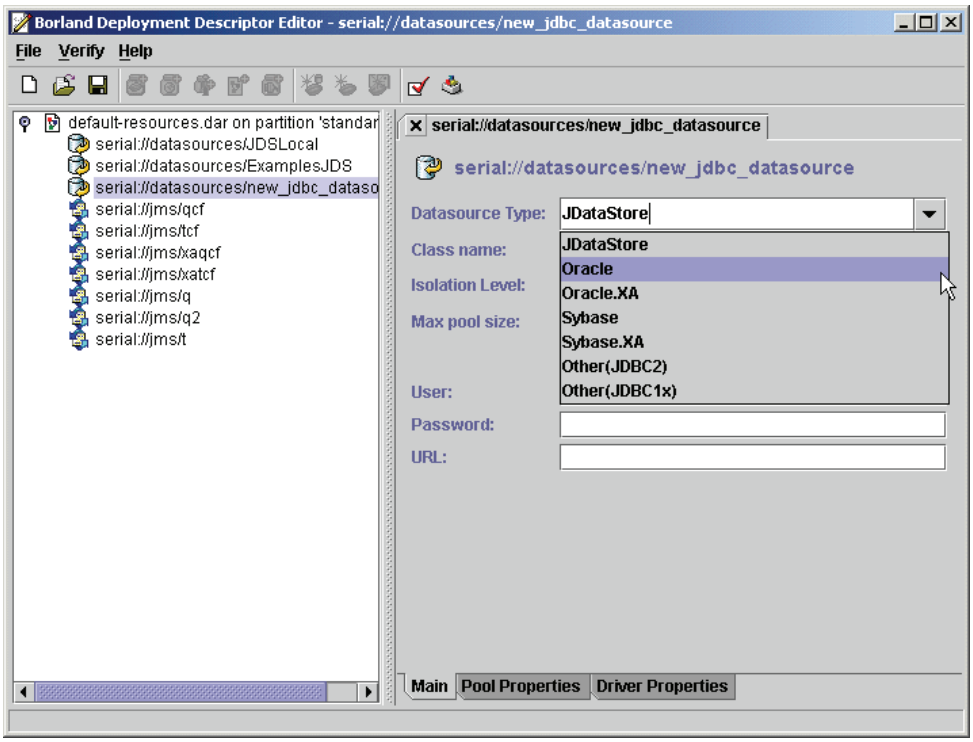
Using the Console, navigate to the “Deployed Modules” list in the Partition whose datasources you need to configure. By default, every Partition has a predeployed JNDI Definitions Module (DAR) called `default-resources.dar`. Right-click on the module and choose “Edit deployment descriptor” from the context menu. The Deployment Descriptor Editor (DDEditor) appears.

In the Navigation Pane of the DDEditor is a list of datasources preconfigured in the product. If needed they can be individually edited to suit the user's requirements.

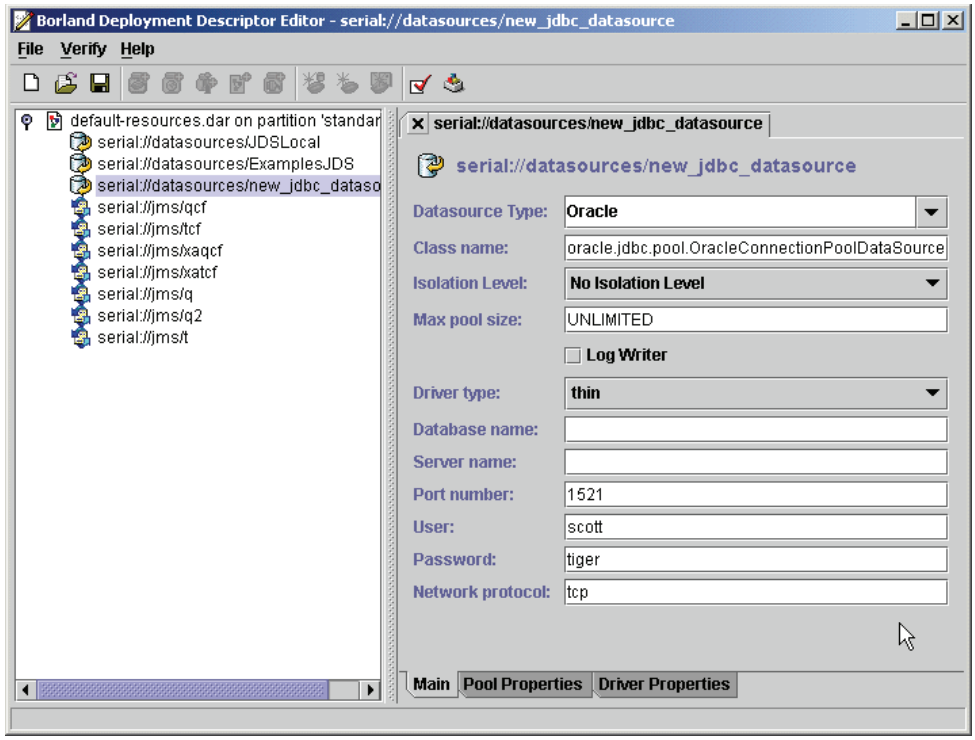
To create a new JDBC datasource, Right-click on the node at the top of the tree in the navigation pane and select “New JDBC Datasource...” from the Context Menu.

A dialog box prompts for a JNDI name for the newly-created datasource. Once given, a representation of this datasource appears in the tree in the Navigation Pane. Click its representation to open its configuration panel.

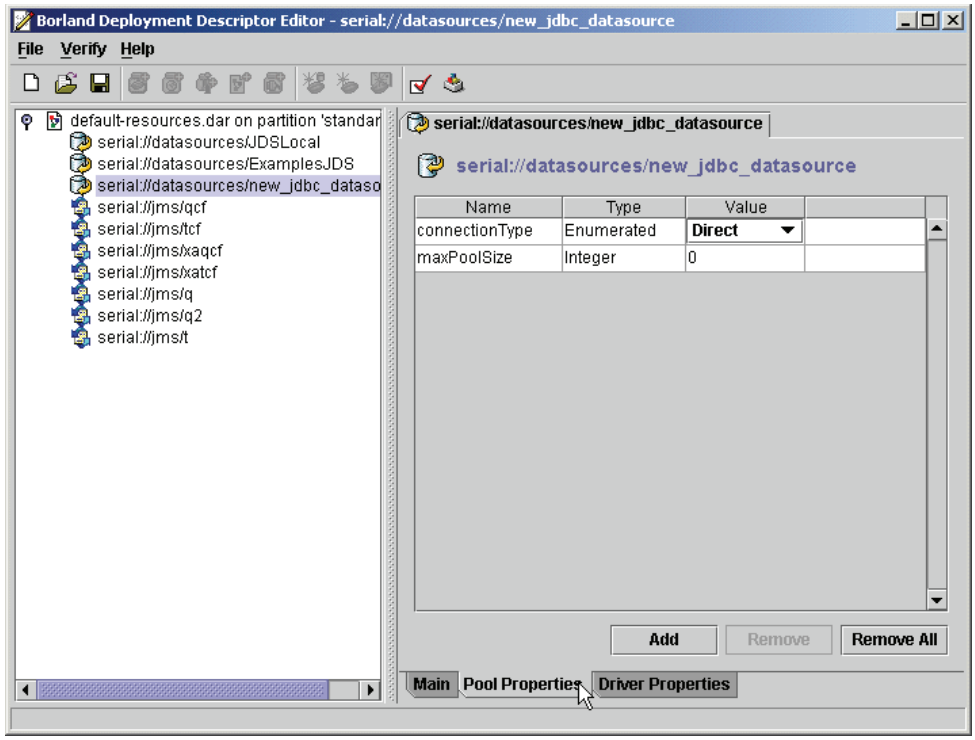
The DDEditor has knowledge of some common JDBC drivers and can autofill the class names and essential properties for the appropriate JDBC datasource. If the JDBC datasource type you want appears in the Datasource Type list then choose it, otherwise select “Other(JDBC2)”.



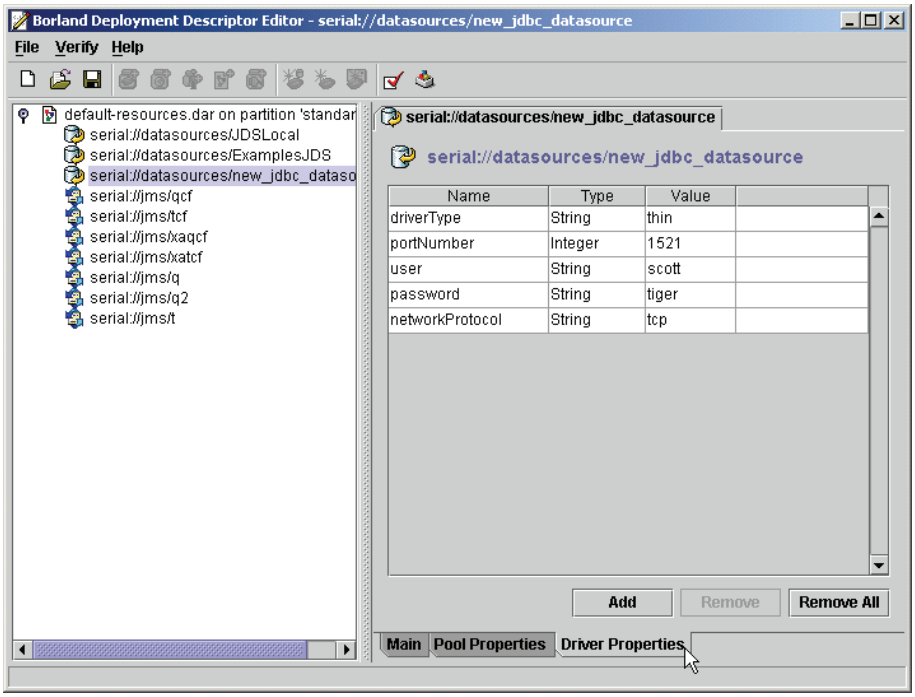
The Main tab in the content pane captures the essential properties needed to define the chosen database. If the database is known to the DDEditor, it will automatically complete these properties.



The Driver Properties and Pool Properties tabs capture some of the information from the Main tab, but also allow you to set any less common properties that do not appear in the Main tab.



To add pool properties, click the Add button and select the property you want to add from the drop-down list under “Name.” Pool Properties are described in “Defining the Connection Pool Properties for a JDBC Datasource” on page 193. The same procedure is used for adding Driver Properties.



Consult your database documentation for any properties you need to define.

Once you're finished, save the module and dismiss the final modal window. The JNDI Definitions Module is automatically re-deployed to the Partition.

Deploying Driver Libraries

If a deployed application component contains a JNDI lookup of third party JDBC datasource, vendor libraries are required and must be deployed to the target Partition as a library archive before the lookup is performed. Note that these steps are not necessary if you are using the native all-Java database, JDataStore. When trying to connect to another database like Oracle or Sybase the respective JDBC driver must first be deployed to the target Partition. To deploy a library to multiple partitions do the following:

1. From the Console, select Wizard->Deployment Wizard. This will open the Deployment Wizard.
2. Click on the Add button and navigate to the library file in the resulting window and click OK. The library name should now appear in the selection box in the Deployment Wizard.
3. Click on the Next button. The names of the partitions will appear in the Deployment Wizard window.
4. Select the partitions to which you want to deploy the library and click on the Finish button. The deployment status will appear in a separate window.
5. Click the Close button to close this window. You can verify that the libraries are deployed correctly by checking the partition's Deployed Modules node in the Management Console navigation pane. The name of the library should appear under the Deployed Modules node.
6. Stop and restart the partition for the deployment to take effect.

To deploy the library to a single partition:

1. Right-click on the partition's name in the navigation pane in the Management Console and select Deploy Modules from the context menu. The Deployment Wizard will open.
2. Click on the Add button and navigate to the library file in the resulting window and click OK. The library name should now appear in the selection box in the Deployment Wizard.
3. Click on the Next button. The partition name will appear in the Deployment Wizard window.
4. Select the partition to which you want to deploy the library and click on the Finish button. The deployment status will appear in a separate window.
5. Click the Close button to close this window. You can verify that the libraries are deployed correctly by checking the partition's Deployed Modules node in the Management Console navigation pane. The name of the library should appear under the Deployed Modules node.
6. Stop and restart the partition for the deployment to take effect.

Defining the Connection Pool Properties for a JDBC Datasource

At runtime each JDBC datasource corresponds to an instance of a connection pool. Connection pools provide for the reuse of connections and optimization of database connectivity. Some datasources may require different treatment as connection pools than others. A number of configuration options exist for these connection pools. Control of pool sizes, statement execution behavior, and transaction parameters are specified as properties in the `<visitransact-datasource>` element in the DAR descriptor file. You specify properties using the `<property>` element, which includes the `<prop-name>`, `<prop-type>`, and `<prop-value>` elements. The complete list of properties, allowed values, defaults, and descriptions appear in the following table:

Name	Allowed Values	Description	Default Value
`FinalizeNoTxBusy Connections`	This property differs from the rest of the Connection Pool properties because it must be set in the BAS Partition by editing the `partition_server.config` to add the following line: `vmpram -DFinalizeNoTxBusyConnections` When set, it will impact all BAS JDBC Connection Pools active in that partition.	When a JDBC Connection enters a state of NoTxBusy, the application must close the connection, otherwise the JDBC Connection Pool will hold onto a reference of it indefinitely and the underlying database connection is never released. This property, when set, configures BES connection pools to hold a weak reference for JDBC connections, allowing out of scope NoTxBusy connections, not closed by applications, to be freed when JVM garbage collection occurs.	`connectionType`

Name	Allowed Values	Description	Default Value
`Enumerated`: ■ `Direct` ■ `XA`	Indicates type transaction association of all connections retrieved from the connection pool, whether “Direct” or “XA”	Not Applicable. Property specification is mandatory	`optimizeXA`
`Boolean`	By default, XAResource API calls are kept to a minimum for optimization of AppServer JDBC connection pool performance. Setting `optimizeXA` to a value of `false` disables this optimization. Under certain conditions, datasources must have optimizeXA property set to false. For instance, conflicts can arise between default XAResource optimizations in AppServer JDBC connection pool and certain vendor resource managers, such as Oracle, during two-phase commit protocol. When setting `optimizeXA` to `false`, applications using a JDBC connection in the scope of a distributed transaction must issue a connection `close()` before completion of the transaction, otherwise unexpected transaction completion conditions will arise.	`True`	`maxPoolSize`
`Integer`	Specifies the maximum number of database connections that can be obtained from this datasource connection pool.	`0` (zero), implying unbounded size	`waitTimeout`
`Integer`	The number of seconds to wait for a free connection when `maxPoolSize` connections are already opened. When using the `maxPoolSize` property and the pool is at its max and can't serve any more connections, the threads looking for JDBC connections end up waiting for the connection(s) to become available for a long time if the wait time is unbounded (set to 0 seconds). You can set the `waitTimeout` period to suit your needs.	`30`	`busyTimeout`
`Integer`	The number of seconds to wait before a busy connection is released	`600` (ten minutes)	`idleTimeout`
`Integer`	A pooled connection remaining in an idle state for a period of time longer than this timeout value should be closed. All idle connections are checked for `idleTimeout` expiration every 60 seconds. The value of the `idleTimeout` is given in seconds.	`600` (ten minutes)	`queryTimeout`
`Integer`	Specifies in seconds the time limit for executing database queries by this datasource.	`0` (zero), implying indefinite period	`dialect`

Name	Allowed Values	Description	Default Value
isolationLevel	Enumerated: ■ oracle ■ sybase ■ interbase ■ jdatastore	Specifies the database vendor as a hint for automatic table creation performed during Container Managed Persistence	This property is optional. There is no default value.
reuseStatements	Enumerated: ■ TRANSACTION_NONE ■ TRANSACTION_READ_COMMITTED ■ TRANSACTION_READ_UNCOMMITTED ■ TRANSACTION_REPEATABLE_READ ■ TRANSACTION_SERIALIZEABLE	Indicates database isolation level associated with all connections opened by this datasource's connection pool. See the J2EE 1.3 specification for details on these isolation levels.	Defaults to whatever level is provided by the JDBC driver vendor.
initSQL	Boolean	Optimization directive requesting prepared SQL statements to be cached for reuse. Applies to all connections obtained from the connection pool.	True
refreshFrequency	String	Specifies a list of ";" separated SQL statements to be executed each time a connection is obtained for a fresh transaction. The SQL is performed before any application work is performed on the connection.	This property is optional. There is no default value.
dbPingSQL	Integer	Using dbPingSQL, this property specifies a timeout, in seconds, for each connection in an idle state. Once the timeout expires, the connection is examined to determine if it is still a valid connection. All idle connections are checked for refreshFrequency at sixty second intervals.	300 (five minutes)
resSharingScope	String	Specifies an SQL statement used to validate open connections present in the connection pool and to refresh connections during a refreshFrequency timeout.	Not defined. When no SQL is specified, the container uses java.sql.ConnectionisClosed() method to validate the connection.
maxPreparedStatement CacheSize	Enumerated: ■ Shareable ■ Unshareable	Indicates whether connection statements and result sets are cached for reuse. If set to Shareable, connection statements and results sets are cached, thereby optimizing throughput of connections. If set to Unshareable, connections are closed once the application closes the connection.	Shareable

Name	Allowed Values	Description	Default Value
Integer	Each connection within an AppServer JDBC pool caches java.sql.PreparedStatement objects for reuse. Each PreparedStatement cache is organized by SQL literal strings representing unique SQL statement requests. This property limits the number of PreparedStatements cached per pooled JDBC connection. It specifies the maximum size of the cache. If a cache reaches the limit, any subsequent javax.sql.Connection.prepareStatement() calls result in non-cached instances of PreparedStatement objects being created and returned to the caller. The lifecycle of the cache is the same as the JDBC connection lifecycle. For example, if an idle connection times out, both the connection and its PreparedStatement cache are discarded. Unresolved parameterized SQL statements are cached, for example, the statement SELECT NAME FROM CUSTOMER WHERE AGE=20 is cached as SELECT NAME FROM CUSTOMER WHERE :age='?'. Note that this property is only effective when the reuseStatements property of the datasource is set to true (default). The default value is 40, which is usually sufficient for applications.	40	maxPreparedStatementsPerQuery
Integer	Under certain conditions such as high concurrency or when CMP 2.0 entity beans are processed, more than one PreparedStatement can be processed concurrently for the same SQL query on the same pooled connection. For example, a SQL query SELECT name FROM table1 WHERE id=? can return distinct result sets when different values are used for ?. Although the PreparedStatement cache has a single entry for each SQL query, two or more PreparedStatements can exist in the cache for the query. This property specifies the maximum number of cached PreparedStatements for a single query. If the limit is exceeded for a particular query, subsequent javax.sql.Connection.prepareStatement() calls result in non-cached instances of PreparedStatement objects created and returned to the caller. Like maxPreparedStatementCacheSize, this property is only effective when the reuseStatements property of the datasource is set to true (default).	20	

Getting debug output

A number of system properties can be set to log activity at datasource, connection pool, connection and statement levels during application processing. It is not necessary to configure these properties during normal application runtime execution but should a situation arise where details of JDBC flow of control is needed these options are useful. Runtime output generated with these properties set can be provided to Borland Technical Support to help resolve issues involving JDBC datasource and connections. Setting these properties for a partition results in log message generation during JDBC activity. Note that additional `log4j` configuration is required to ensure that the messages are actually written to the partition log. Locate the partition's `log4j` configuration file, called `logConfiguration.xml`, and add the following <logger> element:

```
⋮
<log4j:configuration>
 ⋮
    <logger name="com.inprise.visitransact.jdbc2" additivity="true">
       <level value="DEBUG" />
    </logger>
    ⋮
</log4j:configuration>
```

Note BAS logging is based on the Log4j infrastructure. Some user applications which use Log4j may cause the Partition to hang. User applications should use the per-partition log4j Configuration file, rather than deploying a Configuration file in the archive. By default, this file is located under the Partition's Managed Object footprint at: `<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<partition_name>/adm/properties/logConfiguration.xml`. Alternately, you can uncomment the following line in the `<install_dir>/bin/partition.config` file:

```
vmprop borland.enterprise.server.partition.disableSystemRedirect=true
```

System Property Name	Type	Description	Default
`DataSourceDebug`	`Boolean`	Reports activity at datasource level for all datasources	`False`
`ConnectionPoolDebug`	`Boolean`	Reports activity at connection pool level for all datasources	`False`
`ConnetionPoolStateDebug`	`Boolean`	Reports transitions of connections in connection pool	`False`
`JDBCProxyDebug`	`Boolean`	Reports activity at connection level for all connections	`False`
`PreparedStatementCacheDebug`	`Boolean`	Reports activity at prepared statement level for all statements	`False`

Descriptions of AppServer's Pooled Connection States

When the EJB container's statistic gathering option is enabled, the Partition event log contains useful statistics about the JDBC connections pool. The log lists the number of connections in the various lifecycle states of a pooled JDBC2 connection. Following is a description of each state:

- **Free:** a cached/pooled connection that is available for use by an application
- **TxBusy:** a cached connection that is in use in a transaction
- **NoTxBusy:** a cached connection that is in use by an application with no transaction context
- **Committed:** a connection that was associated with a transaction received a `commit()` call from the transaction service
- **RolledBack:** a connection that was associated with a transaction received a `rollback()` call from the transaction service
- **Prepared:** a connection that was associated with a transaction received a `prepare()` call from the transaction service
- **Forgot:** a connection that was associated with a transaction received a `forget()` call from the transaction service
- **TxBusyXaStart:** a pooled connection that is associated with a transaction branch.
- **TxBusyXaEnd:** a pooled connection that has finished its association with a transaction branch
- **BusyTimedOut:** a cached connection that was removed from the pool after it stayed with the transaction longer than the `busyTimeout` pool property
- **IdleTimedOut:** a connection that was removed from the pool due to being idle for longer than the pool's `idleTimeout` property
- **JdbcHalfCompleted:** a transitionary state where the connection is participating in a background housekeeping activity related to pool management (being refreshed, for example) and therefore, unavailable until the activity completes
- **Closed:** the underlying JDBC connection was closed
- **Discarded:** A cached connection got discarded (due to timeout errors, for example)
- **JdbcFinalized:** an unreferenced connection was garbage collected

Support for older JDBC 1.x drivers

JDBC 1x drivers do not provide a datasource object. Under the J2EE specification, however, database connections are always fetched using the `javax.sql.DataSource` interface. To allow users to still use JDBC 1x drivers, AppServer provides an implementation of a JDBC 1x datasource to allow writing portable J2EE code. This implementation is a facade provided on top of the DriverManager connection mechanism of the JDBC 1x specification.

If you want to define a datasource on top of such a driver then in the DD Editor choose the Datasource Type field as “Other(JDBC1x)”. Then in the Main panel you can input the Driver Manager classname and connection URL for your particular database and driver.

The class name `com.inprise.visitransact.jdbc1w2.InpriseConnectionPoolDataSource` is not the `DriverManager` class of the JDBC driver; it is a wrapper class. The vendor's class should be specified in the Driver Class Name text box of the editor panel.

Advanced Topics for Defining JDBC Datasources

Whether you choose to use the server's graphical tools or not, defining a datasource means providing some information to the container in XML format. Let's look at what it takes to define a JDBC datasource and bind it to JNDI. Let's start by examining the DTD of the `jndi-definitions.xml` file. The elements in bold are the main elements specific to JDBC datasources.

```
<!ELEMENT jndi-definitions (visitransact-datasource*, driver-datasource*, jndi-
object*)>
<!ELEMENT visitransact-datasource (jndi-name, driver-datasource-jndiname,
property*)>
<!ELEMENT driver-datasource (jndi-name, datasource-class-name,
log-writer?, property* )>
<!ELEMENT jndi-object (jndi-name, class-name, property* )>
<!ELEMENT property (prop-name, prop-type, prop-value)>
  <!ELEMENT prop-name (#PCDATA)>
  <!ELEMENT prop-type (#PCDATA)>
  <!ELEMENT prop-value (#PCDATA)>
  <!ELEMENT jndi-name (#PCDATA)>
  <!ELEMENT driver-datasource-jndiname (#PCDATA)>
  <!ELEMENT datasource-class-name (#PCDATA)>
  <!ELEMENT log-writer (#PCDATA)>
  <!ELEMENT class-name (#PCDATA)>
```

Defining a JDBC datasource involves two XML elements. The first is the `<visitransact-datasource>` element. This is where you define the datasource your application code will look up. You include the following information:

- **jndi-name:** this is the name of the datasource as it will be referenced by JNDI. It is also the name found in the resource references of your enterprise beans.
- **driver-datasource-jndiname:** this is the JNDI name of the driver class supplied by the database or JMS vendor that you deployed as a library to your Partitions. It is also the name that will be referenced by the `<driver-datasource>` element discussed next.
- **properties:** these are the properties for the datasource's role in its connection pool. We'll discuss these properties in a little more detail in the Defining the Connection Pool Properties for a JDBC Datasource section.

So, let's look at an example of this portion of the datasource definition in the XML. In the following example, we'll look at an example using Oracle:

```
<jndi-definitions>
        <visitransact-datasource>
            <jndi-name>datasources/Oracle</jndi-name>
            <driver-datasource-jndiname>datasources/OracleDriver
              </driver-datasource-jndiname>
            <property>
               <prop-name>connectionType</prop-name>
               <prop-type>Enumerated</prop-type>
               <prop-value>Direct</prop-value>
            </property>
            ⋮
            <!-- other properties as needed -->
            ⋮
        </visitransact-datasource>
        ⋮
    </jndi-definitions>
```

We're not done. Now we must perform the other half of the datasource definition by providing information on the driver. We do this in the <driver-datasource> element, which includes the following information:

- **jndi-name:** This is the JNDI name of the driver class, and its value *must* be identical to the <driver-datasource-jndiname> value from the <visitransact-datasource> element.
- **datasource-class-name:** Here is where you provide the name of the connection factory class supplied from the resource vendor. It must be the same class you deployed to the Partition as a library.
- **log-writer:** This is a boolean element that activates verbose modes for some vendor connection factory classes. Consult your resource's documentation for the use of this property.
- **properties:** These are properties specific to the JDBC resource, such as usernames, passwords, and so forth. These properties are passed to the driver class for processing. Consult your JDBC resource documentation for property information. Specifying the properties in XML is shown below.

Armed with this information, let's complete our datasource definition for the Oracle datasource we started above. In order to be thorough, let's first reproduce the XML we started above:

```
<jndi-definitions>
 <visitransact-datasource>
       <jndi-name>datasources/Oracle</jndi-name>
       <driver-datasource-jndiname>datasources/OracleDriver
          </driver-datasource-jndiname>
     <log-writer>False</log-writer>
     <property>
          <prop-name>connectionType</prop-name>
          <prop-type>Enumerated</prop-type>
          <prop-value>Direct</prop-value>
     </property>
 </visitransact-datasource>
    ⋮
```

Note the driver datasource JNDI name in bold. Now we'll add the following:

```
    <driver-datasource>
 <jndi-name>datasources/OracleDriver</jndi-name>
 <datasource-class-name>oracle.jdbc.pool.OracleConnectionPoolDataSource</
datasource-class-name>
 <property>
   <prop-name>user</prop-name>
   <prop-type>String</prop-type>
   <prop-value>MisterKittles</prop-value>
 </property>
 <property>
   <prop-name>password</prop-name>
   <prop-type>String</prop-type>
   <prop-value>Mittens</prop-value>
 </property>
  ⋮
  // other properties as needed
  ⋮
    </driver-datasource>
</jndi-definitions>
```

Now the JDBC datasource is fully defined. Once you've packaged the XML file as a DAR, you can deploy it to a Partition. Doing so registers the datasource with the Naming Service and makes it available for lookup.

Connecting to JDBC Resources from J2EE Application Components

In Borland proprietary deployment descriptors, such as `ejb-borland.xml` for EJB components, the `<resource-ref>` element is used to map a datasource logical name to actual JNDI location of a JDBC datasource definition. Mapping of the logical name to its location occurs when a JNDI lookup is performed for a desired datasource in the application component. You use the element within your individual component definitions. For example, a `<resource-ref>` for an entity bean must be found within the `<entity>` tags. Let's look at the DTD representation of the `<resource-ref>` element of Borland deployment descriptors:

```
<!ELEMENT resource-ref (res-ref-name, jndi-name, cmp-resource?)>
```

In this element you specify the following:

- **res-ref-name:** this is the logical name for the resource, the same logical name you use in the `<resource-ref>` element of the standard `ejb-jar.xml` descriptor file. This is the name your application components use to look up the datasource.
- **jndi-name:** this is the JNDI name of the datasource that will be bound to its logical name. It must match the value of the corresponding `<jndi-name>` element of the `<visitransact-datasource>` element deployed with the DAR.
- **cmp-resource:** this is an optional boolean element that is relevant to entity beans only. If set to `True`, the container's CMP engine will monitor this datasource.

Let's look at an example entity bean that uses the Oracle datasource we defined above:

```
<entity>
 <ejb-name>entity_bean</ejb-name>
 ⋮
    <resource-ref>
  <res-ref-name>jdbc/MyDataSource</res-ref-name>
  <jndi-name>datasources/Oracle</jndi-name>
  <cmp-resource>True</cmp-resource>
    </resource-ref>
 ⋮
</entity>
```

As you can see, we used the identical JNDI name from the `<visitransact-datasource>` element from the datasource definition. Now let's see how we obtain a datasource object reference. To do so, the application performs a lookup of the `<res-ref-name>` value of the deployed components and the object references are retrieved from the remote CosNaming provider. For example:

```
javax.sql.DataSource ds1;

try {
 javax.naming.Context ctx = (javax.naming.Context)
    new javax.naming.InitialContext();
 ds1 = (DataSource)ctx.lookup("java:comp/env/jdbc/MyDataSource");
}
catch (javax.naming.NamingException exp) {
 exp.printStackTrace();
}
```

A database `java.sql.Connection` can now be obtained from `ds1`.

Chapter

22

Using JMS

Resource related objects such as JMS connection factories and JMS Queue/Topic destinations are obtained in a portable J2EE mandated way through JNDI. A JMS resource object is resolved by performing a JNDI lookup of a J2EE Resource Reference defined in the deployment descriptors of an application component. Resource Reference definitions involve both standard J2EE and Borland's proprietary deployment descriptors. In the standard deployment descriptor, a Resource Reference specifies a logical name relative to the application's JNDI environment naming context, `java:comp/env/`. Borland's deployment descriptor complements the standard descriptor by associating the Resource Reference logical name with the actual JNDI location of the JMS resource definition. For example, in an EJB Jar component, the standard J2EE deployment descriptor, `ejb-jar.xml`, specifies Resource References for an EJB using a `<resource-ref>` element for a JMS connection factory and `<resource-env-ref>` elements for JMS Topics and Queues. In Borland AppServer(AppServer), a JNDI lookup of a Resource Reference involves retrieval of the JMS resource definition from which the desired JMS object is created and returned to caller of lookup. The property values present in the JMS resource definition determine the type and characteristics of resource object created.

Before a Resource Reference look up can be attempted, the required resource definition must first be bound to its physical JNDI location. In the AppServer, JMS resource definitions are bound into a JNDI service provider during deployment of a Definitions ARchive (DAR) module. By default, these objects are bound to the partitions' Naming Service, the JNDI CosNaming service provider in AppServer. This chapter describes how to define JMS resource object definitions in a DAR module and delves into the details of how to get a handle to a JMS resource object from a J2EE application. A discussion of JMS activity and how it relates transactions is also provided.

A DAR contains the JNDI definitions module (`jndi-definitions.xml` file) which contains properties for each resource related object that you want to bind to a JNDI provider (Naming Service). When an application EAR is deployed, the contents of the DAR file get deployed in the Naming Service of a partition. The properties of the resource related object defined in the `jndi-definitions.xml` file are stored in a JNDI bound object in the partition-hosted Naming Service.

When an application client or an EJB component does a JNDI lookup for a resource related object, it calls a `lookup()` method which communicates with the JNDI provider:

1 The Application Client refers to the `<resource-ref>` element in the standard deployment descriptor (in the case of EJBs it is `ejb-jar.xml`) to get the logical name of the resource. (It does a lookup in the component's local namespace, `java:comp/env`, to obtain the logical name of the object.) This logical name is specified in its `<resource-ref-name>` sub element. For example, in `ejb-jar.xml`:

```
⋮
 <description>This example demonstrates JMS XA and JDBC XA in a two-phase
commit transaction.</description>
 <enterprise-beans>
  <session>
   ⋮
     <resource-ref>
        <description />
        <res-ref-name>jms/insurance/ConnectionFactory</res-ref-name>
        <res-type>javax.jms.ConnectionFactory</res-type>
        <res-auth>Container</res-auth>
     </resource-ref>
     ⋮
  </session>
```

2 Using this logical name, the Container obtains the actual JNDI location of the JMS resource definition (a JNDI bound object) from the Borland proprietary deployment descriptor, `ejb-borland.xml`:

```
⋮
 <enterprise-beans>
  <session>
  ⋮
     <resource-ref>
        <res-ref-name>jms/insurance/ConnectionFactory</res-ref-name>
        <jndi-name>jms/xacf</jndi-name>
     </resource-ref>
  </session>
```

3 The Container then creates an instance of the resource object by using the stored property values in the bound object. The following properties were stored in the ConnectionFactory object from the `jndi-definitions.xml` file when the related DAR was deployed:

```
⋮
<jndi-definitions>
  <jndi-object>
     <jndi-name>jms/xacf</jndi-name>
     <classname>com.tibco.tibjms.TibjmsXAConnectionFactory</class-name>
     <property>
        <prop-name>serverUrl</prop-name>
        <prop-type>String</prop-type>
        <prop-value>localhost:7222</prop-value>
     </property>
  </jndi-object>
```

4 This instance is then wrapped in the Borland proprietary API (in the JMS proxy layer) by the container and passed back to the caller (could be an application client or could be another J2EE component) of the `lookup()`.

JMS 1.1 Common APIs

With JMS 1.1, JMS client applications have the option to use domain-independent unified APIs. A client can obtain a handle to a generic JMS ConnectionFactory, and from it a generic Session object that can be used with either a Queue or Topic for message processing. The common APIs along with their domain specific APIs are listed in the table below:

JMS Common APIs (in JMS 1.1)	Point-to-Point Domain APIs	Publish/Subscribe Domain
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber
XAConnectionFactory	XAQueueConnectionFactory	XATopicConnectionFactory
XAConnection	XAQueueConnection	XATopicConnection
XASession	XAQueueSession	XATopicSession

The major change when using common interfaces is that one or more Queue and/or Topic destinations can now be simultaneously accessed through the same session all within the same transaction. With this change, either all the messages in a single transaction (messages to/from the queue(s) and the topic(s)) get sent and the transaction is considered successful or the whole transaction is aborted and none of the messages are delivered.

Borland AppServer supports the domain-independent APIs of JMS 1.1, and the constraints imposed by J2EE 1.4 in use of all JMS 1.1 APIs.

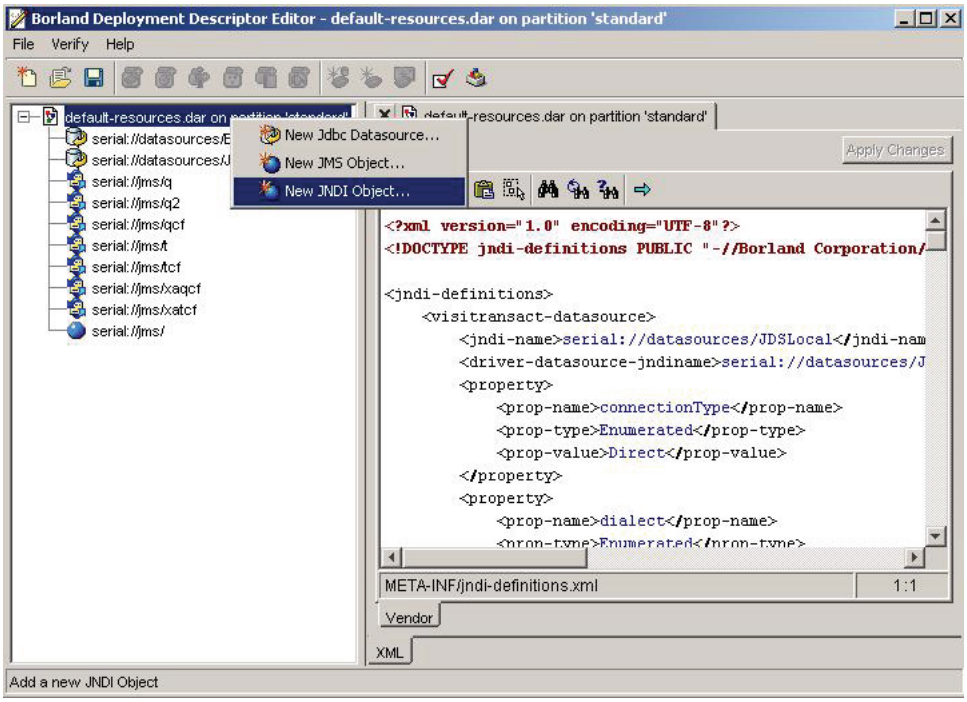
Configuring JMS Connection Factories and Destinations

Using the Management Console, navigate to the “Deployed Modules” list in the Partition whose JMS resource objects you need to configure. By default, every Partition has a predeployed JNDI Definitions Module (DAR) called `default-resources.dar`. Right-click on the module and choose “Edit deployment descriptor” from the context menu. The Deployment Descriptor Editor (DDEditor) opens.

In the Navigation Pane of the DDEditor is a list of JMS connection factories, and queues/topics preconfigured in the product. Click on the connection factory name. The right pane will display the properties for it. For each connection factory, you can choose Tibco, Sonic, WMQ or another (“Other”) JMS provider. The DDEditor has knowledge of Tibco and auto fills the class names for each. You can also choose the object resource type from the JMS Object type drop-down list. For OpenJMS, you must edit the `openjms.xml` file to configure connection factories and destination. See “Configuring JNDI objects for OpenJMS” on page 222 for details on how to access this file.

If you selected “Other” from the JMS Provider list, look up the JMS vendor's documentation to ascertain the correct name of its connection factory, topic, or queue implementation class. In addition, the Main panel will not suggest any properties to fill in and you will need to use the Properties tab to set any appropriate properties.

To create a new JMS object, right-click the root node in the navigation pane and select “New JMS Object” from the Context Menu.



A dialog box prompts for JNDI name of JMS object to be created. By default, the name specified corresponds to a location in the Naming Service. If a JNDI name is specified with a “serial//” prefix, the remaining name following the prefix corresponds to a location in the Naming Service. Once given, a representation of this JMS object appears in the tree in the Navigation Pane. Click the representation to open its configuration panel.

The DDEditor has knowledge of Tibco, Sonic, WMQ and can auto fill the class names for it.

Note The Main panel will not suggest any JMS objects other than Tibco, Sonic and WMQ. You need to use the Properties tab to set any appropriate properties.

When you're finished, choose “File|Save...” and the module will be saved back to the Partition and redeployed.

Defining Connection Pool Properties for JMS Connection Factories

Each JMS connection factory defined in AppServer has an associated connection pool. You can specify connection pool properties for each JMS connection factory defined in the `jndi-definitions.xml` file of a DAR module. The AppServer partition system properties can be specified to dictate the default behavior for all JMS connection pools established in a partition's Java virtual machine. However, properties defined for individual JMS connection factories in the `jndi-definitions.xml` file override system property values.

The AppServer partition system properties used for default configuration of JMS connection pools are listed in the table below :

Property Name	Description	Type	Values
JMSConnectionMaxPoolSize	Maximum number of JMS connections allowed for a JMS connection pool	Integer	0<n, where n is the maximum number of connections allowed for a JMS connection pool. The default is 0, indicating an unlimited number
JMSConnectionWaitTimeout	Time allowed to wait for a free connection in the JMS connection pool	Integer	Default is 30 seconds
JMSConnectionIdleTimeout	Time allowed for connection to remain idle in the JMS connection pool before being discarded	Integer	Default is 60 seconds
JMSConnectionPoolDebug	Enable display of debug messages associated with AppServer JMS connection pooling	Boolean	Default value is true true – Enable Debug false – Disable debug
JMSConnectionPoolDisable	Controls use of AppServer connection pooling for JMS Connection factories.	Boolean	Default value is false false – Enable JMS connection true – Disable JMS connection
JMSConnectionPoolMonitorLevel	Enable JMS pool monitoring for AppServer JMS connection and session pools. **Important:** Each JMS connection can have a set of JMS sessions created, which are maintained in a JMS session pool. Consider the impact on the performance of the connection when you set a value for this property, because each level corresponds to an increased degree of maintenance and collection of values for sets of counters and states. If you choose “maximum”, it will have the greatest performance impact on the connection.	String	■ “none” (Default) ■ “minimum” ■ “medium” ■ “maximum”
JMSSessionMaxPoolSize	Maximum number of JMS sessions for each JMS session pool of a connection factory	Integer	0<n, where n is the maximum number of JMS sessions allowed for a JMS session pool associated with a JMS connection. The default is 0, indicating an unlimited number
JMSSessionWaitTimeout	Time allowed to wait for a free session in a JMS session pool	Integer	Default is 30 seconds
JMSSessionPoolDisable	Controls use of AppServer session pooling per JMS connection. When a JMS connection factory is looked up under JNDI, if the value of this property is true the vendor JMS connection factory is returned. It is not wrapped by any AppServer proxy class.	Boolean	Default value set to false false – Enable JMS session true – Disable JMS session
JMSSessionPoolDebug	Enable display of debug messages associated with AppServer JMS session pooling	Boolean	Default value is false false – Disable debug true – Enable debug

Defining Individual JMS Connection Factory Properties

You can define JMS pool properties for individual connection factories in `jndi-definitions.xml` file. These properties override partition system properties. Use the `<property>` element to add a pool property. For example:

```
<jndi-definitions>
   <!-- *************************** -->
   <!-- * JMS Connection Factories * -->
   <!-- *************************** -->
       <jndi-object>
           <jndi-name>jms/cf</jndi-name>
           <class-name>com.tibco.tibjms.TibjmsConnectionFactory</class-name>
           <property>
               <prop-name>serverUrl</prop-name>
               <prop-type>String</prop-type>
               <prop-value>localhost:7222</prop-value>
           </property>
           <property>
               <prop-name>besConnectionPoolMaxPoolSize</prop-name>
               <prop-type>Integer</prop-type>
               <prop-value>11</prop-value>
           </property>
           <property>
               <prop-name>besConnectionPoolDebug</prop-name>
               <prop-type>Boolean</prop-type>
               <prop-value>true</prop-value>
           </property>
           <property>
               <prop-name>besSessionPoolDisable</prop-name>
               <prop-type>Boolean</prop-type>
               <prop-value>true</prop-value>
           </property>
       </jndi-object>
       ⋮
   </jndi-definitions>
```

The full set of JMS Connection factory pool properties are listed below together with the corresponding system property that each overrides:

Individual Pool Property	Associated System Property
besConnectionPoolMaxPoolSize	JMSConnectionMaxPoolSize
besConnectionPoolWaitTimeout	JMSConnectionWaitTimeout
besConnectionPoolIdleTimeout	JMSConnectionIdleTimeout
besConnectionPoolMonitorLevel	JMSConnectionPoolMonitorLevel
besConnectionPoolDisable	JMSConnectionPoolDisable
besConnectionPoolDebug	JMSConnectionPoolDebug
besSessionPoolMaxPoolSize	JMSSessionMaxPoolSize
besSessionPoolWaitTimeout	JMSSessionWaitTimeout
besSessionPoolDisable	JMSSessionPoolDisable
besSessionPoolDebug	JMSSessionPoolDebug

Obtaining JMS Connection Factories and Destinations in J2EE Application Components

A JMS connection factory object is obtained in much the same way as a JDBC datasource object. The factory object is declared in a <resource-ref> element of both the standard J2EE and Borland-specific deployment descriptors. However, extra configuration is required if an application needs to interact with destinations of a JMS provider. A <resource-env-ref> element must be specified in both descriptors with definition of at least one JMS destination, that is a target queue or topic on which messages can be produced/consumed. While the standard J2EE deployment descriptor provides the logical name and type of a JMS connection factory and a JMS destination, the Borland specific deployment descriptor maps the logical name to a reference of the actual target object, resolved through JNDI lookup.

J2EE 1.2 and J2EE 1.3

Details of <resource-ref> and <resource-env-ref> elements for standard J2EE deployment descriptors are described in J2EE 1.3 specifications. These elements apply to all application components, for instance EJBs, Servlets and application clients, that wish to use JMS APIs. Similarly, corresponding <resource-ref> and <resource-env-ref> elements exist in accompanying Borland-specific deployment descriptors. Let's look at deployment descriptors for an EJB session bean that uses JMS. First, from standard EJB descriptor, ejb-jar.xml:

```
⋮
<session>
 <ejb-name>session_bean</ejb-name>
 ⋮
 <resource-ref>
     <res-ref-name>jms/MyJMSQueueConnectionFactory</res-ref-name>
  <res-type>javax.jms.QueueConnectionFactory</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
 </resource-ref>
 <resource-env-ref>
  <res-env-ref-name>jms/MyJMSQueue</res-env-ref-name>
  <res-env-ref-type>javax.jms.Queue</res-env-ref-type>
 </resource-env-ref>
 ⋮
</session>
```

The portable descriptor above defines logical names for a JMS connection factory and a JMS Queue through <resource-ref> and <resource-env-ref> respectively. In Borland's proprietary deployment descriptors, such as ejb-borland.xml for EJB components, the <resource-ref> element is used to resolve the logical name to actual JNDI location of a JMS connection factory definition when a JNDI lookup is performed for the desired connection factory in the application component. This element is used within descriptor definitions of individual components. For example, a <resource-ref> for an entity bean must be found within the <entity> tags. Let's examine the DTD representation of the <resource-ref> element for J2EE 1.2 and 1.3 Borland deployment descriptors:

```
<!ELEMENT resource-ref (res-ref-name, jndi-name, cmp-resource?)>
```

In this element you specify the following:

- **res-ref-name**: this is the logical name for the resource object, the same logical name you use in the <resource-ref> element of the standard ejb-jar.xml descriptor file. This is the name your application components use to look up the JMS connection factory.
- **jndi-name**: this is the JNDI name of the connection factory that will be bound to the logical name. It must match the value of the corresponding <jndi-name> element of the <jndi-object> element in the deployed DAR where the connection factory is defined.

Just as the <resource-ref> element is used to map logical names for JMS connection factories to actual JNDI location of desired connection factory definition, the <resource-env-ref> element maps the logical name for JMS destinations, such as Queues and Topics, to actual JNDI location of destination definition. The DTD representation of this element for Borland deployment descriptors is as follows:

```
<!ELEMENT resource-env-ref (resource-env-ref-name, jndi-name)>
```

Two elements are specified:

- **resource-env-ref-name**: this is the logical name of the Topic or Queue, and its value must be identical to the value of the <res-env-ref-name> of J2EE standard descriptor.
- **jndi-name**: this is the JNDI name of the topic or queue that resolves the logical name.

The final contents for Borland descriptor ejb-borland.xml accompanying the ejb-jar.xml defined above is:

```
<session>
 <ejb-name>session_bean</ejb-name>
 ⋮
 <resource-ref>
  <res-ref-name>jms/MyJMSQueueConnectionFactory</res-ref-name>
  <jndi-name>resources/qcf</jndi-name>
 </resource-ref>
 <resource-env-ref>
  <resource-env-ref-name>jms/MyJMSQueue</resource-env-ref-name>
  <jndi-name>resources/q</jndi-name>
 </resource-env-ref>
 ⋮
</session>
```

Keep in mind that <resource-ref> and <resource-env-ref> elements can be used for all J2EE components that require JMS related resource objects. For instance, application clients using JMS APIs should obtain connection factories and destinations the same way as EJBs, through JNDI lookup or Resource References in application code and specification of <resource-ref> and <resource-env-ref> elements in the clients deployment descriptors. For example, in the J2EE standard descriptor, application-client.xml:

```
<application-client>
⋮
<resource-ref>
  <res-ref-name>jms/MyJMSTopicConnectionFactory</res-ref-name>
  <res-type>javax.jms.TopicConnectionFactory</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
<resource-env-ref>
  <res-env-ref-name>jms/MyJMSTopic</res-env-ref-name>
  <res-env-ref-type>javax.jms.Topic</res-env-ref-type>
</resource-env-ref>
⋮
  </application-client>
```

and its accompanying Borland descriptor `application-client-borland.xml`:

```
 <application-client>
⋮
<resource-ref>
 <res-ref-name>jms/MyJMSTopicConnectionFactory</res-ref-name>
 <jndi-name>resources/tcf</jndi-name>
</resource-ref>
<resource-env-ref>
 <resource-env-ref-name>jms/MyJMSTopic</resource-env-ref-name>
 <jndi-name>resources/t</jndi-name>
</resource-env-ref>
⋮
 </application-client>
```

Now let's see how we obtain object references to a JMS connection factory and a destination in the application logic. In retrieval of the connection factory, the application performs a JNDI lookup of the `<res-ref-name>` value from `<resource-ref>` element in the J2EE deployment descriptor. To retrieve the destination object, a JNDI lookup is performed against the `<res-env-ref-name>` value of `<resource-env-ref>` element in the J2EE deployment descriptor. The names specified for `<jndi-name>` are identical to JNDI names in `<jndi-object>` elements of the JMS resource definitions in a deployed DAR module. When a lookup succeeds a JMS resource object is obtained, that is, for JMS connection factory identified through logical name `jms/MyJMSTopicConnectionFactory` a deployed JMS definition object is retrieved from the Naming Service under `resources/tcf` and from it a connection factory object is created and returned to the application.

For example, the application client code associated with client descriptors provided above resolves JMS resource objects as follows:

```
javax.jms.TopicConnectionFactory myTCF;
javax.jms.Topic myTopic;
try {
 javax.naming.Context ctx = (javax.naming.Context) new
javax.naming.InitialContext();
  myTCF = (TopicConnectionFactory) ctx.lookup("java:comp/env/jms/
MyJMSTopicConnectionFactory");
  // Now ready to obtain a connection from myTCF
  myTopic = (Topic) ctx.lookup("java:comp/env/jms/MyJMSTopic");
  ⋮
 }
 catch (javax.naming.NamingException exp) {
  exp.printStackTrace();
 }
```

J2EE 1.4

In earlier versions of J2EE, each application component had to declare an `<resource-env-ref>` in the standard deployment descriptor for look up of a JMS destination from its own local namespace. If separate application components have references to the same destination, there was no way for a deployer to know that these `<resource-env-ref>`s should be bound to the same destination.

Here is an example which uses the `<resource-env-ref>` to define the same JMS destination from two separate application components, Session beans, in this case:

```
⋮
<ejb-jar ... >
   <enterprise-beans>
         <session>
            <ejb-name>SenderEJB</ejb-name>
            ⋮
```

```
      <resource-ref>
         <res-ref-name>jms/ConnectionFactory</res-ref-name>
         <res-type>javax.jms.ConnectionFactory</res-type>
         <res-auth>Container</res-auth>
      </resource-ref>
      <resource-env-ref>
         <resource-env-ref-name>jms/LogicalNameA</resource-env-ref-name>
         <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
      </resource-env-ref>
   </session>
   <session>
      <ejb-name>ReceiverEJB</ejb-name>
      ⋮
      <resource-ref>
         <res-ref-name>jms/ConnectionFactory</res-ref-name>
         <res-type>javax.jms.ConnectionFactory</res-type>
         <res-auth>Container</res-auth>
      </resource-ref>
      <resource-env-ref>
         <resource-env-ref-name>jms/LogicalNameB</resource-env-ref-name>
         <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
      </resource-env-ref>
   </session>
```

In J2EE 1.4, although you can continue to use `<resource-env-ref>`, a new element, `<message-destination-ref>` has been introduced for specification of JMS destinations. The same example above can be rewritten using the `<message-destination-ref>` element instead of `<resource-env-ref>` in the standard deployment descriptor, `ejb-jar.xml`, as follows:

```
⋮
<ejb-jar ... >
   <enterprise-beans>
      <session>
         <ejb-name>SenderEJB</ejb-name>
         ⋮
         <resource-ref>
            <res-ref-name>jms/ConnectionFactory</res-ref-name>
            <res-type>javax.jms.ConnectionFactory</res-type>
            <res-auth>Container</res-auth>
         </resource-ref>
         <message-destination-ref>
            <message-destination-ref-name>jms/LogicalNameA
              </message-destination-ref-name>
           <message-destination-type>javax.jms.Queue
             </message-destination-type>
          <message-destination-usage>Produces</message-destination-usage>
           <message-destination-link>MsgQueue1</message-destination-link>
         </message-destination-ref>
      </session>
      <session>
         <ejb-name>ReceiverEJB</ejb-name>
         ⋮
         <resource-ref>
            <res-ref-name>jms/ConnectionFactory</res-ref-name>
            <res-type>javax.jms.ConnectionFactory</res-type>
            <res-auth>Container</res-auth>
         </resource-ref>
         <message-destination-ref>
            <message-destination-ref-name>jms/LogicalNameB
              </message-destination-ref-name>
```

```
                <message-destination-type>javax.jms.Queue
                  </message-destination-type>
                 <message-destination-usage>Consumes</message-destination-usage>
                <message-destination-link>MsgQueue1</message-destination-link>
              </message-destination-ref>
           </session>
        </enterprise-beans>
        <assembly-descriptor>
        <message-destination>
              <message-destination-name>MsgQueue1</message-destination-name>
           </message-destination>
        </assembly-descriptor>
     </ejb-jar>
```

The above example shows two `<message-destination-ref>` elements that refer to the same destination, the Queue MsgQueue1. Each of these `<message-destination-ref>`s have a `<message-destination-link>` element whose value is MsgQueue1. The link elements maps the `<message-destination-ref>`s to a `<message-destination>` element in the `<assembly-descriptor>` resolving the two `<message-destination-ref>`s to the same queue. The ejb-borland has a corresponding `<message-destination>` element within its `<assembly-descriptor>` element. At runtime, the `<message-destination>` element from `ejb-jar.xml` is resolved against the `<jndi-name>` of the `<message-destination>` element specified in the `ejb-borland.xml` descriptor:

```
  ⋮
     <ejb-jar ... >
        <enterprise-beans>
           ⋮
        </enterprise-beans>
        <assembly-descriptor>
        <message-destination>
              <message-destination-name>MsgQueue1</message-destination-name>
              <jndi-name>jms/TibcoQueue1</jndi-name>
           </message-destination>
        </assembly-descriptor>
     </ejb-jar>
```

To show JMS message flow in an application where more than one `<message-destination-ref>`s resolves to the same underlying destination, each one of them should declare a `<message-destination-link>` with a value corresponding to a `<message-destination>` element in the `<assembly-descriptor>`. The value in the `<message-destination-link>` must match the value of the `<message-destination-name>` in the `<message-destination>` element. At runtime, the `<message-destination-ref>`s will resolve to the same destination.

You can also link to a `<message-destination>` defined in a different J2EE module within the same application. For example, `<message-destination-link> ../other/other.jar#destination </message-destination-link>` would link to a <message-destination> with the name destination in the JAR file at the relative path `../other/other.jar`.

You must also specify a `<message-destination>` element in the `ejb-jar.xml` for every destination you use.

Important The JNDI name of a `<message-destination>` in Borland deployment descriptor takes precedence over a specified JNDI name of a `<message-destination-ref>` that has a link to the `<message-destination>` element.

JMS and Transactions

The rules for using JMS APIs in EJB bean code with transactions are discussed in the EJB 2.0 specification section 17.3.5.

Following is an extract:

> 17.3.5 Use of JMS APIs in transactions
>
> The Bean Provider must not make use of the JMS request/reply paradigm (sending of a JMS message, followed by the synchronous receipt of a reply to that message) within a single transaction.
>
> Because a JMS message is not delivered to its final destination until the transaction commits, the receipt of the reply within the same transaction will never take place. Because the container manages the transactional enlistment of JMS sessions on behalf of a bean, the parameters of the createSession(boolean transacted,int acknowledgeMode), createQueueSession(boolean transacted,int acknowledgeMode) and createTopicSession(boolean transacted, int acknowledgeMode) methods are ignored. It is recommended that the Bean Provider specify that a session is transacted, but provide 0 for the value of the acknowledgment mode.
>
> The Bean Provider should not use the JMS acknowledge() method either within a transaction or within an unspecified transaction context. Message acknowledgment in an unspecified transaction context is handled by the container. Section 17.6.5 describes some of the techniques that the container can use for the implementation of a method invocation with an unspecified transaction context.

Avoiding use of the JMS request/reply paradigm and JMS `acknowledge()` method is equally relevant for other J2EE components such as application clients, as it is to EJB bean code. In addition to rules described above, application code should not use any JMS XA APIs. The program should look exactly as if the code is written in a non-transactional JMS program. It is the Container's responsibility to handle any XA handshakes required when a global transaction is active. The only configuration required is that deployment descriptor element `<resource-ref>`, with reference to the JMS Connection factory JNDI object, be set up to use the XA variant. If it is non-XA, the program still runs, but there are no atomicity guarantees, in other words, it is a local transaction. Also note that for AppServer to automatically handle the transaction handshakes it is necessary to have the application run in a Container, either EJB, Web or appclient. For example, a java client with no JMS XA API calls will not have its JMS activity participate in a global transaction, one has to write it as a J2EE application client instead. Also make sure that all connection factories are looked up through deployment descriptor element `<resource-ref>`. This allows the Container to trap the JMS API calls and insert appropriate hooks.

Let us examine in more detail the following sentences extracted from the EJB 2.1 specification:

> Because the container manages the transactional enlistment of JMS sessions on behalf of a bean, the parameters of the createSession(boolean transacted,int acknowledgeMode), createQueueSession(boolean transacted,int acknowledgeMode) and createTopicSession(boolean transacted, int acknowledgeMode) methods are ignored. It is recommended that the Bean Provider specify that a session is transacted, but provide 0 for the value of the acknowledgment mode.

The assumption here is that messages produced/consumed by JMS sessions should be included as part of the unit of work maintained by a global transaction, should a global transaction be active. In order for transactional enlistment to occur, the parent connection factory of connections on which `createSession()`, `createQueueSession()` or `createTopicSession()` are invoked must be defined as a `javax.jms.XAConnectionFactory`, `javax.jms.XAQueueConnectionFactory` or `javax.jms.XATopicConnectionFactory`, respectively. That is, the value for `<res-type>` of J2EE deployment descriptor element `<resource-ref>`, with definition of JMS connection factory to be used for the J2EE component, must be either `javax.jms.XAConnectionFactory`, `javax.jms.XAQueueConnectionFactory` or `javax.jms.XATopicConnectionFactory`. If the connection factory has a non XA connection factory `<res-type>`, the program still runs but work performed on JMS sessions will not be included in the global transaction; in this case the `transacted` and `acknowledgeMode` parameters will influence the behavior of message production/consumption. For instance:

```
import javax.jms.*;

QueueConnectionFactory nonXAQCF;
Queue                  myQueue;

try {
 javax.naming.Context ctx = (javax.naming.Context) new
javax.naming.InitialContext();
 nonXAQCF = (QueueConnectionFactory) ctx.lookup("java:comp/env/jms/
MyJMSQueueConnectionFactory");
 myQueue = (Queue) ctx.lookup("java:comp/env/jms/MyJMSQueue");
}
catch (javax.naming.NamingException exp) {
 exp.printStackTrace();
}

// Note: A global transaction context is currently active when the Session
         is being created

QueueSession qSession = conn.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
QueueSender = qSession.createSender(myQueue);
TextMessage msg = qSession.createTextMessage("A Message ");
sender.send(msg);
```

Here, the TextMessage `msg` is queued regardless of the outcome of the active global transaction. This is in line with test cases in the J2EE Compatibility Test Suite. It seems useful to have this capability in a global transaction, whereby a log message needs to be sent irrespective of the enclosing global transaction's completion result.

Multiple resource access within a single global transaction is supported in AppServer. This provides the capability to do a unit of work which is composed of sending/receiving JMS messages along with some other type of resource manager access. That is, it is desirable to write code (in an EJB for example) which does some work against a non JMS resource such as a database and also send a message to a queue with the Container providing transactional completion for all work performed. Upon completion of the transaction, either the work performed against the database is committed AND the message is queued, or should something fail during the transaction database work is rolled back AND the message is not delivered to the queue.

In application code, an EJB method such as `doSomeWork()` shown below is supported in AppServer:

```
// Business method in a session bean, the EJB container marks the transaction
     void doSomeWork()
     {
       // Establish a database connection
          java.sql.Connection dbConn = datasource.getConnection();

        // Execute SQL
          ⋮

       // Call a remote EJB in the same transaction

          ejbRemote.doWork();

       // Send a JMS message to a queue
          jmsSender.send(msg);

      }
```

Enabling the JMS services security

See Chapter 23, “JMS provider pluggability” for vendor-specific information on JMS services such as security.

Advanced Concepts for Configuring JMS Connection Factories and Destinations

JDBC datasource resource objects and JMS Connection Factories and Destinations for JMS providers Tibco, SonicMQ and WMQ, are defined in a DAR module using a `jndi-definitions.xml` descriptor. Module `tibco-resources.dar`, deployed to Welcome Partition in sample BAS configuration j2eeSample contains some default Tibco Connection Factories, Topics and Queues, defined for the AppServer install with Tibco JMS server option. You can edit these existing definitions to suit your environment or create new definitions using the DDEditor. JMS connection factories, similar to JDBC datasources, are classes that wrap the connection factory classes provided by JMS vendors. If you want to use a JMS vendor not bundled or certified to work with AppServer, you need to deploy that vendor's connection factory classes to your Partition.

See Chapter 23, “JMS provider pluggability” for vendor-specific information on JMS queues.

Chapter

23

JMS provider pluggability

The Borland AppServer (AppServer) is designed to support any arbitrary JMS provider as long as certain requirements are met. There are three aspects of JMS pluggability: runtime pluggability, configuration of JMS admin objects (connection factories and queues/topics), and service management. You will achieve the best results if all three are met, but just having the runtime level pluggability, as well as vendor-specific ways to achieve the other levels, may be sufficient in many situations.

Borland AppServer 6.6 bundles the Tibco EMS 4.2.0 V12 and OpenJMS 0.7.6.1 JMS providers. OpenJMS is bundled as a partition level service.

Runtime pluggability

Runtime pluggability is determined by compliance to the J2EE specification. A CTS compliant JMS product that additionally implements the JMS specification optional APIs can seamlessly plug into the AppServer runtime. All features like transactions and MDB support are retained.

JMS products must possess the capability to perform transactional messaging to support MDBs and J2EE container intercepted messaging. That is, a JMS queue or topic must be a transactional resource. AppServer requires that JMS products implement the JTA XAResource interface and support JMS XA APIs.

In addition, the JMS product should support the `javax.jms.ConnectionConsumer` interface. The latter is vital since a central idea of MDBs is the concurrent consumption of messages. The ConnectionConsumer interface achieves this. The mechanism also works in conjunction with some optimal methods of the `javax.jms.Session` objects, namely `Session.run()` and `Session.setMessageListener()`.

Configuring JMS administered objects (connection factories, queues and topics)

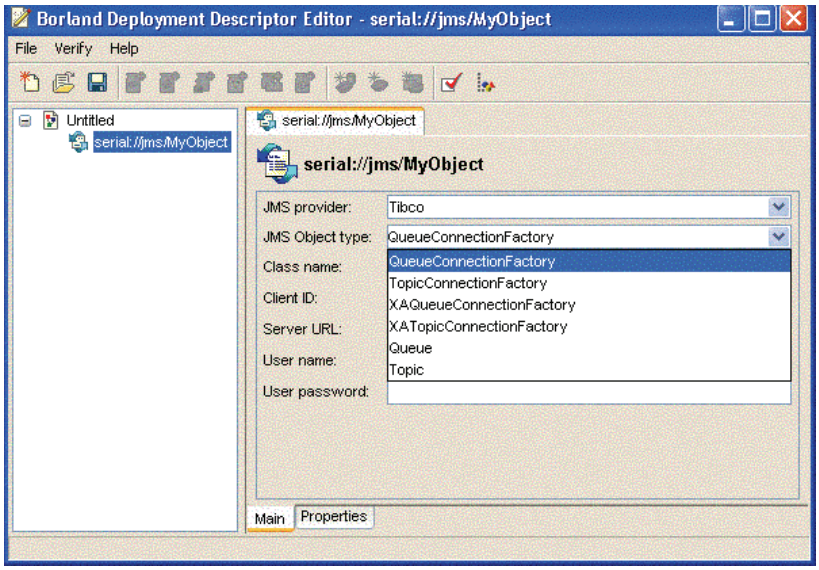
If the JMS providers' admin objects, like connection factories and destinations follow the JavaBeans specification (as encouraged in the JMS specification), the Borland Deployment Descriptor Editor tool can define, edit and deploy these objects into the AppServer JNDI tree without needing a JMS product-specific mechanism.

For specific information on using other JMS service providers with AppServer and requirements for admin objects (queues, topics, and connection factories), see “Other JMS providers” on page 229.

Setting Admin Objects Using Borland Deployment Descriptor

You can set the admin object properties for Tibco from the Borland Deployment Descriptor Editor. To do so:

1. Launch the Borland Deployment Descriptor Editor from within the Management Console or standalone from the Start menu.
2. Select File|New and click on the JNDI Definitions tab to bring it forward.
3. Select the JNDI Definitions Archive and click OK to create a new JMS object.
4. Right-click on the Untitled JMS object in the left pane and select New JMS Object.
5. Give the JMS object a name in the New JMS Object window and click OK. Your JMS object will appear under your archive.
6. Click on your JMS object, and select the Main tab.
7. Configure your object by selecting various fields from the drop-down menus and entering information in the properties' fields.
8. To add additional properties for the JMS object, select the Properties tab and click “Add” to add properties (name, type, value).



Service Management for JMS Providers

The AppServer service control infrastructure can manage the JMS service process (either a JVM or native process, whatever form it takes in the JMS provider) as a first class managed object. Operations like starting, stopping and server configuration is provided for supported (Tibco, and OpenJMS) providers out of the box.

Tibco EMS 4.2

Tibco has achieved the runtime level of pluggability that is determined by the compliance to the J2EE specification. Tibco 4.2 is JMS 1.1 compliant and supports unified JMS APIs.

Added value for Tibco

Tibco provides this added value:

- Transparent installation
- Tibco Admin Console is available from AppServer Management Console *Tools* menu.

Configuring Admin Objects for Tibco

Tibco's admin object properties are defined in AppServer and can be configured graphically using the Borland Deployment Descriptor Editor.

See "Setting Admin Objects Using Borland Deployment Descriptor" on page 218.

Auto Queue Creation Feature in Tibco

Tibco has an auto queue creation feature by which if a specified queue does not exist in the server, the Tibco server will create the queue as necessary.

Tibco Admin Console

Note You can launch the Tibco Admin Console from within AppServer only for the Windows platform. For all other platforms, run the executable from `<tibco_home>` directory to launch the console.

AppServer includes the Tibco Admin Console for additional configuration. To launch the Tibco Admin Console, select it from the Tools menu in the AppServer Management Console.

Configuring clients for fault tolerant Tibco connections

To connect to a backup server in the event of failure of a primary server, a client application must specify multiple server URLs in the `jndi-object` XML for the connection factories as below:

```
<jndi-object>
   <jndi-name>jms/XAConnectionFactory</jndi-name>
   <class-name>com.tibco.tibjms.TibjmsXAConnectionFactory</class-name>
   <property>
      <prop-name>serverUrl</prop-name>
      <prop-type>String</prop-type>
      <prop-value>localhost:7222,anotherhost:7222</prop-value>
   </property>
</jndi-object>
<jndi-object>
   <jndi-name>jms/ConnectionFactory</jndi-name>
   <class-name>com.tibco.tibjms.TibjmsConnectionFactory</class-name>
   <property>
      <prop-name>serverUrl</prop-name>
      <prop-type>String</prop-type>
      <prop-value>localhost:7222,anotherhost:7222</prop-value>
   </property>
</jndi-object>
<jndi-object>
 <jndi-name>jms/XAQueueConnectionFactory</jndi-name>
 <class-name>com.tibco.tibjms.TibjmsXAQueueConnectionFactory</class-name>
 <property>
  <prop-name>serverUrl</prop-name>
  <prop-type>String</prop-type>
  <prop-value>localhost:7222,anotherhost:7222</prop-value>
 </property>
</jndi-object>
<jndi-object>
 <jndi-name>jms/QueueConnectionFactory</jndi-name>
 <class-name>com.tibco.tibjms.TibjmsQueueConnectionFactory</class-name>
 <property>
  <prop-name>serverUrl</prop-name>
  <prop-type>String</prop-type>
  <prop-value>localhost:7222,anotherhost:7222</prop-value>
 </property>
</jndi-object>
<jndi-object>
 <jndi-name>jms/XATopicConnectionFactory</jndi-name>
 <class-name>com.tibco.tibjms.TibjmsXATopicConnectionFactory</class-name>
 <property>
  <prop-name>serverUrl</prop-name>
  <prop-type>String</prop-type>
  <prop-value>localhost:7222,anotherhost:7222</prop-value>
 </property>
</jndi-object>
 <jndi-name>jms/TopicConnectionFactory</jndi-name>
 <class-name>com.tibco.tibjms.TibjmsTopicConnectionFactory</class-name>
 <property>
  <prop-name>serverUrl</prop-name>
  <prop-type>String</prop-type>
  <prop-value>localhost:7222,anotherhost:7222</prop-value>
 </property>
</jndi-object>
```

Enabling Security for Tibco

Note For information on SSL, please refer to the Tibco documentation. Tibco documentation is located in `<install_dir>\jms\tibco\doc\html`.

To enable security for Tibco, you can either modify the `tibemsd.conf` file located in `/<install_dir>/jms/tibco/bin`, or you can set it using the Tibco Admin tool.

Note Make sure that the Tibco service is active before following the steps below.

1. From the Tools menu in Borland Management Console launch the Tibco Admin Console tool.
2. Type `connect`.
3. Enter Login name and Password.
4. Once connected, type `set server authorization=enabled`.
5. Security is now enabled. For client authentication, users should be created and added to an authorization group. For instance, create a user using the command:

   ```
   create user <name> [<description>] [password=<password>]
   ```

6. Add a member, type `add member <group-name> <user-name> [,<user-name2>,...]`.

Disabling security for Tibco

Follow the steps for Enabling Security for Tibco described above, but in step 4 rather than enabling security set the server authorization to disabled:

```
set server authorization=disabled
```

OpenJMS

OpenJMS is tied to the lifecycle of an AppServer partition. The AppServer contains a complete footprint of OpenJMS.

Note OpenJMS 7.6 is JMS 1.0 compliant and does not support the unified APIs.

OpenJMS provides the following added value:

- Transparent Installation
- Support for automatic table creation
- Out-of-box integration with AppServer's Naming Service (JNDI), Transaction Service and Datasource
- Provides partition-level service management
- Support for RMI connector using VisiBroker
- Lifecycle management using Borland Management Console

When you install AppServer with OpenJMS in Borland AppServer version 6.6, OpenJMS comes packaged as a partition level template. This means that partitions created from this template get OpenJMS as an in-process service.

The following properties in the partition template are set to `true` by default when you install AppServer with OpenJMS:

- `ejb.mdb.use_jms_threads=true`

  This property is needed to allow the transaction that was started by OpenJMS to propagate into AppServer

- `ejb.mdb.local_transaction_optimization=true`

  This property is needed to allow the use of non-XA JMS connection factories to be used in transactional scenarios. If not set, the MDBs that have transaction `onMessage` method will fail to deploy

- `jts.allow_unrecoverable_completion=true`

  By default, AppServer uses the JDataStore database for message persistence. If your application database is different from your message persistence database, you will need to set this property to true in order to achieve the two phase commit.

See Chapter 36, "EJB, JSS, and JTS Properties" for more information on each of the properties.

Even though OpenJMS can be used in a standalone mode where it is the only service in the partition, there are some advantages to using OpenJMS as an in-process service:

- It helps avoid the two-phase commit (2PC), and hence the performance cost and deployment complexity associated with it. This involves JDBC connection sharing among different components in the partition that access the database. It can be achieved by making OpenJMS persist messages in the same database as the one in which application data is stored. This configuration where 2PC can be avoided is possible only when OpenJMS is accessed through its embedded or RMI connector inside the AppServer partition. See "Configuring Datasource to Achieve 2PC Optimization" on page 225 for details.
- Since all the components are centralized in a single virtual machine, you can avoid the cost of TCP/IP. The JMS client library calls an in-process JMS service using regular Java call and vice versa.
- Since VBJ provides local call optimizations, the application won't need to have two types of connectors. It can use only the RMI connector irrespective of client location with respect to the JMS server.

You can find the OpenJMS product documentation in the `<appserver_install>/jms/openjms/docs` directory.

Configuring JNDI objects for OpenJMS

Since each partition in AppServer can host an instance of OpenJMS, there is a dedicated configuration file, `openjms.xml`, for each partition. The `openjms.xml` file contains information about various OpenJMS connectors and JNDI objects that the instance hosts.

Note AppServer does not support the administration GUI in OpenJMS. You can create and delete queues for OpenJMS by editing the `openjms.xml` file.

To add new Queues, Topics or Connection Factories for OpenJMS, you must modify its configuration file, `openjms.xml`. To access this file:

1. Select the relevant partition in the left pane of the Borland Management Console.
2. Right-click on the OpenJMS service in the left pane.
3. Select **Properties** from the drop-down menu.
4. Click on the openjms.xml tab in the properties pane to bring it forward.

5 Edit the file to add the JNDI objects.

See the OpenJMS documentation in the <appserver_install>/jms/openjms/docs directory for more details on this file.

The code sample below shows you how to add a Queue and a Topic connection factory.

- You can add any number of factories as necessary for your application.
- Make sure that each object you add has a unique name so as to avoid being overwritten in JNDI. The name has to be unique among multiple instances.
- You must have at least one TopicConnectionFactory and one QueueConnectionFactory in the embedded scheme.
- If you do not specify a port for the connector, the default port will be used. Refer to the OpenJMS documentation for details at <appserver_install>/jms/openjms/docs.

```
<Configuration>

  <ServerConfiguration host="localhost" embeddedJNDI="false" />

  <JndiConfiguration>
    <property name="java.naming.factory.initial"
              value="com.inprise.j2ee.jndi.CtxFactory" />
    <property name="java.naming.provider.url"
              value="serial://" />
  </JndiConfiguration>

  <Connectors>
    <Connector scheme="embedded">
      <ConnectionFactories>
        <QueueConnectionFactory name="jms/EmbeddedQueueConnectionFactory" />
        <TopicConnectionFactory name="jms/EmbeddedTopicConnectionFactory" />
      </ConnectionFactories>
    </Connector>

   <Connector scheme="tcp">
      <ConnectionFactories>
        <QueueConnectionFactory name="jms/TcpQueueConnectionFactory" />
        <TopicConnectionFactory name="jms/TcpTopicConnectionFactory" />
        <QueueConnectionFactory name="jms/qcf" />
        <QueueConnectionFactory name="jms/QueueConnectionFactory" />
        <QueueConnectionFactory name="jms/xaqcf" />
        <TopicConnectionFactory name="jms/tcf" />
        <TopicConnectionFactory name="jms/TopicConnectionFactory" />
        <TopicConnectionFactory name="jms/xatcf" />
      </ConnectionFactories>
    </Connector>

    <Connector scheme="rmi">
      <ConnectionFactories>
        <QueueConnectionFactory name="jms/qcf" />
        <QueueConnectionFactory name="jms/QueueConnectionFactory" />
        <QueueConnectionFactory name="jms/xaqcf" />
        <TopicConnectionFactory name="jms/tcf" />
        <TopicConnectionFactory name="jms/TopicConnectionFactory" />
        <TopicConnectionFactory name="jms/xatcf" />
      </ConnectionFactories>
    </Connector>

  </Connectors>
```

Note When configuring JMS resource objects in Borland deployment descriptors, in preparation for an application JNDI lookup, be sure you add a serial:// prefix to the value of jndi-name elements. For example, serial://jms/q. OpenJMS resource objects are deployed independent of DAR files. They are bound directly under JNDI with a serial:// prefixed name upon BAS partition start. Applications that perform a JNDI lookup of an OpenJMS resource object must use the serial:// prefix to resolve the object.

Connection Modes in OpenJMS

OpenJMS supports multiple ways for the client to access it—using Embedded, TCP, and RMI connectors.

Use the Embedded connector when OpenJMS is installed as an in-process service. Specify all the connection factories needed locally under the embedded connector section in the `openjms.xml` file. You can take advantage of the 2PC optimization only if you use OpenJMS as a partition level (in-process) service using the embedded or RMI connector. In embedded mode, the JMS clients access the JMS server using local Java calls and use embedded queue/connection factories. These factories provide optimal ways to avoid the cost of TCP/IP.

If you use OpenJMS as an out-of-process service, you must use the RMI or TCP connectors. The RMI connector in AppServer is configured to use RMI-over-IIOP and hence can carry transaction context from client to JMS server. It has the capability to optimize local calls when the clients are co-located with the OpenJMS server and so makes it more efficient. The TCP connector, which is based on custom protocol doesn't carry transaction context.

Important You can disable the TCP or RMI connectors when not using them. Do not disable the Embedded connector even if you are not using it. The Embedded connector is used internally to control service management (start, stop etc.) of OpenJMS as a part of the AppServer partition level service.

Changing the Datasource for OpenJMS

By default, when the OpenJMS service starts, it looks in the `partition.xml` file for the location of the datasource where the OpenJMS messages will be persisted. This datasource entry must exist in a DAR file. In the event that the entry is not found in the DAR file, OpenJMS service will default to the datasource mentioned in the `openjms.xml` file. If you want OpenJMS to use the datasource configured in the `partition.xml` file only, you can comment the <DatabaseConfiguration> entry out in `openjms.xml` file. This way, if the datasource is not found, you will see an error message. You can change the datasource and make it point to the same datasource which your J2EE application uses. To change the datasource:

1. Right-click on the OpenJMS service in the left pane of the Borland Management Console.
2. Select **Properties** from the drop-down menu.
3. Enter the path to the datasource in the **Name** text box in the **General** tab.
4. (Optional) Uncheck the “Clean messages on startup” checkbox if you do not want the previously stored (undelivered) messages to be deleted when you restart the partition. The messages that get delivered, are automatically deleted from the database. The messages that could not be delivered for any reason remain in the database. It is these messages that get cleaned up when you check this box. This checkbox is checked by default.

You must also specify the right database driver in the `openjms.xml` configuration file. To access this file, right-click on the OpenJMS service and select Properties from the menu. Click on the `openjms.xml` tab in the properties pane.

Creating Tables for OpenJMS

If you choose to use a database other than JDataStore, you must create appropriate tables in the database before using it. In case of JDataStore, the tables are pre-created. Use the scripts provided by OpenJMS to create tables in other databases. You can find these scripts in `<bas_install>\jms\openjms\config\db` directory. See the OpenJMS User's Guide for details. This guide is available in `<appserver_install>/jms/openjms/docs` directory.

Configuring Datasource to Achieve 2PC Optimization

By using OpenJMS as a partition-level service you can achieve the two-phase commit optimization. OpenJMS can be configured to persist data in any relational database. By default, AppServer uses the partition's JDataStore database for persisting JMS messages. You can change the default datasource and make it point to the same datasource which your J2EE application uses. (See "Changing the Datasource for OpenJMS" on page 224 for details on how to change the datasource.) This way, OpenJMS and your application will use a single transaction resource and you will be able to avoid the two-phase commit.

Important When there are multiple messaging applications in the partition, and each of them uses a distinct datasource for application data, the two-phase commit optimization is not possible for each of those applications. The two-phase commit optimization will work only with the one that has the same datasource as OpenJMS. OpenJMS can persist data in only one datasource for all applications in a partition, regardless of the number of applications in that partition. So, if a partition has multiple applications and each application stores its data in its own separate database, you can make OpenJMS datasource to point to only one of those databases. The application that stores its data in this database will achieve the 2PC optimization.

Configuring Security with OpenJMS

The authentication and security features that come with OpenJMS version 0.7.6. work with the following AppServer configurations:

1. OpenJMS authentication with TCP Connector
2. OpenJMS authentication with VBJ-based RMI Connector

   Note HTTPS and TCPS connections are not supported in AppServer 6.6 release.

   The following XML code in `openjms.xml` file shows an example of how to turn on security for configurations 1 only. It provides a list of authenticated users:

```
<SecurityConfiguration securityEnabled="true"/>
  <Users>
    <User name="admin" password="admin"/>
    <User name="j2ee" password="j2ee"/>
  </Users>
```

3. OpenJMS authentication with VBJ-based RMI connector using AppServer security

   Refer to the `<appserver_install>/examples/security/Readme.html` document on how to configure security for this configuration.

Refer to the OpenJMS documentation in `<appserver_install>/jms/openjms/docs` directory for details on how to use security in OpenJMS.

Specifying Partition Level Properties for OpenJMS

To integrate OpenJMS as a partition level service in AppServer, it is introduced as a new service in the partition's configuration. The following properties are for OpenJMS in the `partition.xml` file. This file is located in `<appserver_install>/var/domains/base/configurations/<my_config>/mos/<openjms_partition>/adm/properties` directory.

The following code in the `partition.xml` file creates OpenJMS as a partition level service:

```
- <service name="jms"
        runas.propstorage="management_runas.properties"
        version="6.5" description="JMS Service based on OpenJMS(tm)
           version 0.7.6.1"
        vendor="Borland Software Corporation"
        class="com.borland.enterprise.server.services.PartitionService"
        startup.synchronization="service_ready"
        startup.service_ready.max_wait="0"
        shutdown.synchronization=""
        shutdown.phase="1">
    <properties lifecycle.class="com.borland.jms.JmsPartitionService"
        openjms.configfile="adm/openjms/conf/openjms.xml"
        openjms.home="../../../../../../../../jms/openjms"
        openjms.clean_messages_on_startup="true"
        openjms.datasource="serial://datasources/OpenJmsDataSource"
        openjms.sql_file="adm/openjms/conf/openjms.sql"
        openjms.datasource_lookup_interval="1"
        openjms.max_datasource_lookup_retries="1" />
    </service>
```

The properties are described in the table below:

Property Name	Description	Default Value
`lifecycle.class`	Used to add an inprocess JMS service. In case of OpenJMS(tm) this property has to have a value of com.borland.jms.JmsPartitionService. The reflection based code of the partition launcher dynamically detects if the class specified in this property is in the Java CLASSPATH. If it finds it, it tries to load and start the service.	`com.borland.jms.JmsPartitionService`
`openjms.configfile`	This property specifies the location of the configuration file. The location is relative to the current working directory of your partition. This file is the central place where configuration of OpenJMS(tm) is stored. This file is needed for the embedded OpenJMS(tm) service to come up with a AppServer partition. This file also contains the list of JNDI objects (Queues, and Topics) that need to be created as part of the OpenJMS(tm) service startup.	`adm/openjms/conf/openjms.xml`
`openjms.home`	This property specifies where OpenJMS(tm) is installed. OpenJMS uses the value specified here to locate various resources.	`<AppServerInstallRoot>/jms/openjms`

Property Name	Description	Default Value
openjms.clean_messages_on_startup	This property indicates whether to clean up the database tables containing JMS messages across partition restart. Currently, this property is only available for JDataStore. For other databases, you must delete messages manually.	true
openjms.datasource	This property specifies the JNDI name of the datasource that is used to persist messages in OpenJMS(tm). If this datasource is the same as the one that your application uses, then the JDBC connection pool will be shared among them and a single JDBC connection will be used both to persist messages and provide data access to your application, thereby avoiding a need for 2PC. If the specified datasource is not available in the JNDI namespace at the startup time, the startup code will use the properties openjms.datasource_lookup_interval, and openjms.max_datasource_lookup_retries described below, to make multiple attempts to access the target datasource. Despite that, if the lookup fails, the initialization code will internally construct a datasource from the information specified in the configuration (openjms.xml) file. **Note:** The startup code will use the information from openjms.xml file only if the user specified datasource is not available. If the datasource is pre-deployed or available in JNDI , RDBMS configuration in the openjms.xml file will be ignored.	serial://datasources/JDSLocal
openjms.sql_file	This property is used to specify the file that contains the SQL statements to drop, and create database tables. These tables are used by OpenJMS(tm) for message persistence.	adm/openjms/conf/openjms.sql
openjms.datasource_lookup_interval	This property specifies the duration between successive datasource lookup attempts. See the property openjms.datasource property above.	1 second
openjms.max_datasource_lookup_retries	This property specifies the number of attempts to lookup for the datasource before attempting to use the default datasource.See the property openjms.datasource property above.	5 seconds
openjms.recreate_database_on_startup	This property when set will cause re-creation of database across each startup of the service. This is useful for cases where previous messages are not needed in the subsequent runs (for example, while testing).	false
openjms.database.softcommit	This property is only applicable when JDataStore is used to persist JMS messages. This property provides improved performance during commit process, but with lack of recoverability in some rare failure scenarios. See JDataStore documentation for more details.	true

Property Name	Description	Default Value
openjms.database	This property only applies to JDS and is used to specify the name of the JDS database.	openjms.jds
openjms.use_bes_transactions	OpenJMS starts a transaction before it dispatches messages. It uses the transaction service that is part of the partition which contains OpenJMS. If no transaction service is available in the partition, one from the Smart Agent domain is selected. Use this property when you use OpenJMS during transactions. This property does not affect the messaging applications that don't involve transactions. However, to avoid an extra cost of transaction startup and propagation turn this property off.	true

OpenJMS Topologies

Important If you have two OpenJMS services with both of them using the TCP connector, make sure you have different port numbers specified for them in the `openjms.xml` file. To open this file, right-click on the OpenJMS service in the Borland Management Console and select Properties from the drop down menu. Click on the openjms.xml tab in the properties pane.

OpenJMS can be configured to run in the following two topologies:

- **Server shared mode**—where the OpenJMS service is hosted in a dedicated partition with other services in that partition disabled. It is available as a shared service to all partitions that are in the same osagent domain as the configuration in which OpenJMS partition resides. The remote clients can access OpenJMS via an RMI or TCP connector. Since OpenJMS needs a naming service to bind the JNDI objects that are specified in its configuration file, the Transaction and Naming services must be enabled in the partition that hosts OpenJMS or they should be available in the SmartAgent domain.
- **Embedded Service mode**—where OpenJMS is run as an embedded service in each of the partitions in the configuration. Each partition uses the embedded (intra-virtual machine) connector of OpenJMS instead of the TCP or RMI connector. The JMS clients use the embedded queue/topic connection factories. These factories provide optimal ways to avoid the cost of TCP/IP. Even though JMS clients can use the RMI connector in this mode, to achieve maximum performance it must use the local (embedded) connector.

Note If multiple OpenJMS service instances are running in a single SmartAgent domain of AppServer, there will be no database sharability or automatic failover to a running instance. This is because there is no support for clustering for OpenJMS.

Using Message Driven Beans (MDB) with OpenJMS

For an AppServer partition to support MDBs, the MDB must be able to access a JMS server. To make the MDB access the OpenJMS server, make sure that:

1. OpenJMS is installed and enabled as an in-process service in your partition or available in the domain. Right-click on the OpenJMS service and select Start from the menu to enable the service.
2. The resource references are properly configured in the `ejb-jar.xml` file to point to the right type of connection factory.

Important If your MDB needs transactional access, you must use Embedded or RMI connection factories with your MDB so as to support transaction propagation.

Other JMS providers

Borland AppServer supports the SonicMQ 6.0/6.1 and WebSphereMQ 5.3/6.0 JMS providers. For information on how to integrate SonicMQ with AppServer see Chapter 24, “Integrating SonicMQ into Borland AppServer.” For information on how to integrate WebSphereMQ with AppServer see Chapter 25, “Integrating WebSphereMQ into Borland AppServer (BAS).”

Chapter

24

Integrating SonicMQ into Borland AppServer

This document provides the steps to enable Borland AppServer (AppServer) to work with an independent installation of a SonicMQ 6.0/6.1 JMS provider. Both SonicMQ versions 6.0 and 6.1 are JMS 1.1 compliant.

Note You must purchase SonicMQ separately. It is not bundled with AppServer 6.6.

Installing SonicMQ

Install SonicMQ to a location independent of the AppServer installation. Make sure that the Management features are installed so that you can manage the SonicMQ services through the Sonic Management Console.

Configuring SonicMQ Administered Objects in AppServer

You must define the JMS administered objects accessed through JNDI in the Borland proprietary DAR modules. The Borland Deployment Descriptor Editor(DDEditor) tool in AppServer allows you to create the administered objects in a DAR module. See "Setting Admin Objects Using Borland Deployment Descriptor" on page 218.

See the *SonicMQ V6.1 Configuration and Management Guide* for information on all the properties that can be configured for administered objects using the Sonic JMS Administered Objects tool.

See Chapter 22, "Using JMS" for a description of AppServer related properties that can be applied to JMS connection factory object definitions in DAR modules.

Resolving SonicMQ library modules in the AppServer environment

SonicMQ 6.0/6.1 client libraries, `sonic_Client.jar` and `sonic_XA.jar`, and their dependent libraries must be loaded by AppServer for deployment of J2EE application(s) that wish to access a SonicMQ server.

The suggested approach for enabling SonicMQ client libraries in AppServer is to apply the following updates to JMS related configuration files located under `<AppServer>/bin`:

- Edit the `sonic.config` file and set the value of `jms.home` to the root directory of the external SonicMQ installation. For example:

```
set jms.home=C:/SonicMQ/V61
```

- Edit the `jms.config` file. Uncomment the statement to include `sonic.config`. Make sure that the include statements for other JMS providers are commented out:

```
#include $var(installRoot)/bin/tibco.config
#include $var(installRoot)/bin/openjms.config
include $var(installRoot)/bin/sonic.config
```

This allows SonicMQ client libraries to be resolved by all AppServer partitions and by J2EE client applications run by AppServer appclient tool.

Configuring Automatic Queue Creation for SonicMQ Queues deployed to AppServer

When a DAR module containing definition of SonicMQ JMS Queues is deployed to a partition, AppServer can be configured to automatically create the JMS Queues in the target SonicMQ server. Certain SonicMQ management libraries need to be available to AppServer for automatic JMS Queue creation to occur. These libraries must be loaded from the partition's classpath. This can be achieved by updating AppServer configuration files `sonic.config` and `jms.config` as described above. Additionally, the following steps must be performed:

- Make sure that the naming service definition in the partition's configuration file, `partition.xml`, has `jns.auto-create-queues` property set to true as follows:

```
<service name="visinaming"
        runas.propstorage="management_runas.properties" version="6.6"
      description="Naming Service" vendor="Borland Software Corporation"
     class="com.borland.enterprise.server.services.naming.NamingService"
         startup.synchronization="service_ready"
startup.service_ready.max_wait="0"
         shutdown.synchronization="" shutdown.phase="1">
         <properties jns.name="namingservice"
         jns.auto-create-queues="true">
         </properties>
</service>
```

- Update `partition-server.config` file of the partition to ensure target SonicMQ server can be located:

  a Open the Management Console.

  b Switch to the Installations view by clicking on the Installations icon on the extreme left side of the console.

  c In the left pane, navigate to the partition for which you want to make the change. The General Properties page for the partition will open in the right pane.

  d Click on the Files tab at the bottom of the right pane.

e Select partition-server.config in the lower left pane.

f Scroll to the end of the file and enter the following for only the properties you want to change:

```
vmprop <property_name>=<value>
```

You can do this for the following 5 properties:

Property	Default Value
sonicmq.domainName	domain1
sonicmq.brokerURL	tcp://localhost:2506
sonicmq.user	Administrator
sonicmq.pwd	Administrator
sonicmq.brokerName	/Brokers/Broker1

g Save edits and restart the partition.

Note SonicMQ Server must be active prior to deployment of a DAR module with SonicMQ JMS Queues in order to successfully auto-create the queues.

Chapter

25

Integrating WebSphereMQ into Borland AppServer (BAS)

This document provides the steps to enable Borland AppServer (AppServer) to work with an independent installation of a WebSphereMQ 5.3/6.0 JMS provider.

Note You must purchase WebSphereMQ separately. It is not bundled with AppServer 6.6.

Supported Versions

Both WebSphereMQ 5.3 and 6.0 are certified to work with the product.

WebSphereMQ Configuration

To configure WebSphereMQ:

WebSphereMQ 5.3

Out of the box installation of WMQ 5.3 does not support JMS 1.1 APIs. To take advantage of the JMS1.1 features, fix pack 06 (CSD06) or above should be installed on top of WMQ 5.3 installation.

The “standard” WebSphereMQ Client supports only the local (i.e. one-phase commit) transactions, managed by the queue manager to which the client application is connected. To support distributed transactions(2PC), you must install WebSphereMQ Extended Transactional Client.

The WebSphereMQ Extended Transactional Client is a fee-based feature of WebSphereMQ version 5.3. It extends the WebSphereMQ capabilities by allowing WebSphereMQ client applications to participate in globally coordinated transactions. In other words, it offers two-phase commit (XA compliant) processing support to WebSphereMQ client applications so that they can participate in global transactions managed by some external transaction managers.

WebSphereMQ 6.0

The default installation of WebSphereMQ 6.0 has support for JMS 1.1 APIs.

WebSphereMQ 6.0 has built in support for distributed transactions(2PC) and MQ Extended Transactional Client installation is not required.

Configuring Admin Objects with WebSphereMQ

WebSphereMQ's admin object properties are defined in BES and can be configured graphically using the Borland Deployment Descriptor Editor. See "Setting Admin Objects Using Borland Deployment Descriptor" on page 218.

For a complete list of JNDI properties and other configuration options available with WebSphereMQ 5.3, refer to WebSphereMQ's *Using Java* document published at `http://publibfp.boulder.ibm.com/epubs/pdf/csqzaw12.pdf`.

For a complete list of JNDI properties and other configuration options available with WebSphereMQ 6.0, refer the WebSphereMQ *Using Java* document published at `http://publibfp.boulder.ibm.com/epubs/pdf/csqzaw13.pdf`.

Locating WebSphereMQ Library modules at runtime

WMQ 5.3 Client libraries need to be loaded in BAS partition for deployment of J2EE application(s) that wish to access a WMQ5.3 server.Following are the full set of libraries required by a BAS partition

- `com.ibm.mq.jar`
- `com.ibm.mqjms.jar`
- `com.ibm.mqbind.jar`
- `com.ibm.mqetclient.jar` (this jar is a part of WMQ Extended Transactional Client installation)

One approach in making these available to BAS would be to deploy them to the BAS partition hosting the J2EE application. However, a better approach is to update JMS related configuration files located under `<BAS_install>/bin`:

- Edit the `wmq53.config` and set the value of `jms.home` to the root directory of the external WMQ5.3 installation.
- Edit the `jms.config` file. Uncomment the include statement to include `wmq53.config`. Make sure that the include statements for other JMS providers are commented out:

```
#include $var(installRoot)/bin/tibco.config
#include $var(installRoot)/bin/openjms.config
#include $var(installRoot)/bin/sonic.config
include $var(installRoot)/bin/wmq53.config
```

This allows WMQ5.3 client libraries to be resolved by all BAS partitions and by J2EE client applications run by BAS tool appclient.

WebSphereMQ 6.0

WebSphereMQ 6.0 Client libraries need to be loaded in BAS partition for deployment of J2EE application(s) that will be accessing a WebSphereMQ 6.0 server. A full set of libraries required by a BAS partition are:

- com.ibm.mq.jar
- com.ibm.mqjms.jar
- dhbcore.jar
- com.ibm.mqetclient.jar (Extended transactional client)

One approach in making these available to BAS would be to deploy them to the BAS partition hosting the J2EE application. However, a better approach is to update JMS related configuration files located under <BAS_install>/bin:

- Edit the wmq60.config file and set the value of jms.home to the root directory of the external WebSphereMQ 6.0 installation.
- Edit the jms.config file. Uncomment the include statement to include wmq53.config. Make sure that the include statements for other JMS providers are commented out:

```
#include $var(installRoot)/bin/tibco.config
#include $var(installRoot)/bin/openjms.config
#include $var(installRoot)/bin/sonic.config
include $var(installRoot)/bin/wmq60.config
```

This allows WebSphereMQ 6.0 client libraries to be resolved by all BAS partitions and by J2EE client applications run by BAS tool appclient.

Chapter

26

Using JACC

The Java Authorization Contract for Containers (JACC) specification defines a contract between a J2EE application server and an authorization policy provider. All J2EE application containers, web containers, and enterprise bean containers are required to support this contract. The contract defined by this specification is divided into three subcontracts. Taken together, these subcontracts describe the installation and configuration of authorization providers such that they will be used by containers in performing their access decisions.

JACC Contracts

The three subcontracts are the Provider Configuration Subcontract, the Policy Configuration Subcontract, and the Policy Decision and Enforcement Subcontract.

Provider Configuration Subcontract

The Provider Configuration Subcontract defines the requirements placed on providers and containers such that Policy providers may be integrated with containers.

Policy Configuration Subcontract

The Policy Configuration Subcontract defines the interactions between container deployment tools and providers to support the translation of declarative J2EE authorization policy into policy statements within a J2SE Policy provider.

Policy Decision and Enforcement Subcontract

The Policy Decision and Enforcement Subcontract defines the interactions between container policy enforcement points and policy decisions required by J2EE containers.

How the JACC-based authorization works

JACC allows the EJB and Web containers in an application server to interact with third party authorization providers to make authorization decisions when a J2EE resource is accessed. The Web and EJB containers in a J2EE application server use JACC-compliant authorization providers to restrict client access to the resources and services. The providers do this based on the policy information propagated to them by the deploy tool during application deployment. The provider stores this information in its repository for use when making the authorization decisions. Authorization decisions are made by the provider based on whether the principal (user) belongs to a role that has the necessary privileges to access a particular resource.

When an application is being deployed, the AppServer does the following:

1. Create a unique contextID that uniquely identifies the module that is being deployed.
2. Build the PolicyConfiguration with the set of Permissions that will be required to access each resource of the module.
3. Propagate the security policy information to the provider through the JACC APIs.

When a client/user makes a request to access an EJB method or a servlet or URL:

1. The EJB container or the Web container creates an appropriate permission object and the ProtectionDomain object containing the principals of the caller.
2. The container then calls the Policy.implies method of the java.security.Policy object implemented by the provider and passes the two objects to the provider.
3. The provider makes the decision based on the policy information it has stored (by checking its principal-to-role map) and returns a boolean value to the container.
4. If the role to which the principal belongs has access permissions to the resource, the implies method returns true and the user is allowed to access the resource by the container. Otherwise, it returns a false and the user is denied access to the resource.

Configuring JACC provider in Borland AppServer

The JACC provider in AppServer implements the standard java.security.Policy object specified in the Provider Configuration Subcontract section, which it uses to make the access decisions. The JACC provider also implements the PolicyConfigurationFactory class and the PolicyConfiguration interface, which enables deployment tools to propagate all security elements to the provider during application deployment.

The following properties control the installation of the AppServer JACC provider:

Property Name	Description	Default Value
`javax.security.jacc.policy.provider`	Specifies the policy implementation class that will be used by the application server for policy replacement.	`com.borland.security.jacc.provider.BESJACCPolicy`
`javax.security.jacc.PolicyConfigurationFactory.rovider`	Specifies the providers PolicyConfigurationFactory implementation class.	`com.borland.security.jacc.provider.BESPolicyConfigurationFactory`

Configuring a JACC provider using AppServer Management Console

You can configure the JACC provider using the AppServer Management Console or you can configure the JACC provider properties in the `partition_server.config` file.

To configure the properties using the AppServer Management Console:

1. Select the Partition name in the left pane of the console.
2. Right-click on the partition name and select Properties from the resulting menu.
3. The Partition Properties page will open.
4. Click on the Security tab.
5. Configure the two properties in the JACC Properties box.

Configuring a JACC provider through the configuration file

To configure the JACC provider properties in the `partition_server.config` file:

1. Go to the following directory:

```
<install_dir>\var\domains\base\configurations\j2eeSample\mos\
<partition_name>\adm\properties
```

2. Open the `partition_server.config` file.
3. Locate the following lines:

```
#JACC provider configuration
vmprop javax.security.jacc.policy.provider=com.borland.security.jacc.
   provider.BESJACCPolicy
vmprop javax.security.jacc.PolicyConfigurationFactory.provider=com.borland.
   security.jacc.provider.BESPolicyConfigurationFactory
```

4. Configure the properties as desired.

Note If you leave these properties blank, the JACC provider will not be enabled and the system will fall back to security framework as existed in the previous AppServer releases.

Enabling/Disabling the JACC provider

You have the option of using one of the following:

- Configure AppServer security as a JACC provider (this is the default setting)
- JACC disabled in AppServer security—the underlying security mechanism is the same as it existed in the previous releases of AppServer
- Configure AppServer to use external JACC providers

By default, when you install the AppServer you will receive Borland VisiSecure as the JACC provider. The JACC provider shipped with AppServer is compliant with all JACC APIs and implements the Provider Configuration subcontracts specified in the JACC specification.

The default settings for security properties in the Management Console of the AppServer are set such that you can use AppServer security with the JACC APIs. If you choose to not use the JACC with the AppServer security provider, you must clear the security properties in the management console.

Alternatively, you can extend your security infrastructure by plugging in a third party JACC-based security provider into AppServer. If you choose to use an external provider, you must enter the appropriate values for the properties in the JACC Properties box in the Partition Properties dialog box. Also, make sure that the external JACC provider related jar files are deployed to the partition as library modules.

Configuring external JACC providers

Any JACC-compliant external provider can be plugged into the AppServer. The provider implementation and configuration should follow the guidelines as mentioned below:

- The provider should provide an implementation for `java.security.Policy` and the configuration has to happen correctly through the admin console or the configuration file as discussed in the earlier sections.
- The provider should provide an implementation for the `PolicyConfigurationFactory` and the configuration has to happen correctly through the admin console or the configuration file.
- All the provider dependent jar files should be deployed to the partition as library modules.

An example which demonstrates how a provider should be implemented and configured with BES is shipped with the product. Please refer to `<install dir>/examples/security/jacc` for details.

You can configure an external JACC provider using the Borland Management Console or you can configure the security properties in `partition_server.config` file.

Chapter

27

Using ADLoginModule in BAS

Active Directory is Microsoft's implementation of directory service for the Windows platform. It provides the means to manage the identities , resources, and the relationships between them, all of which make up the network environment. ADLoginModule is a new LoginModule bundled with BAS which inherits from the LDAPLoginModule and specifically works with Active Directory as backend user store.

How ADLoginModule works

User Principal Name

Different from the LDAPLoginModule, by default, ADLoginModule uses user principal name (UPN) to bind to Active Directory Server, thus performs the authentication. UPN is formed by combining object name with the Fully Qualified Domain Name (FQDN)—*objectname@QFDN*. For example, for `user1` in domain `abc.def.net`, the user principal name `user1@abc.def.net` will be used as the security principal (instead of the DN as in LDAPLoginModule).

Authentication

Authentication process includes two steps:

1 Validate the username/password pair against the user backend store

2 Populate user attributes, which will be used for authorization at latter stage

During the first step, ADLoginModule forms the User Principal Name with the provided username and the domain name option. With the password provided by the user, ADLoginModule binds to the Active Directory. A successful binding operation means the user is authenticated by the Active Directory server.

After the successful authentication, ADLoginModule gets the Distinguished Name (DN) for the user entry from the Active Directory, and populates the designated set of attributes (from options specified in JAAS configuration). In doing so, ADLoginModule searches from SEARCHBASE context and looks for entry satisfying the filter "userPrincipalName=*UPN*".

With the DN information in hand, ADLoginModule populates the required attributes of that entry based on options specified in JAAS configuration.

Configuring ADLoginModule

A new option DOMAINNAME is added specifically for ADLoginModule, which indicates the domain to which this entity is authenticated against. A sample configuration looks as follows:

```
adrealm {
    com.borland.security.provider.authn.ADLoginModule required
    INITIALCONTEXTFACTORY=com.sun.jndi.ldap.LdapCtxFactory
    PROVIDERURL="ldap://testing.net"
    DOMAINNAME=abc.def.net
    SEARCHBASE="cn=users,dc=abc,dc=def,dc=net"
  };
```

with this configuration, the user will be authenticated against Active Directory Server at host `testing.net`, and for the domain `adc.def.net`. The user entry will be searched from SEARCHBASE "cn=users,dc=abc,dc=def,dc=net".

Detailed Configuration Options

Similar to LDAPLoginModule, ADLoginModule can be configured with following entry inside JAAS configuration file:

```
<realm-name> {
     com.borland.security.provider.authn.ADLoginModule
        authentication-requirements-flag
     INITIALCONTEXTFACTORY=connection-factory-name
     PROVIDERURL=backend-url
     DOMAINNAME=[domain name as in DNS-mapped format, for example, abc.def.net]
     SEARCHBASE=search-start-point
     USERATTRIBUTES=attribute1, attribute2, ...
     USERNAMEATTRIBUTE=attribute
     QUERY=dynamic-query
  };
```

The detailed description for the options are summarized below:

Property Name	Description
INITIALCONTEXTFACTORY	The InitialContextFactory class that is used by JNDI to bind to LDAP.
PROVIDERURL	The URL to the directory server of the form ldap://<servername>:<port>. This attribute is mandatory.
DOMAINNAME	Anew attribute for Active Directory, indicates the domain name for the user. This is the recommend way to perform login with AD though is not mandatory. For login using DN, USERNAMEATTRIBUTE must be set to “DN”.
SEARCHBASE	Explicitly set the search base for the directory to lookup. This attribute is optional, if this is not specified, the search will be performed from the root context of domain.
USERATTRIBUTES	Comma-separated list of attributes that will be retrieved and stored for an authenticated user. This attribute is optional, if this is not specified all the attributes for the entry will be populated. Refer to LDAPLoginModule in the Security User Guide for more information.
USERNAMEATTRIBUTE	When user is being authenticated to the system—either through CallbackHandler or IdentityWallet, a name-password pair is required. This attribute defines the meaning of “name”—a username within a domain or a DN. This attribute is optional—if this is not specified (which is the default case), DOMAINNAME option must be specified, and user's input is treated as the username within the domain—a UPN is formed as <username>@<domainname>. On the other hand, if DN is used for login, this option must be set to “DN”. In this case, the input from user will be treated as the DN directly.
QUERY	Provides a mechanism to dynamically query the directory server for other information and represent the results as attributes. Refer to LDAPLoginModule in the Security User Guide for more information. This attribute is optional.

Chapter

28

Using JAXR

This document describes the Java API for XML Registries (JAXR). JAXR is part of J2EE 1.4 specification. It gives the J2EE developer a common standard API to access various XML registries particularly used in web services. The JAXR specification from Sun is available at `http://java.sun.com/xml/jaxr/index.jsp`.

The Borland AppServer (BAS) integrates Apache jUDDI and Apache scout to provide a UDDI registry and JAXR compliance. Apache jUDDI is an open source Java implementation of the Universal Description, Discovery, and Integration (UDDI) specification for Web Services.

JAXR specification defines two types of providers each with a different Capability Level. Each provider offers a different level of support for interacting with the two popular registry specifications, UDDI and ebXML. A type 0 provider offers support for accessing UDDI registries and type 1 provider supports access to both UDDI AND ebXML registries.

Apache scout, which is integrated with BAS, is a type 0 jUDDI JAXR provider. It adapts the jUDDI client to standard JAXR API.

Using JAXR in BAS

Before you use the JAXR APIs, you must set the classpath and system properties settings for the running JVM. You must deploy the `juddi.ear` to a BAS partition. The `juddi.ear` file is located at a BAS repository, `<BAS_home>/var/repository/archives/ears`.

You must include the following libraries that are required by the BAS partition to host the `juddi.ear`:

- `<BAS_home>/lib/scout.jar`
- `<BAS_home>/lib/juddi.jar`
- `<BAS_home>/lib/axis/axis.jar`
- `<BAS_home>/lib/axis/commons-discovery-0.2.jar`

You can include the jar files as library in to your J2EE application (ear, jar or war files) or you can deploy the jar files as a static library into the BAS partition.

If you are running JAXR in a Java client application, all the above mentioned libraries and the libraries below must be included in classpath:

- `<BAS_home>/lib/axis/commons-logging.jar`
- `<BAS_home>/lib/axis/asrt.jar`

System Property

To use the JAXR Provider for UDDI, the name of the ConnectionFactory implementation class must first be specified by setting the System Property `javax.xml.registry.ConnectionFactoryClass` to `org.apache.ws.scout.registry.ConnectionFactoryImpl`. By default, BAS partition has this property automatically set to its JVM. If you are an application user you do not need to set this property. If you are running JAXR in a standalone java application, this system property must be set to point to the JVM. Failure to specify this will result in the value defaulting to `com.sun.xml.registry.common.ConnectionFactoryImpl`, which will not be found. This will result in a `JAXRException` when the `ConnectionFactory.newInstance()` method is called. The BAS JAXR Provider for UDDI does not support lookup of the ConnectionFactory via JNDI.

JAXR Connection Properties

Connection specific properties must be set to ConnectionFactory before getting Connection from the factory. See the JAXR specification for a detailed list of the properties and their descriptions. The following is a subset of properties that are required to get a connection:

Property	Description
javax.xml.registry.queryManagerURL	The URL of the jUDDI registry's inquiry API for UDDI. This url will be of the form: http://<hostname>:<port>/juddi/inquiry. This property is required.
javax.xml.registry.lifeCycleManagerURL	The URL of the UDDI registry's publish API for UDDI. This url will be of the form: http://<hostname>:<port>/juddi/publish.
javax.xml.registry.authenticationMethod	The method of authentication to use when authenticating with the registry. This may take one of two values, `UDDI_GET_AUTHTOKEN` or `HTTP_BASIC`. The default value is `UDDI_GET_AUTHTOKEN` if none is specified.

BAS JAXR Example code

The following example shows you how to create a connection using JAXR API:

```
import javax.xml.registry.Connection;
import javax.xml.registry.ConnectionFactory;
import java.util.Properties;

public class TestConnection
{
   public static void main(String[] args)
   {
      Properties prop = new Properties();
      try
      {
         String queryurl = "http://localhost:8080/juddi/inquiry";
         prop.setProperty("javax.xml.registry.queryManagerURL", queryurl);
         prop.setProperty("javax.xml.registry.lifeCycleManagerURL", queryurl);
         ConnectionFactory factory = ConnectionFactory.newInstance();
         factory.setProperties(prop);
         Connection con = factory.createConnection();
         if(con == null)
            System.out.println("No Connection");
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

Chapter

29

Using the Scheduler Service

Borland AppServer 6.6 (AppServer) supports J2EE 1.4 compliant EJB timer service. In AppServer this service is known as the Scheduler Service. The AppServer Scheduler Service is based on Quartz. Refer to the EJB 2.1 specification for generic information on EJB Timer Service. To obtain Quartz-related documentation, go to `http://www.opensymphony.com/quartz/documentation.action`

The Scheduler Service is a partition-level service, which means that each time you create a partition, it will automatically be included as one of the partition services. The Scheduler Service can be used even when the EJB container goes down.

Configuring the Scheduler Service

You can configure some of the commonly used scheduler service properties in the AppServer Management Console. To do so:

1. Open the AppServer Management Console.
2. Double-click on the partition name whose scheduler service you want to configure to expand the node.
3. Right-click on Scheduler Service node under the partition.
4. Select Properties... from the resulting menu. The Properties dialog box will open.
5. Configure the following Scheduler Service Settings in the General tab:

   Transaction Timeout—specifies the time within which a transaction should be successful. If a transaction is not successfully completed within the time set in this field, the transaction will be marked for rollback.

   Max Redelivery Count—specifies the number of attempts that the Scheduler Service will make to redeliver a message to an application whose transaction (of which the scheduler event was a part) has been rolled back.

Clean events on startup—If this checkbox is checked, all the jobs and triggers will be deleted from the database when the partition is (re)started. It is applicable only if you specify JobStoreCMT to persist the scheduler events. This option is currently supported only for JDataStore.

Soft Commit—Check this box if you want to enable soft commit. With soft commit enabled, the operating system cache can buffer file writes from committed transactions. Soft commit improves performance, but cannot guarantee the durability of the most recently committed transactions.

6 Click on the Quartz tab to bring it forward.

7 Configure the following properties:

Maximum number of threads—Specifies the maximum number of threads in a thread pool

Job Store Type—The default choice in the drop-down menu is Memory. This allows you to store scheduler events in-memory. Select JDBC(CMT) from the menu if you want to persist events in a database.

If you select JDBC(CMT) as your Job Store Type, you must configure the following in the Settings for Job Store box:

Database—Select a database from the drop-down menu

Container Managed DataSource—Specifies the URL for the container managed datasource. Please see the Quartz documentation for details on Container Managed DataSource.

Non Container Managed DataSource—Specifies the URL for non-container managed datasource

8 To set more properties, click on the Advanced... button. The Scheduler (Quartz) Properties page will open. You can configure additional properties here.

Using JDataStore to persist scheduler events

The AppServer Scheduler Service can be configured to persist data in any relational database. By default, AppServer uses the JDataStore for persistence. If you do not specify a database in which to store the scheduler events, AppServer defaults to storing these events in the JDataStore database.

Configuring other databases to persist scheduler events

The partition's JDataStore database is used by default to persist scheduler data. However, you can configure a different database if you want to use the database used for the application data to persist the scheduler data too. To use a database other than JDataStore, you must do the following:

- Create appropriate tables in the database using the scripts provided by Quartz for that database. These scripts are available in the Quartz footprint.
- Choose the right database driver in Quartz's configuration file located at `<partition_working_directory>/adm/scheduler/bes.properties`

Setting up for 2PC Optimization

If the timer is tied to a transaction in an application, if for any reason the transaction is rolled back, then the creation or deletion of the timer will also be rolled back along with the transaction. Similarly, if the scheduler event is delivered to an EJB as part of a transaction which is eventually rolled back, the scheduler service will attempt to redeliver the event. As per EJB 2.1 specification, there must be at least one re-delivery attempt. You can configure the number of redelivery attempts that the scheduler service makes. The default is 1. This means that when a transaction is rolledback, the Scheduler Service in AppServer will try to redeliver the message once. See "Configuring the Scheduler Service" on page 251 for details on how to configure the maximum redelivery count.

To achieve 2PC optimization, you must use a common datasource to persist scheduler events and store application data which the J2EE application uses. If there are multiple applications in the partition and each of them uses a distinct datasource than 2PC optimization is not possible for each of those applications, but would work only with the one that has the same datasource as the Scheduler Service.

In some deployments, it will be necessary to use a 2PC-enabled (XA) datasource. This means the datasource JNDI name that you specify for transaction use in the `bes.properties` file will need to point to an XA-datasource in the DAR file.

Note The transactional behavior, for example the rollback operation, is only applicable if you set the persistent store to CMT.

Partition Service properties for Scheduler Service

Quartz is introduced as a new service in the partition's configuration file, `partition.xml`. The table below lists the partition service properties that are specific to Quartz integration.

Property Name	Description	Default Value
lifecycle.class	BES partition makes it possible to dynamically add new services which can follow the life cycle of the partition process.	com.borland.jms. SchedulerPartitionService
properties.location	Specifies the location of the configuration file.	<appserverInstallRoot>\ var\domains\base\ configurations\ <configName>\mos\ <partitionName>\adm\ scheduler\bes.properties
sql.location	Specifies the location of the sql scripts which you can use to create tables in the database	<partition_dir>\adm\ scheduler\ tables_jdatastore.sql
scheduler.clean_persistent_data_on_startup	Indicates whether to clean up the database tables containing scheduling data across partition restart.	false
scheduler.database_softcommit	This property is only relevant when JDataStore is used as the backing store for persistence. This property provides improved performance during commit process, but with lack of recoverability in some rare failure scenarios. See the JDataStore documentation for more details. This property is also used in AppServer JSS.	true
scheduler.transaction_timeout	Transaction timeout	No timeout. You can override the default by specifying the time in seconds before timeout occurs.
scheduler.auto_create_tables	Auto create Quartz tables if they do not already exist	true
scheduler.max_redelivery_count	Number of times scheduler service will try to redeliver the event in case transaction rolls back	1
scheduler.use_default_datasource	Whether or not to use our default datasource which has the JNDI URL `jdbc/ quartz` and points to JDataStore database at `adm/scheduler/database/scheduler.jds`	yes

Quartz properties used in AppServer

The table below lists the properties in Quartz that are used by AppServer Scheduler Service. These properties are listed in the `<appserver-install>\var\domains\base\configurations\<configuration_name>\mos\<partition_name>\adm\scheduler\bes.properties` file. For a detailed description of these properties, see the Quartz documentation.

Property Name	Description	Default Value
org.quartz.scheduler.instanceName	Specifies the name of the scheduler	TestScheduler
org.quartz.scheduler.instanceId	Specifies the id of the scheduler	AUTO
org.quartz.scheduler. wrapJobExecutionInUserTransaction	Set this property to true to start a UserTransaction before calling execute on the job. The transaction will commit after the job's execute method completes, and the JobDataMap is updated	True
org.quartz.scheduler.userTransactionURL	Specifies the JNDI URL of Application Server's UserTransaction manager. This is only used together with JobStoreCMT	java:comp/ UserTransaction
org.quartz.threadPool.class	Specifies the threadpool class	org.quartz.simpl. SimpleThreadPool
org.quartz.threadPool.threadCount	Specifies the number of threads that are available for concurrent execution of jobs. The practical value is from 1-100	30
org.quartz.threadPool.threadPriority	Specifies the thread priority. The value is between Thread.MIN_PRIORITY (1) and Thread.MAX_PRIORITY(10)	5
org.quartz.threadPool.makeThreadsDaemons	Set this property to true to make the threads in the pool created as daemon threads	True
org.quartz.jobStore.class	Specifies the JobStore class. Set this property to RAMJobStore for non-persistent and to JobStoreTx or JobStoreCMT for persistent jobs and triggers. JobStoreTx is for standalone Scheduler Service; JobStoreCMT is used if datasources are to managed by the appserver.	RAMJobStore (Memory). Currently, the AppServer Scheduler Service only supports RAMJobStore and JobStoreCMT
org.quartz.jobStore.driverDelegateClass	org.quartz.impl.jdbcjobstore.oracle.OracleDelegate for Oracle database and org.quartz.impl.jdbcjobstore.HSQLDB Delegate for JDataStore	org.quartz.impl.jdbcjobstore.HSQLDBDelegate. Currently AppServer Scheduler Service only supports JdataStore and Oracle database
org.quartz.jobStore.dataSource	Specifies the name of the container managed transaction(CMT) datasource. JobStoreCMT requires one CMT and one non-CMT datasource.	myDS
org.quartz.jobStore.nonManagedTXDataSource	Specifies the name of the non-container managed transaction datasource	myDSNoTx

Property Name	Description	Default Value
org.quartz.dataSource.NAME_CMT.jndiURL	Specifies the JNDI URL of the CMT data source. NAME_CMT is the name of the CMT datasource	jdbc/Quartz
org.quartz.dataSource.NAME_NOT_CMT.jndiURL	Specifies the JNDI URL of the non-CMT data source. NAME_NOT_CMT is the name of the non-CMT datasource	jdbc/Quartz

Clustering support

The Borland AppServer provides clustering support for the Scheduler Service. For example, if you have two identical partitions with Scheduler Service enabled on both of them. If you deploy the same application on them and register a timer in one of the applications, if that partition goes down, assuming that both applications are pointing to the same database, the replica could continue to get the timer events. The AppServer Scheduler Service supports failover.

Chapter 30

Implementing Partition Interceptors

Implementing Partition Interceptors requires the following steps:

1. Defining your interceptor using the `module-borland.xml` descriptor file.
2. Creating the interceptor class.
3. JARing the class and the descriptor file.
4. Deploy the JAR to the Partition of interest.

Defining the Interceptor

You define the interceptor by creating a `module-borland.xml` file. This file uses the following DTD:

```
<!ELEMENT module (Partition-interceptor?)>
<!ELEMENT Partition-interceptor (class-name, argument?, priority?)>
<!ELEMENT class-name (#PCDATA)>
<!ELEMENT argument (key, value)>
<!ELEMENT key (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT priority (#PCDATA)>
```

The `<class-name>` element must contain the full-path class name of the implementation contained within the JAR.

The `<priority>` element is an optional field that controls the order in which a set of interceptors for a particular Partition are fired. This value must be between 0 and 9. Priority 0 ranks before priority 9. Interceptors are fired in order during load time and in reverse order during shutdown. If two or more interceptors share the same priority, there is no way to determine or enforce which of that set will be fired relative to the other.

The `<argument>` is an optional element which contains a pair of elements, `<key>` and `<value>`. These are passed into your class implementation as a `java.util.HashMap`. Your code must extract the appropriate values from this type. The limit on arguments is imposed by the JVM implementation.

For example, the following XML defines an interceptor called `InterceptorImpl`:

```
<module>
<Partition-interceptor>
    <class-name>com.borland.enterprise.examples.InterceptorImpl</class-name>
    <argument>
        <key>key1</key>
        <value>value1</value>
    </argument>
    <argument>
        <key>key2</key>
        <value>value2</value>
    </argument>
    <argument>
        <key>key3</key>
        <value>value3</value>
    </argument>
    <priority>1</priority>
</Partition-interceptor>
</module>
```

Creating the Interceptor Class

Your class must implement:

```
com.borland.enterprise.server.Partition.service.PartitionInterceptor
```

The following methods are available:

- `public void initialize(java.util.HashMap args);`

  This method is called before any Partition services like the Tomcat container are created and initialized. This method is not subject to the <priority> parameter, since it is invoked as each interceptor is loaded.

- `public void startupPreLoad();`

  This method is called after Partition services are started and before the Partition services load modules.

- `public void startupPostLoad();`

  This method is invoked after all Partition services have loaded their respective modules.

- `public void shutdownPreUnload();`

  This method is called before the Partition services unload their respective modules. The <priority> parameter now reverses its meaning; priority 9 interceptors are called first, then priority 8, and so forth.

- `public void shutdownPostUnload();`

  This method is called after the services have unloaded their modules.

- `public void PartitionTerminating();`

  This method is called after the services have been shut down, just before the Partition shuts down.

The following code sample shows the class `InterceptorImpl` defined in the `module-borland.xml` descriptor above:

```
package com.borland.enterprise.examples;

// This interface is contained in xmlrt.jar
import com.borland.enterprise.server.Partition.service.PartitionInterceptor;

public class InterceptorImpl implements PartitionInterceptor {
    static final String _className = "InterceptorImpl";

    public void initialize(java.util.HashMap args) {
    // Writing to System.out and System.err will
    // cause the output to be logged.
    // There is no requirement to log.
     System.out.println(_className + ": initialize");
     System.out.println("key1 has value " + args.get("key1").toString());
     System.out.println("key2 has value " + args.get("key2").toString());
     System.out.println("key3 has value " + args.get("key2").toString());
    }
    public void startupPreLoad() {
    // Writing to System.out and System.err will
    // cause the output to be logged.
    // There is no requirement to log.
    System.out.println(_className + ": startupPreLoad");
    }
    public void startupPostLoad() {
    // Writing to System.out and System.err will
    // cause the output to be logged.
    // There is no requirement to log.
      System.out.println(_className + ": startupPostLoad");
    }
    public void shutdownPreUnload() {
    // Writing to System.out and System.err will
    // cause the output to be logged.
    // There is no requirement to log.
     System.out.println(_className + ": shutdownPreUnload");
    }
    public void shutdownPostUnload() {
    // Writing to System.out and System.err will
    // cause the output to be logged.
    // There is no requirement to log.
      System.out.println(_className + ": shutdownPostUnload");
    }
    public void PartitionTerminating() {
    // Writing to System.out and System.err will
    // cause the output to be logged.
    // There is no requirement to log.
    System.out.println(_className + ": PartitionTerminating");
    }
}
```

Creating the JAR file

Use Java's JAR utility to create a JAR file of the class and its descriptor file.

Deploying the Interceptor

Use the Deployment Wizard to deploy the interceptor to the Partition. **Do not** check either the “Verify deployment descriptors” or the “Generate stubs” checkboxes.

Important You must restart the Partition after deploying your interceptor.

You can also simply copy your JAR file into one of these two directories, making sure you restart the Partition manually afterward:

- `<install_dir>/var/servers/<server_name>/Partitions/<Partition_name>/lib`
- `<install_dir>/var/servers/<server_name>/Partitions/<Partition_name>/lib/system`

Chapter

31

VisiConnect overview

J2EE Connector Architecture

In the information technology environment, enterprise applications generally access functions and data associated with Enterprise Information Systems (EIS). This traditionally has been performed using non-standard, vendor-specific architectures. When multiple vendors are involved, the number of architectures involved exponentiate the complexity of the enterprise application environment. With the introduction of the Java 2 Enterprise Edition (J2EE) 1.4 Platform and the J2EE Connector Architecture (Connectors) 1.5 standards, this task has been greatly simplified.

VisiConnect, the Borland implementation of the Connectors 1.5 standard, provides a simplified environment for integrating various EISs with the Borland AppServer (AppServer). The Connectors provides a solution for integrating J2EE-platform application servers and EISs, leveraging the strengths of the J2EE platform—connection, transaction and security infrastructure—to address the challenges of EIS integration. With the Connectors, EIS vendors need not customize integration to their platforms for each application server. Through VisiConnect's strict conformance to the Connectors, the AppServer itself requires no customization in order to support integration with a new EIS.

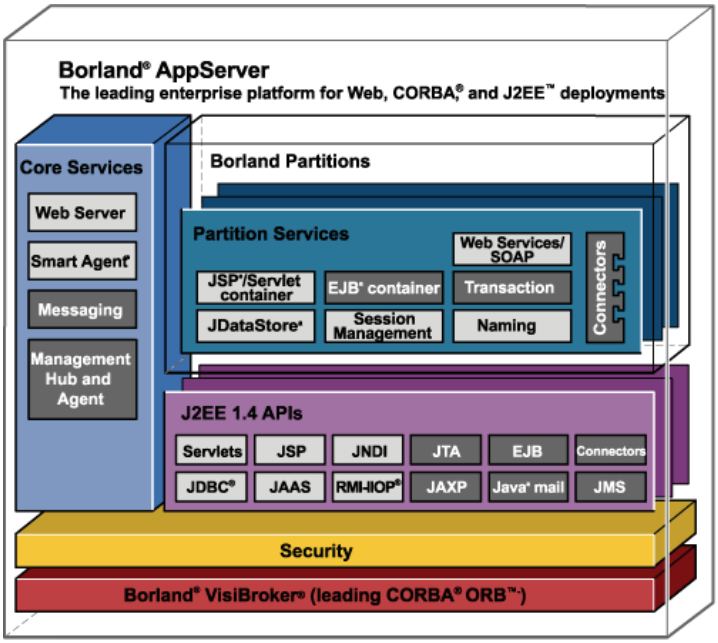
Connectors enables EIS vendors to provide standard Resource Adapters for their EISs. These Resource Adapters are deployed to the AppServer, each providing the integration implementation between the EIS and the AppServer. With VisiConnect, the AppServer ensures access to heterogeneous EISs. In turn, the EIS vendors need provide only one standard Connectors-compliant resource adapter. By default, this resource adapter has the capability to deploy to the AppServer.

Components

The Connectors environment consists of two major components—the implementation of the Connectors in the application server, and the EIS-specific Resource Adapter.

In the J2EE 1.4 Architecture, the Connectors is an extension of the J2EE Container, otherwise known as the application server. In compliance with the J2EE 1.4 Platform and Connectors 1.5 specifications, VisiConnect is an extension of the AppServer, and not a service in and of itself. The following diagram illustrates VisiConnect within the AppServer Architecture:

Figure 31.1 VisiConnect within the AppServer



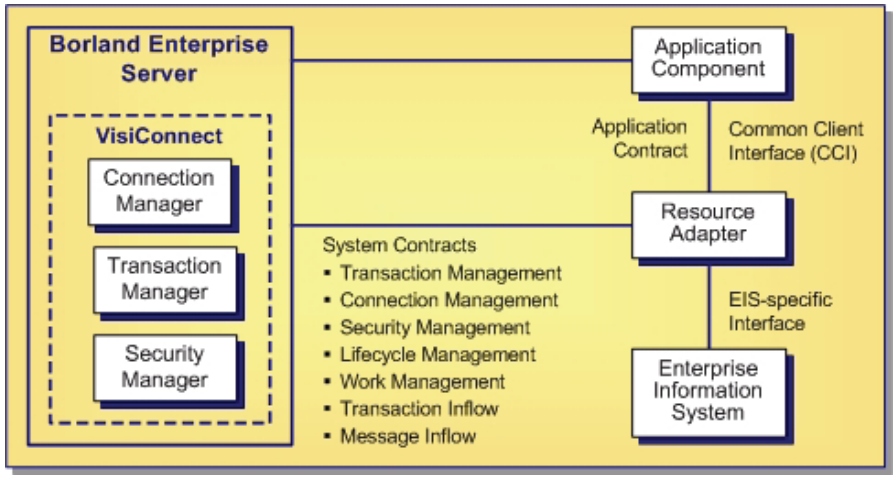
(VisiConnect is represented above by the module titled "**Connectors**.")

A Resource Adapter is a system-level driver specific to an EIS, which provides access to that EIS. To put it simply, a Resource Adapter is analogous to a JDBC driver. The interface between a Resource Adapter and the EIS is specific to the EIS. It can be either a Java interface or a native interface.

The Connectors consists of three main components:

- **System Contracts** that provide the integration between the Resource Adapter and the application server (AppServer).
- **Common Client Interface** that provides a standard client API for Java applications, frameworks, and development tools to interact with the Resource Adapter.
- **Packaging and Deployment** that provides the capacity for various Resource Adapters to plug into J2EE applications in a modular manner.

The following diagram illustrates the Connectors architecture:



A Resource Adapter and its collateral serve as the Connector. VisiConnect supports Resource Adapters developed by EIS vendors and third-party application developers written to the Connectors 1.5 standard. Resource Adapters contain the components—Java, and if necessary, native code—required to interact with the specific EIS.

System Contracts

The Connectors specification defines a set of system level contracts between the application server and an EIS-specific Resource Adapter. This collaboration keeps all system-level mechanism transparent from the application components. Thus, the application component provider focuses on the development of business and presentation logic, and need not delve into the system-level issues related to EIS integration. This promotes the development of application components with greater ease and maintainability.

VisiConnect, in compliance with the Connectors specification, has implemented the standard set of defined contracts for:

- **Connection Management**, that allows an application server to pool connections to underlying EISs, providing application components with connection services to EISs. This leads to a highly scalable application environment that supports a large number of clients requiring access to heterogeneous EISs.
- **Transaction Management**, the contract between the application server transaction manager and an EIS supporting transactional access to EIS resource managers, that enables the application server to manage transactions across multiple resource managers.
- **Security Management**, that enables secure access to underlying EISs. This provides support for a secure application environment, which reduces security threats to the EIS and protects valuable information resources managed by the EIS.
- **Lifecycle Management** allows an application server to manage the lifecycle of a resource adapter. This contract provides a mechanism for the application server to bootstrap a resource adapter instance during its deployment or application server startup, and to notify the resource adapter instance during its undeployment or during an orderly shutdown of the application server.

- **Work Management** allows a resource adapter to do work (monitor network endpoints, call application components, etc.) by submitting `Work` instances to an application server for execution. The application server dispatches threads to execute submitted `Work` instances. This allows a resource adapter to avoid creating or managing threads directly, and allows an application server to efficiently pool threads and have more control over its runtime environment. The resource adapter can control the security context and transaction context with which `Work` instances are executed.
- **Transaction Inflow** allows a resource adapter to propagate an imported transaction to an application server. This contract also allows a resource adapter to transmit transaction completion and crash recovery calls initiated by an EIS, and ensures that the ACID properties of the imported transaction are preserved.
- **Message Inflow** allows a resource adapter to asynchronously deliver messages to message endpoints residing in the application server independent of the specific messaging style, messaging semantics, and messaging infrastructure used to deliver messages. This contract also serves as the standard message provider pluggability contract that allows a wide range of message providers (Java Message Service (JMS), Java API for XML Messaging (JAXM), etc.) to be plugged into any J2EE compatible application server via a resource adapter.

Connection Management

Connections to an EIS are expensive resources to create and destroy. To support scalable applications, the application server needs to be able to pool connections to the underlying EISs. To simplify application component development, this connection pooling mechanism needs to be transparent to the components accessing the underlying EISs.

The Connectors specification supports connection pooling and management, optimizing application component performance and scalability. The connection management contract, defined between the application server and the Resource Adapter, provides:

- A consistent application development model for connection acquisition for both managed (n-tier) and non-managed (two-tier) applications.
- A framework for the Resource Adapter to provide a standard connection factory and connection interface based on the Common Client Interface (CCI), opaque to the implementation for the underlying EIS.
- A generic mechanism for providing different quality of services (QoS) advanced connection pooling, transaction management, security management, error tracing and logging—for a configured set of Resource Adapters.
- Support for the application server to implement its connection pooling facility.

VisiConnect uses connection management to:

- Create new connections to an EIS
- Configure connection factories in the Java Naming and Directory Interface (JNDI) namespace.
- Find the right connection to an EIS from an existing set of pooled connections, and reuse that connection.
- Hook in AppServer's transaction and security services.

The AppServer establishes, configures, caches and reuses connections to the EIS automatically through VisiConnect.

The application component performs a lookup of a Resource Adapter connection factory in the JNDI namespace, using the connection factory to get a connection to the underlying EIS. The connection factory delegates the connection creation request to the VisiConnect connection manager instance. On receiving this request, the connection manager performs a lookup in the connection pool. If there is no connection in the pool that can satisfy the connection request, VisiConnect uses the `ManagedConnectionFactory` implemented by the Resource Adapter to create a new physical connection to the underlying EIS. If VisiConnect finds a matching connection in the pool, it then uses the matching `ManagedConnection` instance to satisfy the connection request. If a new `ManagedConnection` instance is created, the server adds the new `ManagedConnection` instance to the connection pool.

VisiConnect registers a `ConnectionEventListener` with the `ManagedConnection` instance. This listener enables VisiConnect to receive event notifications related to the state of the ManagedConnection instance. VisiConnect uses these notifications to manage connection pooling, transactions, connection cleanup and handle error conditions.

VisiConnect uses the `ManagedConnection` instance to provide a Connection instance that acts as an application-level handle to the underlying physical connection, to the application component. The component in turn uses this handle—and not the underlying physical connection directly—to access EIS resources.

Transaction Management

Transactional access to multiple EISs is an important and often critical requirement for enterprise applications. The Connectors supports transaction access to multiple, heterogeneous EISs—where a number of interactions must be committed together, or not at all, in order to maintain data consistency and integrity.

VisiConnect utilizes the AppServer's transaction manager and supports Resource Adapters conforming to the following transaction support levels.

- **No Transaction support**: if a Resource Adapter supports neither Local Transactions nor XA Transactions, it is non-transactional. If an application component uses a non-transactional Resource Adapter, the application component must not involve any connections to the respective EIS in a transaction. If the application component is required to involve EIS connections in a transaction, the application component must use a Resource Adapter which support Local or XA Transactions.
- **Local Transaction support**: the application server manages resources directly, which are local to the Resource Adapter. Unlike XA Transactions, local transactions can neither participate in the two-phase commit (2PC) protocol, nor participate as a distributed transaction (whereas the transaction context is simply propagated); instead, local transactions solely target one-phase commit (1PC) optimization. A Resource Adapter defines the type of transaction support in its Sun standard deployment descriptor. When an application component requests an EIS connection as part of a transaction, AppServer starts a local transaction based on the current transaction context. When the application closes the connection, AppServer commits the local transaction, and cleans up the EIS connection once the transaction is completed.
- **XA Transaction support**: a transaction is managed by a transaction manager external to the Resource Adapter and the EIS. A Resource Adapter defines the type of transaction support in its Sun-standard deployment descriptor. When an application component demarcates an EIS connection request as part of a transaction, the AppServer is responsible for enlisting the XA resource with the

transaction manager. When the application component closes that connection, the application server unlists the XA resource from the transaction manager, and cleans up the EIS connection once the transaction is completed.

In compliance with the Connectors 1.5 specification, VisiConnect provides full support for all three specified transaction levels.

One-Phase Commit Optimization

In many cases, a transaction is limited in scope to a single EIS, and the EIS resource manager performs its own transaction management—this is the Local Transaction. An XA Transaction can span multiple resource managers, thus requiring transaction coordination to be performed by an external transaction manager, typically one packaged with an application server. This external transaction manager can either use the 2PC protocol, or propagate the transaction context as a distributed transaction, to manage a transaction that spans multiple EISs. If only one resource manager is participating in an XA Transaction, it uses the 1PC protocol. In an environment where a singleton resource manager is handling its own transaction management, 1PC optimization can be performed, as this involves a less expensive resource than a 1PC XA Transaction.

Security Management

In compliance with the Connectors 1.5 specification, VisiConnect supports both container-managed and component-managed sign-on. At runtime, VisiConnect determines the selected sign-on mechanism based on information specified in deployment descriptor of the invoking component. If VisiConnect is unable to determine the sign-on mechanism requested by the component (most often due to an improper JNDI lookup of the Resource Adapter connection factory), VisiConnect will attempt container-managed sign-on. If the component has specified explicit security information, this will be presented in the call to obtain the connection, even in the case of container-managed sign-on.

Component-Managed Sign-on

When employing component-managed sign-on, the component provides all the required security information—most commonly a username and a password—when requesting to obtain a connection to an EIS. The application server provides no additional security processing other than to pass the security information along on the request for the connection. The Resource Adapter uses the component-provided security information to perform EIS sign-on in an implementation-specific manner.

Container-Managed Sign-on

When employing container-managed sign-on, the component does not present any security information, and the container must determine the necessary sign-on information, providing this information to the Resource Adapter in the request to obtain a connection. The container must determine an appropriate resource principal and provide this resource principal information to the Resource Adapter in the form of a Java Authentication and Authorization Service (JAAS) Subject object.

EIS-Managed Sign-on

When employing EIS-managed sign-on, the Resource Adapter internally obtains all of its EIS connections with a pre-configured, hard-coded set of security information. In this scenario the Resource Adapter does not depend upon the security information passed to it in the invoking component's requests for new connections.

Authentication Mechanisms

The AppServer user must be authenticated whenever they request access to a protected AppServer resource. For this reason, each user is required to provide a credential (a username/password pair or a digital certificate) to AppServer. The following types of authentication mechanisms are supported by AppServer:

- Password authentication a user ID and password are requested from the user and sent to AppServer in clear text. Borland Enterprise Server checks the information and if it is trustworthy, grants access to the protected resource.
- The SSL (or HTTPS) protocol can be used to provide an additional level of security to password authentication. Because the SSL protocol encrypts the data transferred between the client and AppServer, the user ID and password of the user do not flow in the clear. Therefore, AppServer can authenticate the user without compromising the confidentiality of the user's ID and password.
- Certificate authentication: when an SSL or HTTPS client request is initiated, AppServer responds by presenting its digital certificate to the client. The client then verifies the digital certificate and an SSL connection is established. The CertAuthenticator class then extracts data from the client's digital certificate to determine which AppServer User owns the certificate and then retrieves the authenticated User from the AppServer security realm.
- You can also use mutual authentication. In this case, Borland Enterprise Server not only authenticates itself, it also requires authentication from the requesting client. Clients are required to submit digital certificates issued by a trusted certificate authority. Mutual authentication is useful when you must restrict access to trusted clients only. For example, you might restrict access by accepting only clients with digital certificates provided by you.

For more information, see “Getting Started with Security” in the Developer's Guide.

Security Map

In Section 8.5 of the Connectors 1.5 specification, a number of possible options are identified for defining a Resource Principal on the behalf of whom sign-on is being performed. VisiConnect implements the Principal Mapping option identified in the specification.

Under this option, a resource principal is determined by mapping from the identity of the initiating caller principal for the invoking component. The resulting resource principal does not inherit the identity of security attributes of the principal that is it mapped from. Instead, the resource principal derives its identity and security attributes based on the defined mapping. Thus, to enable and use container-managed sign-on, VisiConnect provides the Security Map to specify the initiating principal association with a resourceprincipal. Expanding upon this model, VisiConnect provides a mechanism to map initiating caller roles to resource roles.

If container-managed sign-on is requested by the component and no Security Map is configured for the deployed Resource Adapter, an attempt is made to obtain the connection using a null JAAS Subject object. This is supported based upon the Resource Adapter implementation.

While the defined connection management system contracts define how security information is exchanged between the AppServer and the Resource Adapter, the determination to use container-managed sign-on or component-managed sign-on is based on deployment information defined for the component requesting a connection.

The Security Map is specified with the security-map element in the ra-borland.xml deployment descriptor. This element defines the initiating role association with a resource role. Each security-map element provides a mechanism to define appropriate resource role values for the Resource Adapter and EIS sign-on processing. The security-map elements provide the means to specify a defined set of initiating roles and the corresponding resource role to be used when allocating managed connections and connection handles.

A default resource role can be defined for the connection factory in the security-map element. To do this, specify a user-role value of "*" and a corresponding resource-role value. The defined resource-role is then utilized whenever the current identity if not matched elsewhere in the Security Map.

This is an optional element. However, it must be specified in some form when container-managed sign-on is supported by the Resource Adapter and any component uses it. Additionally, the deployment-time population of the connection pool is attempted using the defined default resource role, given that one is specified.

Security Policy Processing

The Connectors 1.5 specification defines default security policies for any Resource Adapters running in an application server. It also defines a way for a Resource Adapter to provide its own specific security policies overriding the default.

In compliance with this specification, AppServer dynamically modifies the runtime environment for Resource Adapters. If the Resource Adapter has not defined specific security policies, AppServer overrides the runtime environment for the Resource Adapter with the default security policies specified in the Connectors 1.5 specification. If the Resource Adapter has defined specific security policies, Borland Enterprise Server first overrides the runtime environment for the Resource Adapter first with a combination of the default security policies for Resource Adapters and the specific policies defined for the Resource Adapter. Resource Adapters define specific security policies using the security-permission-spec element in the ra.xml deployment descriptor file.

For more information on security policy processing requirements, see Section 18.2, "Security Permissions", in the Connectors 1.5 specification (`http://java.sun.com/j2ee/download.html#connectorspec`).

Common Client Interface (CCI)

The Common Client Interface (CCI) defines a standard client API for application components. The CCI enables application components, Enterprise Application Integration (EAI) frameworks, and development tools to drive interactions across heterogeneous EISs using a common client API.

The CCI is targeted for use by EAI and enterprise tool vendors. The Connectors 1.5 specification recommends that the CCI be the basis for richer functionality provided by the tool vendors, rather than being an application-level programming interface used by most application developers. Application components themselves may also write to the API. As the CCI is a low-level interface, this use is generally reserved for the migration of legacy modules to the J2EE 1.4 Platform. Through the CCI, legacy EIS clients can integrate directly with the AppServer; this provides for a smoother, less costly migration path to J2EE 1.4.

The CCI defines a remote function call interface that focuses on executing functions on an EIS and retrieving the results. The CCI is independent of a specific EIS; in other words, it is not bound to the data types, invocation hooks, and signatures of a particular EIS. The CCI is capable of being driven by EIS-specific metadata from a repository.

The CCI enables the AppServer to create and manage connections to an EIS, execute an interaction, and manage data records as input, output, or return values. The CCI is designed to leverage the Java Beans architecture and Java Collection framework.

The Connectors 1.5 specification recommends that a Resource Adapter support CCI as its client API, while it requires the Resource Adapter to implement the system contracts. A developer may choose to write the Resource Adapter to provide a client API different from the CCI, such as:

- the Java Database Connectivity (JDBC) API (an example of a general EIS-type interface), or
- for example, the client API based on the IBM CICS Java Gateway (an example of a EIS-specific interface)

The CCI (which form the application contract) consists of the following:

- **ConnectionFactory** A ConnectionFactory implementation creates a connection and interaction object as a means of interacting with an EIS. Its getConnection method gets a connection to an EIS instance.
- **Connection** A Connection implementation represents an application level handle to an EIS instance. The actual connection is represented by a ManagedConnection. An application gets a Connection object by using the getConnection method of a ConnectionFactory object.
- **Interaction** An Interaction implementation is what drives a particular interaction. It is created using the ConnectionFactory. The following three arguments are needed to carry out an interaction via the Interaction implementation: InteractionSpec, which identifies the characteristics of the concrete interaction, and Input and Output, which both carry the exchanged data.
- **InteractionSpec** An InteractionSpec implementation defines all interaction-relevant properties of a connector (for example, the name of the program to call, the interaction mode, and so forth). The InteractionSpec is passed as an argument to an Interaction implementation when a particular interaction has to be carried out.
- **Input and output** The input and output are records.

A **record** is a logical collection of application data elements that combines the actual record bytes together with its type. Examples are COBOL and C data structures. Record implementation in CCI uses streams. In the javax.resource.cci.Streamable interface, reading and writing from streams is handled by read and write methods. In the javax.resource.cci.Record interface, getRecordName() and getRecordShortDescription(), and setRecordName() and setRecordShortDescription() get and set the record data.

You must create records for all of the data structures that are externalized by the EIS functions you want to reuse. You then use the records as input and output objects that pass data via a Resource Adapter to and from an EIS. You will want to consider the following options when creating a record:

- **Having direct access to nested, or hierarchical, records** A direct, or 'flattened', set of accessor methods may be more convenient, or seem more natural, to some users. For example, programmers accustomed to COBOL may expect to be able to refer directly to the field of a sub-record if the field name is unique within the record. This is similar to the way COBOL field names are scoped. There is no need to qualify field names if the field name is unique.
- **Custom and Dynamic Records** You can generally create two types of records: custom and dynamic. The main difference between these is the way fields are accessed. For dynamic records, the fields are found by taking the field name, looking up the offset and the marshalling of the information, and then accessing it.

For custom records, the offset and the marshalling of the information is in the code, resulting in faster access. Generating custom records results in more efficient code, but there are restrictions on their use.

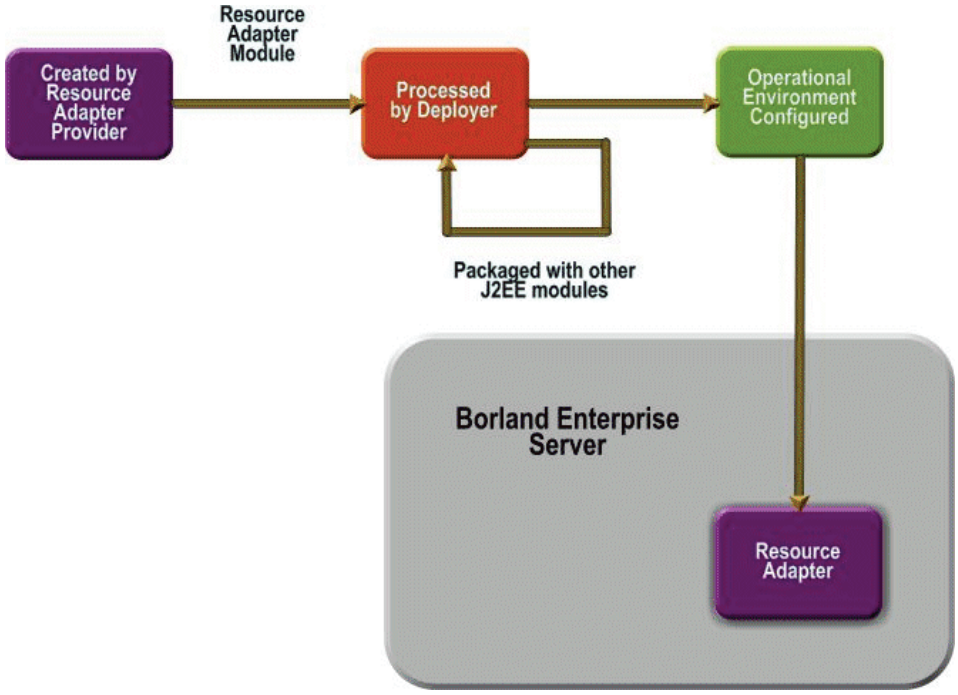
- **Records with or without notification** If a record is created with notification, then the properties of the record are bound.

Note If bound properties are not required, then it is more efficient to create a record without notification.

Packaging and Deployment

The Connectors provides packaging and deployment interfaces so that various Resource Adapters can be deployed to J2EE 1.4 Platform compliant application servers, such as the AppServer.

Figure 31.2 Packaging and Deployment in the AppServer and VisiConnect



A Resource Adapter packages a set of Java interfaces and classes, which implement the Connectors-specified system contracts and EIS-specific functionality to be provided by the Resource Adapter. The Resource Adapter can also require the use of native libraries specific to the underlying EIS, and other collateral, for example:

- Documentation
- Help files
- A code generator for EJBs
- A tool that directly provides configuration utilities so you can configure the EIS directly
- A tool that provides additional deployment facilities for remote Resource Adapter components
- For example, with IBM CICS, a set of JCL scripts that you may need to run on the mainframe

The Java interfaces and classes are packaged together, with required collateral and deployment descriptors, to create a Resource Adapter module. The deployment descriptors define the deployment contract between a Resource Adapter and the application server

A Resource Adapter can be deployed as a shared, standalone module, or packaged as part of a J2EE application. During deployment, the Resource Adapter module is installed on the AppServer and configured for the target operational environment. The configuration of a Resource Adapter is based on the properties defined in the deployment descriptors.

VisiConnect Features

Among the value-added features provided by VisiConnect as enhancements to the Connectors standard are the following:

- VisiConnect Partition Service
- Additional Classloading Support
- Secure Password Credential Storage
- Connection Leak Detection
- Security Policy Processing of ra.xml Specifications

VisiConnect Partition Service

The Borland Partition with the VisiConnect service enabled is designed to support development and deployment of J2EE applications which bundle Resource Adapters, or standalone Resource Adapter components. The AppServer Partition provides integrated VisiConnect services. Tools include a Deployment Descriptor Editor (DDE) and a set of task wizards for packaging and deploying Resource Adapters and their related descriptor files.

This provides a highly modular environment for running VisiConnect. The AppServer provides a default VisiConnect Service in Partitions for deployment.

Additional Classloading Support

VisiConnect supports the loading of properties or classes that are specified in ClassPath entry of the Resource Adapter's Manifest.mf file. The following is a description of how you configure properties and classes that are in and used by a Resource Adapter.

The Resource Adapter (RAR) archive file and the application component using it (for example, an EJB jar) are contained in an Enterprise Application (EAR) archive. The RAR requires resources such as Java properties that are stored in a JAR file, and that JAR file is contained within the EAR file (not in the RAR itself).

You specify a reference to the RAR Java classes by adding a ClassPath= entry in the RAR Manifest.mf file. You can also store the EJB Java classes in the same JAR file that is contained within the EAR. This scenario provides a “support” JAR file that contains Java classes for the components in the EAR that require them.

Secure Password Credential Storage

VisiConnect provides a standard method for Resource Adapter deployers to plug in their specified authorization/authentication mechanism through secure password credential storage.

This storage mechanism is used to map user roles (AppServer roles, which may be associated with AppServer username and password combinations or credentials) to resource roles (EIS roles, which may be associated with EIS user name and password combinations or credentials).

Connection Leak Detection

VisiConnect provides two mechanisms for preventing connection leaks:

- Leveraging a garbage collector
- Providing an idle timer for tracking the usage of connection objects

Security Policy Processing of ra.xml Specifications

VisiConnect provides a set of security permissions for execution of a Resource Adapter in a managed runtime environment. The AppServer also grants a Resource Adapter explicit permissions to access system resources.

Resource Adapters

Source code for several Resource Adapters are provided with VisiConnect as examples. Some of these Resource Adapters are wrappers for JDBC 2.0 calls, some using the CCI and some not. Deployment descriptors supporting the three transaction levels are provided for each Resource Adapter.

Simplified application examples for these JDBC Resource Adapters are provided with VisiConnect. An EJB is used to model the data in the EIS, and a J2EE client and a Servlet are used to query the Resource Adapter and display the output. The example uses any RDBMS which is supported by a JDBC 2.0 compliant driver. By default, the examples are configured to use JDataStore as the EIS, but it is a straightforward task to configure them to use any JDBC 2.0 RDBMS. The components are packaged as a J2EE Application. For more information, refer to the VisiConnect example `README` provided with the AppServer.

Other sample resource adapters provided with the product include an open-source generic JMS resource adapter with instructions for integrating with JMS providers such as Tibco and OpenJMS, and a mail resource adapter which allows you to use an email server as an EIS. These samples demonstrate the use of message inflow to allow for inbound communication from the EIS to the application server, as well as outbound connection capability.

Chapter

32

Using VisiConnect

The Java 2 Enterprise Edition (J2EE) Connector Architecture enables EIS vendors and third-party application developers to develop Resource Adapters that can be deployed to any application server supporting the J2EE 1.4 Platform Specification. The Resource Adapter provides platform-specific integration between the J2EE component and the EIS. When a Resource Adapter is deployed to the Borland AppServer (AppServer), it enables the development of robust J2EE applications which can access a wide variety of heterogeneous EISs. Resource Adapters encapsulate the Java components, and if necessary, the native components required to interact with the EIS.

Before using VisiConnect, Borland recommends that you read the Connectors 1.5 specification.

VisiConnect service

Resource adapters are hosted by Partitions with the VisiConnect Partition Service enabled. Multiple Resource Adapters can be deployed in the same Partition. VisiConnect is responsible for making the connection factories of its deployed Resource Adapters available to the client through JNDI. Thus, the client can look up the connection factory for a specific Resource Adapter using JNDI.

Service overview

The VisiConnect Service is a complete implementation of the Connectors 1.5 specification, including all optional functionality.

Every Resource Adapter object in the deployed Connector is simultaneously both a Resource Adapter object and a CORBA object.

Unlike other Connectors implementations, VisiConnect has no restrictions on partitioning. Any number of Resource Adapters can go into any number of Partitions running on any number of machines. Plus, support for distributed transactions protocol allows Resource Adapters to be partitioned arbitrarily. Partitioning enables you to configure the application during deployment to optimize its overall performance.

Connection management

The ra.xml deployment descriptor file contains a config-property element to declare a single configuration setting for a ManagedConnectionFactory instance. The resource adapter provider typically sets these configuration properties. However, if a configuration property is not set, the resource adapter deployer is responsible for providing a value for the property.

Borland provides its own deployment descriptor for defining connectors and their connection factory properties: ra-borland.xml. See Borland DTDs for more information on using the ra-borland.xml descriptor.

Configuring connection properties

The following connection pool properties can be set:

Property	Value type	Description	Default
wait-timeout	Integer	The number of seconds to wait for a free connection when maximum-capacity connections are already opened. When using the maximum-capacity property and the pool is at its max and can't serve any more connections, the threads looking for connections end up waiting for the connection(s) to become available for a long time if the wait time is unbounded (set to 0 seconds). You can set the wait-timeout period to suit your needs.	30
busy-timeout	Integer	The number of seconds to wait before a busy connection is released. If a connection is busy for a long time, the application using it may have hung and be unable to release the connection. This timeout will ensure that connections will be timed out when they have been busy for much longer than necessary.	600 (ten minutes)
idle-timeout	Integer	A pooled connection remaining in an idle state for a period of time longer than this timeout value should be closed to conserve resources. All idle connections are checked for idle-timeout expiration every 60 seconds. The value of the idle-timeout is given in seconds. A value of 0 (zero) indicates that connection cleanup is disabled.	600 (ten minutes)
maximum-capacity	Integer	Identifies the maximum number of managed connections which VisiConnect will allow. Throws ResourceAllocationException when requests for newly allocated managed connections go beyond this limit.	10

The following properties have been deprecated and are now ignored by VisiConnect. They have been replaced by the pool properties busy-timeout, idle-timeout, and wait-timeout, listed in the table above. You do not have to delete the old-style properties from ra-borland.xml.

Unused Pool properties

Property	Default	Description
initial-capacity	1	Identifies the initial number of managed connections which VisiConnect will attempt to obtain during deployment.
capacity-delta	1	Identifies the number of additional managed connections which the VisiConnect will attempt to obtain during resizing of the maintained connection pool.
cleanup-enabled	true	Indicates whether or not the Connection Pool should have unused Managed Connections reclaimed as a means to control system resources.
cleanup-delta	1	Identifies the amount of time the Connection Pool Management will wait between attempts to reclaim unused Managed Connections.

Security management with the Security Map

The Security Map enables the definition of user roles that can be

1 Used directly with the EIS for container-managed sign-on (use-caller-identity).

2 Mapped to an appropriate resource role for container-managed sign-on (run-as).

In the first case, when the user role identified at run time is found in the mapping, the user role itself is used to provide security information for interacting with an EIS. In the second case, when the user role identified at run time is found in the mapping, the associated resource role is used to provide security information for interacting with an EIS.

The use-caller-identity option is used when user identities in the user role identified at run time are available to the EIS as well. For example, a user identity, “borland”/”borland”, belonging to role “Borland”, is available to the AppServer, and the available EIS, a JDataStore database, has an identity of “borland”/”borland” available to it. When a Resource Adapter serving JDataStore is deployed with a Security Map specifying:

```
<security-map>
 <user-role>Borland</user-role>
   <use-caller-identity></use-caller-identity>
</security-map>
```

Applications on this server instance which use this JDataStore database can use use-caller-identity to access it.

Note Due to a limitation currently in VisiSecure, you must define the caller identity in the resource vault as well as the user vault.

The run-as option is used when it makes sense to map user identities in the user role identified at run time to identities in the EIS. For example, a user identity, “demo”/”demo”, belonging to role “Demo”, is available to the AppServer, and the available EIS, an Oracle database, has an identity of “scott”/”tiger”, which is ideal for a demo user. When a Resource Adapter serving Oracle is deployed with a Security Map specifying:

```
<security-map>
 <user-role>Demo</user-role>
 <run-as>
 <role-name>oracle_demo</role-name>
  <role-description>Oracle demo role</role-description>
 </run-as>
</security-map>
```

The role `oracle_demo` is defined in the resource vault (see below), applications on this server instance which use this Oracle database can use run-as to access it.

When run-as is used, the vault must be provided for VisiConnect to use to extract the security information for the resource role. A resource role name and a set of credentials are written to this vault. When VisiConnect loads a Resource Adapter with a defined Security Map using run-as, it will read in the credentials for the defined role name(s) from the vault.

Authorization domain

The `<authorization-domain>` element in the `ra-borland.xml` descriptor file specifies the authorization domain associated with a specified user role. If `<security-map>` is set, you should set `<authorization-domain>` with its associated domain. If `<authorization-domain>` is not set, VisiConnect assumes the use of the **default** authorization domain. See “Getting started with security” in the *Security Guide* for more information on using authorization domains.

Default roles

In addition, the `<security-map>` element enables the definition of a default user role that can be associated with the appropriate resource role. This default role would be preferred to if the user role identified at run-time is not found in the mapping. The default user role is defined in the `<security-map>` element with a `<user-role>` element given a value of “`*`”. For example:

```
<user-role>*</user-role>
```

A corresponding `<role-name>` entry must be included in the <security-map> element. The following example illustrates the association between an AppServer user role and a resource role.

```
<security-map>
    <user-role>*</user-role>
    <run-as>
        <role-name>SHME_OPR</role-name>
    </run-as>
</security-map>
```

The default user role is also used at deployment time if the connection pool parameters indicate that the AppServer should initialize connections. The absence of a default user role entry or the absence of a `<security-map>` element may prevent the server from creating connections using container-managed security.

Generating a resource vault

To use run-as security mapping as described above, a resource role(s) must be defined in a vault which is provided to the AppServer. This is known as the resource vault.

VisiConnect provides a tool, `ResourceVaultGen`, to create a resource vault and to instantiate role objects in this vault. A role name and its associated security credentials are written to the resource vault by `ResourceVaultGen`. At this time only credentials of type Password Credential can be written to the resource vault. The usage of `ResourceVaultGen` is as follows:

```
java -Dborland.enterprise.licenseDir=<install_dir/var/domains/base/
configurations/<configuration_name>/mos/<partition_name>/adm> -
Dserver.instance.root=<install_dir/var/domains/base/configurations/
<configuration_name>/mos/<partition_name>/adm/properties/
management_vbroker.properties>
com.borland.enterprise.visiconnect.tools.ResourceVaultGen -rolename <role_name>
-username <user_name> -password <password> -vaultfile <full path to vault file>
-vpwd <vault_password>
```

where:

`-rolename`	Resource role name to store in the resource vault.
`-username`	Resource username to associate with the resource role.
`-password`	Resource password to associate with the resource role.
`-vaultfile` (optional)	Path to the vault file you write the resource role(s)to. If not specified, ResourceVaultGen will attempt to write to the default resource vault file <*install_dir*/var/domains/base/configurations/<configuration_name>/mos/<partition_name>/adm/properties/management_vbroker.properties>. If the vault file is does not already exist, a new vault file will be written to the specified location.
`-vpwd` (optional)	Password to assign to the vault for access authorization. If not specified, the vault will be created without a password.

When using ResourceVaultGen, ensure that the following jars are in your CLASSPATH:

- lm.jar
- visiconnect.jar
- vbsec.jar
- jsse.jar
- jnet.jar
- jcert.jar
- jaas.jar
- jce1_2_1.jar
- sunjce_provider.jar
- local_policy.jar
- US_export_policy.jar

Note If you fail to include these jars in your CLASSPATH when you attempt to generate a vault, you may end up with a vault file which is invalid. If you attempt to reuse the invalid vault file, you will encounter an EOFException. To resolve, delete the invalid vault file and regenerate with ResourceVaultGen, ensuring that you have the proper jars in your CLASSPATH.

VisiConnect will use the vault if Security Map information is specified in at deployment time for a Resource Adapter. If the resource vault is password protected, VisiConnect will need to have the following property passed to it:

```
-Dvisiconnect.resource.security.vaultpwd=<vault_password>
```

If the resource vault is in a user specified location (-vaultfile ...), VisiConnect will need to have the following property passed to it:

```
-Dvisiconnect.resource.security.login=<path of specified vault file>
```

The following examples illustrate the use of ResourceVaultGen:

Example 1:

```
java -Dborland.enterprise.licenseDir=/opt/BES/var<install_dir/var/domains/base/
configurations/<configuration_name>/mos/<partition_name>/adm/properties/
management_vbroker.properties>
 -Dserver.instance.root=/opt/BES/var/servers/servername -
Dpartition.name=standard
 com.borland.enterprise.visiconnect.tools.ResourceVaultGen -rolename
administrator
 -username red -password balloon -vaultfile
 /opt/BES/var/servers/servername/adm/properties/partitions/standard/
resourcevault -vpwd
 lock
```

This usage generates a resource vault named `resourcevault` to /opt/BES/var/servers/servername/adm/properties/partitions/standard, with a role `administrator` associated with a Password Credential with username `red` and password `balloon`. The vault file itself is password protected, using the password `lock`. For VisiConnect to use this vault, the following properties must be set for it:

```
-Dvisiconnect.resource.security.vaultpwd=lock
-Dvisiconnect.resource.security.login=resourcevault
```

Example 2:

```
java -Dborland.enterprise.licenseDir=/opt/BES/var/domains/base/configurations/
<configuration_name>/mos/<partition_name>/adm/properties/
management_vbroker.properties>
-Dserver.instance.root=/opt/BES/var/domains/base/configurations/
<configuration_name>/mos/<partition_name>/adm/properties/
management_vbroker.properties>
-Dpartition.name=petstore
com.borland.enterprise.visiconnect.tools.ResourceVaultGen
-rolename manager accounts -username mickey daffy
-password mouse duck -vpwd goofy
```

This usage generates a default resource vault (named `resource_vault`) to /opt/BES/var/servers/servername/adm/properties/partitions/petstore, with a role `manager` associated with a Password Credential with username `mickey` and password `mouse`, and another role `accounts` associated with a Password Credential with username `daffy` and password `duck`. The vault file itself is password protected, using the password `goofy`. For VisiConnect to use this vault, the following properties must be set for it:

```
-Dvisiconnect.resource.security.vaultpwd=goofy
```

Example 3:

```
java -Dborland.enterprise.licenseDir=/opt/BES/var/servers/servername/adm -
Dserver.instance.root=/opt/BES/var/servers/servername
-Dpartition.name=standard
com.borland.enterprise.visiconnect.tools.ResourceVaultGen
-rolename OClone ENolco -username darkstar geraldo -password meteor rivera
```

This usage generates a default resource vault (named `resource_vault`) to /opt/BES/var/domains/base/configurations/<configuration_name>/mos/<partition_name>/adm/properties/management_vbroker.properties>, with a role `developer` associated with a Password Credential with username `darkstar` and password `meteor`, and a role `host` associated with a Password Credential with username `geraldo` and password `rivera`. The vault file itself is not password protected. VisiConnect requires no additional parameters to use this vault.

Note `ResourceVaultGen` cannot be used to write vault information to an existing file containing invalid characters. For example, a file generated by 'touch', or a StarOffice or Word document. `ResourceVaultGen` can only write vault information to a new file that it itself generates, or a valid existing vault file.

Resource Adapter overview

According to the Connectors 1.5 specification, you must be able to deploy a Resource Archive (RAR) as part of an Enterprise Archive (EAR). With AppServer and VisiConnect you can also deploy a standalone RAR. Once the RAR is deployed, you must do the following:

- Write code to obtain a connection.
- Create an Interaction object.
- Create an Interaction Spec.
- Create record and/or result set instances.
- Run the execute command so the record objects become populated.

In addition to some introductory conceptual information, this chapter provides steps to help you understand the code you must write.

The J2EE Connector Architecture enables Enterprise Information System (EIS) vendors and third-party application developers to develop Resource Adapters that can be deployed to any J2EE 1.4 compliant application server. The Resource Adapter is the main component of the J2EE Connector Architecture (Connectors), providing platform-specific integration between J2EE application components and the EIS. When a Resource Adapter is deployed to the AppServer, it enables the development of robust J2EE applications which can access a wide variety of heterogeneous EISs. Resource Adapters encapsulate the Java components and, if necessary, the native components required to interact with the EIS.

Development overview

See "Developing the Resource Adapter" on page 285 for more information.

Developing a Resource Adapter from scratch requires implementing the necessary interfaces and deployment descriptors, packaging these into a Resource Adapter Archive (RAR), and finally deploying the RAR to the AppServer. The following summarizes the main steps for developing a Resource Adapter:

1. Write Java code for the various interfaces and classes required by the Resource Adapter within the scope of the Connectors 1.5 specification.
2. Specify these classes in the ra.xml standard deployment descriptor file.
3. Compile the Java code for the interfaces and implementation into class files.
4. Package the Java classes into a Java Archive (JAR) file.
5. Create the Resource Adapter-specific deployment descriptors:
   - `ra.xml`: describes the Resource Adapter-related attributes and deployment properties using the Sun standard DTD.
   - `ra-borland.xml`: add additional AppServer-specific deployment information. This file contains the parameters for connection factories, connection pools, and security mappings.
6. Create the Resource Adapter Archive (RAR) file (that is, package the Resource Adapter)
7. Deploy the Resource Adapter Archive to the AppServer, or include it in an Enterprise Application Archive (EAR) file to be deployed as part of a J2EE application.

Editing existing Resource Adapters

If you have existing Resource Adapters you would like to deploy to the AppServer, it may only be necessary to edit the Borland-specific deployment descriptor described above and repackage the adapter. Doing so involves the following steps, with illustrative example:

1. Create an empty staging directory for the RAR:

   ```
   mkdir c:/temp/staging
   ```

2. Copy the Resource Adapter to be deployed into the staging directory:

   ```
   cp shmeAdapter.rar c:/temp/staging
   ```

3. Extract the contents of the Resource Adapter Archive:

   ```
   jar xvf shmeAdapter.rar
   ```

The staging directory should now contain the following:

- a JAR containing Java classes that implement the Resource Adapter
- a `META-INF` directory containing the files `Manifest.mf` and `ra.xml`

1. Create the `ra-borland.xml` file using the Borland Deployment Descriptor Editor (DDEditor) and save it into the staging area's `META-INF` directory. See “Using the Deployment Descriptor Editor” in the *Management Console User’s Guide* for information on using the DDEditor.
2. Create the new Resource Adapter Archive

   ```
   jar cvf shmeAdapter.rar -C c:/temp/staging
   ```

3. You may now deploy the Resource Adapter to the AppServer.

Resource Adapter Packaging

The Resource Adapter is a J2EE component contained in a RAR. Resource Adapters use a common directory format. The following is an example of a Resource Adapter's directory structure:

Resource Adapter Directory Structure:

```
.META-INF/ra.xml
.META-INF/ra-borland.xml
./images/shmeAdapter.jpg
./readme.html
./shmeAdapter.jar
./shmeUtilities.jar
./shmeEisSdkWin32.dll
./shmeEisSdkUnix.so
```

As shown in the structure above, the Resource Adapter can include documentation and related files not directly used by the Resource Adapter—for example, the image and readme files. Packaging the Resource Adapter means packaging these files as well.

Packaging a Resource Adapter includes the following steps:

1. Create a temporary staging directory.
2. Compile the Resource Adapter Java classes into the staging directory. (Or, as above, simply copy pre-compiled classes into the staging directory.)
3. Create a JAR file to store the Resource Adapter Java classes. Add this JAR to the top level of the staging directory.
4. Create a `META-INF` subdirectory in the staging area.
5. Create a ra.xml deployment descriptor in this subdirectory and add entries for the Resource Adapter. Refer to Sun Microsystems' documentation for information on the ra.xml document type definition, at `http://java.sun.com/dtd/connector_1_0.dtd`.
6. Create a ra-borland.xml deployment descriptor in this same META-INF subdirectory and add entries for the Resource Adapter. Refer to the DTD at the end of this document for details on the necessary entries.
7. Create the Resource Adapter Archive:

   ```
   jar cvf resource-adapter-archive.rar -C staging-directory
   ```

   This command creates a RAR file that you can deploy to the server. The `-C staging-directory` option instructs the JAR command to change to the `staging-directory` so that the directory paths recorded in the RAR file are relative to the directory where the Resource Adapters were staged.
   One or more Resource Adapters can be staged in a directory and packaged in a JAR file.

Deployment Descriptors for the Resource Adapter

The AppServer uses two XML files to specify deployment information. The first of these is `ra.xml`, based on Sun Microsystems' DTD for resource adapters. The second is Borland's proprietary `ra-borland.xml`, which includes additional deployment information necessary for AppServer.

Configuring ra.xml

If you do not already have an `ra.xml` file associated with your Resource Adapter, it is necessary to manually create a new one or edit an existing one. You can use a text editor or the Borland DDEditor to edit these properties. For the most up-to-date information on creating an `ra.xml` file, refer to the Connectors specification at `http://java.sun.com/j2ee/connector`.

Configuring the transaction level type

It is of critical importance that you specify the transaction level type supported by your Resource Adapter in the `ra.xml` deployment descriptor. The following table shows the transaction levels supported and how they are rendered in XML.

Transaction Support Type	XML representation
None	<transaction-support>**NoTransaction**</transaction-support>
Local	<transaction-support>**LocalTransaction**</transaction-support>
XA	<transaction-support>**XA**</transaction-support>

Configuring ra-borland.xml

The `ra-borland.xml` file contains information required for deploying a Resource Adapter to the AppServer. Certain attributes need to be specified in this file in order to deploy the RAR file. This functionality is consistent with the equivalent `.xml` extensions for EJBs, EARs, WARs, and client components for the AppServer.

Until Borland-specific deployment properties are provided in the `ra-borland.xml` file, the RAR cannot be deployed to the server. The following attributes are required in ra-borland.xml for a deployable RAR:

- Resource Adapter instance name. This name must be unique among RARs deployed to the partition. It is used by the VisiConnect service to uniquely identify the deployed Resource Adapter. When a Resource Adapter supports inbound communication, this is the name used in the `ejb-borland.xml` descriptor for the endpoint MDB to identify the Resource Adapter from which the MDB expects to receive incoming messages.
- For each connection-definition in the `ra.xml` file:
  - Connection factory interface class name. This must be unique among all connection-definitions in the Resource Adapter. This class name is used to associate the specifications for a particular connection factory in the `ra-borland.xml` with the specifications for the corresponding factory in the `ra.xml` file.
  - Connection factory name. This must be unique among all Resource Adapters deployed to the partition.
  - Connection factory JNDI name. This must be unique among all Resource Adapters deployed to the partition.

The following optional attributes may also be specified in the `ra-borland.xml` file:

- Reference to a separately deployed connection factory that contains Resource Adapter components that should be shared with the current Resource Adapter.
- Directory where all shared libraries should be copied.
- Mapping of security principals for Resource Adapter/EIS sign-on processing. This mapping identifies resource principals to be used when requesting EIS connections for applications that use container-managed security and for EIS connections requested during initial deployment.
- For each connection-definition in the `ra.xml` file:
  - Connection factory description
  - Logging-required flag. This indicates whether logging should be done for the `ManagedConnectionFactory` and `ManagedConnection` classes.
  - Log file location
  - Connection pool properties:
    - `busy-timeout`: the number of seconds to wait before a busy connection is released. The default is 600 seconds
    - `idle-timeout`: a pooled connection remaining in an idle state for a period of time longer than this timeout value should be closed. All idle connections are checked for expiration every 60 seconds. The value of the `idle-timeout` is given in seconds. The default is 600 seconds.
    - `wait-timeout`: the number of seconds to wait for a free connection. The default is 30 seconds.

Changes to the Deployment Descriptors for Connectors 1.5

BAS supports both Connectors 1.0 and Connectors 1.5. The following lists the significant changes in the deployment descriptor for Connectors 1.5:

- A new resource adapter implementation class needs to be specified in the ra.xml.

```
<resourceadapter>
   <resourceadapter-class>
      .ResourceAdapter.Implementation.Class
   </resourceadapter-class>
⋮
```

- Multiple connection definitions can be specified within the same RAR within the <outbound-resourceadapter>.
- Multiple message adapters can be specified within the <inbound-resourceadapter>
- Configuration properties can be specified at the Resource Adapter level using the `config-property` element. Note that the config-property type can only be objects and not primitives, for example, you need to use `java.lang.Integer` rather than `int`. Also, make sure that the set methods in the Resource Adapter implementation use the same types. For example, in this case: `setCount(java.lang.Integer value)`:

```
<config-property>
   <description>Open User Name</description>
   <config-property-name>Count</config-property-name>
   <config-property-type>java.lang.Integer</config-property-type>
   <config-property-value>100</config-property-value>
</config-property>
```

  Config properties can also be set specific to each connection definition and also for each of the connection definitions.

Note The Message Driven Bean that is configured as an Endpoint to receive messages through an inbound resource adapter has to implement the `messagelistener-type` interface specified in the `ra.xml`.

```
<inbound-resourceadapter>
   <messageadapter>
      <messagelistener>
         <messagelistener-type> </messagelistener-type>
```

The EJB must include the activation configuration required to activate the endpoint.

Resource Adapter Classloader Considerations

Borland AppServer provides a per-module classloader policy: each deployed module, whether .ear, .war, .rar, or ejb.jar, has its own classloader whose classpath includes all the classes packaged in that module. This allows each module to completely control the classes it will have access to; several modules can have their own copies of a library in different versions, or can use the same package names without clashing with packages used by other modules. This is a powerful feature, but can lead to complications when multiple modules need to interoperate with each other and share some classes.

While each module has its own classloader, the partition in which VisiConnect runs has several additional classloaders of its own. Classes in these per-partition classloaders are available to all modules running in the partition. Generally, the per-module classloaders take precedence over partition-level classloaders, so if a class is found in both, the per-module copy is the one that will be used.

There are two types of situations where VisiConnect users may need to exercise some care in packaging their applications to account for classloader issues:

- Connection factories and connections for outbound communication
- Message listeners for inbound communication

Connection Factories and Connections

The connection factory and connection interface and implementation classes for outbound communication may be provided along with the resource adapter. In some cases, the interface classes may in fact be standards-based and supplied as part of the JDK or an extension, while the implementation classes are proprietary and specific to the resource adapter. For example, a JMS Resource Adapter may provide its own proprietary implementation of the standard `javax.jms.QueueConnectionFactory` and `javax.jms.QueueConnection` classes.

Classes in the `javax.jms` package are provided by Borland AppServer and will be present in the partition's classloader, and all modules will share the same class definitions for those classes. But the proprietary classes that implement these `javax.jms` interfaces will be supplied by the Resource Adapter, and each module may have its own copy of these proprietary class definitions.

In order to get a connection to a Resource Adapter, the client program does a JNDI lookup for one of the Resource Adapter's connection factories. The object returned by this JNDI lookup must be created using the class definitions available to the `.ear`, `.war`, or `.jar` module where the client resides; otherwise, when the client code attempts to make use of the connection factory object, it will receive a `ClassCastException`. The client next uses the connection factory to create Connection instances. These objects, too, must be created using the client module's classloader, so that the client code can manipulate the objects.

At the same time, the VisiConnect service, running inside the AppServer, needs to make use of some of the classes provided by the Resource Adapter. For example, 1.5 level Resource Adapters contain an implementation of `javax.resource.spi.ResourceAdapter`, which will be used by VisiConnect to start and stop of the Resource Adapter. Any use by VisiConnect of the Resource Adapter classes will be done using the classloader of the Resource Adapter. If the Resource Adapter was deployed as a standalone `.rar` (rather than being embedded in an `.ear` with its clients), it will have its own classloader, and thus its own copies of the class definitions for the Resource Adapter classes. In this situation there may be a potential for `ClassCastExceptions`.

One such problem area is the method `ManagedConnectionFactory.setResourceAdapter(javax.resource.spi.ResourceAdapter)`. The `ManagedConnectionFactory` instance is created using the client classloader, while the ResourceAdapter instance is created using the `.rar` classloader. If the implementation of this method casts the ResourceAdapter instance to a proprietary implementation class, a `ClassCastException` will be thrown.

Message Listeners

An inbound resource adapter must specify a class to be its message listener. This class will be implemented by any MDB which is to serve as an endpoint for inbound communications from this Resource Adapter. When the Resource Adapter has a message which is to be passed on to the MDB, it will invoke a method in the message listener class. For example, many JMS Resource Adapters will use `javax.jms.MessageListener` as their message listener class, and the `onMessage(javax.jms.Message)` method in this class to actually receive the incoming messages. Since all these classes are provided by the `javax.jms` package, which is in the partition's classloader and therefore shared by both the Resource Adapter and the MDB client, there is no possibility here of `ClassCastExceptions`.

However, a Resource Adapter is free to provide its own proprietary message listener class, and this class may have any number of methods to actually deliver messages, all of which may use proprietary objects as arguments. This may be a source of `ClassCastExceptions`.

VisiConnect will ensure that even if the message listener class is proprietary, calls to the MDB will be properly handled such that the message delivery method in the MDB is invoked using the definition of the message listener from the MDB's own classloader. However, if that method takes an argument which is a proprietary object, VisiConnect cannot map from the Resource Adapter's class definition of that object to the MDB's class definition. This will lead to `ClassCastExceptions`.

For example, the Mail Resource Adapter provided as a sample with the product provides a message listener class called `com.borland.enterprise.ra.mail.api.MailListener`. This class contains a message delivery method called `onMessage(javax.mail.Message)`. Notice that the message listener class is proprietary, but its `onMessage()` method takes a non-proprietary object as an argument. This situation will NOT cause `ClassCastExceptions`. The message listener class itself may be proprietary. Only if one or more arguments on a message delivery method are proprietary objects will there be a classloader problem.

Correcting ClassCastExceptions

In any of the problem situations described above, you have two basic possibilities for a resolution:

- Create an `.ear` that contains the Resource Adapter `.rar` and its clients. Since the entire `.ear` will share a classloader, there can be no `ClassCastExceptions` moving objects between the Resource Adapter and the client.
- Remove the Resource Adapter classes from both the `.rar` and the client modules, and deploy these classes to the partition as a library `.jar`. A library `.jar` will be placed in the partition's classloader, shared by all deployed modules. Therefore, all modules will use the same class definitions for the Resource Adapter classes, and no `ClassCastExceptions` will be thrown.

Developing the Resource Adapter

This section describes how to develop a Connectors 1.5-compliant Resource Adapter. Resource Adapters must implement the following system contract requirements, discussed in detail below:

- Connection management
- Security management
- Transaction management
- Packaging and deployment

Connection management

The connection management contract for the resource adapter specifies a number of classes and interfaces necessary for providing the system contract. The resource adapter must implement the following interfaces:

- `javax.resource.spi.ManagedConnection`
- `javax.resource.spi.ManagedConnectionFactory`
- `javax.resource.spi.ManagedConnectionMetaData`

  The ManagedConnection implementation provided by the Resource Adapter must, in turn, supply implementations of the following interfaces and classes to provide support to the application server. It is the application server which will ultimately be managing the connection and associated transactions.

Note If your environment is non-managed (that is, not managed by the application server), you are not required to use these interfaces or classes.

- `javax.resource.spi.ConnectionEvent`
- `javax.resource.spi.ConnectionEventListener`

In addition, support for error logging and tracing must be provided by implementing the following methods in the Resource Adapter:

- `ManagedConnectionFactory.setLogWriter()`
- `ManagedConnectionFactory.getLogWriter()`
- `ManagedConnection.setLogWriter()`
- `ManagedConnection.getLogWriter()`

The resource adapter must also provide a *default implementation* of the `javax.resource.spi.ConnectionManager` interface for cases in which the Resource Adapter is used in a non-managed two-tier application scenario. A default implementation of `ConnectionManager` enables the Resource Adapter to provide services specific to itself. These services can include connection pooling, error logging and tracing, and security management. The default `ConnectionManager` delegates to the `ManagedConnectionFactory` the creation of physical connections to the underlying EIS.

In an application-server-managed environment, the Resource Adapter should not use the default `ConnectionManager` implementation class. Managed environments do not allow resource adapters to support their own connection pooling. In this case, the application server is responsible for connection pooling. A Resource Adapter can, however, have multiple `ConnectionManager` instances per physical connection transparent to the application server and its components.

Transaction management

Resource Adapters are easily classified based on the level of transaction support they provide. These levels are:

- **NoTransaction**: the Resource Adapter supports neither local not JTA transactions, and implements no transaction interfaces.
- **LocalTransaction**: the Resource Adapter supports resource manager local transactions by implementing the `LocalTransaction` interface. The local transaction management contract is specified in Section 6.7 of the Connectors 1.5 specification from Sun Microsystems.
- **XATransaction**: the Resource Adapter supports both resource manager local and JTA/XA transactions by implementing the LocalTransaction and XAResource interfaces, respectively. The XA Resource-based contract is specified in Section 6.6 of the Connectors 1.5 specification from Sun Microsystems.

The transaction support levels above reflect the major steps of transaction support that a Resource Adapter must implement to allow server-managed transaction coordination. Depending on its transaction capabilities and the requirements of its underlying EIS, a Resource Adapter can choose to support any one of the above levels.

Security management

The security management contract requirements for a Resource Adapter are as follows:

- The Resource Adapter is required to support the security contract by implementing the `ManagedConnectionFactory.createManagedConnection()` method.
- The Resource Adapter is not required to support re-authentication as part of its `ManagedConnection.getConnection()` method implementation.
- The Resource Adapter is required to specify its support for the security contract as part of its deployment descriptor. The relevant deployment descriptor elements are:
  - `<authentication-mechanism></authentication-mechanism>`
  - `<authentication-mechanism-type></authentication-mechanism-type>`
  - `<reauthentication-support></reauthentication-support>`
  - `<credential-interface></credential-interface>`

Refer to section 10.3.1 of the Connectors 1.5 specification for more details on these descriptor elements.

Packaging and deployment

The file format for a packaged Resource Adapter module defines the contract between a Resource Adapter provider and a Resource Adapter deployer. A packaged Resource Adapter includes the following elements:

- Java classes and interfaces that are required for the implementation of both the Connectors system-level contracts and the functionality of the Resource Adapter
- Utility Java classes for the Resource Adapter

- Platform-dependent native libraries required by the Resource Adapter
- Help files and documentation
- Descriptive meta information that ties the above elements together

For more information on packaging requirements, refer to Section 10.3 and 10.5 of the Connectors 1.5 specification, which discuss deployment requirements and supporting JNDI configuration and lookup, respectively.

Deploying the Resource Adapter

Deployment of Resource Adapters is similar to deployment of EJBs, Enterprise Applications and Web Applications. As with these modules, a Resource Adapter can be deployed as an archive file or as an expanded directory. A Resource Adapter can be deployed either dynamically using the AppServer Console or the `iastool` utilities, or as a part of an EAR. See the Borland AppServer *User's Guide* for deployment details.

When a Resource Adapter is deployed, a name must be specified for the module. This name provides a logical reference to the Resource Adapter deployment that, among other things, can be used to update or remove the Resource Adapter. The AppServer implicitly assigns a deployment name that matches the filename of the RAR file or deployment directory containing the Resource Adapter. This logical name can be used to manage the Resource Adapter after the server has started. The Resource Adapter deployment name remains active in the AppServer until the module is undeployed.

Application development overview

Developing application components

Common Client Interface (CCI)

The client APIs used by application components for EIS access can be categorized as follows:

- The standard common client interface (CCI) defined in Section 9 of the Connectors 1.5 specification.
- A general client interface specific to the type of Resource Adapter and its underlying EIS. For example, JDBC is one such interface for RDBMSs.
- A proprietary client interface specific to the particular Resource Adapter and its underlying EIS. For example, the CICS Java Gateway is one such interface for the IBM CICS transaction processor, and the JFC for the SAP R/3 enterprise resource planner is another.

The Connectors 1.5 specification defines the CCI for EIS access. The CCI is a standard client API for application components that enables these and EAI frameworks to drive interactions across heterogeneous EISs. The CCI is primarily targeted for Enterprise Application Integration (EAI), third-party enterprise tool vendors, and migration of legacy modules to the J2EE Platform.

In the CCI, a connection factory is a public interface that enables connection to an EIS instance. The ConnectionFactory interface is implemented by the Resource Adapter to provide this service. An application looks up a ConnectionFactory instance in the JNDI namespace, and uses it to request to obtain EIS connections.

The application then uses the returned Connection interface to access the EIS. To provide a consistent application programming model across both CCI and EIS-specific APIs, the ConnectionFactory and Connection interfaces comply to the Interface Template design pattern. This defines the skeleton of the connection creation and connection closing, deferring the appropriate steps to subclasses. This allows for these interfaces to be easily extended and adapted to redefine certain steps of connection creation and closing without changing these operations' structure. For more information on the application of the Interface Template design pattern to these interfaces, refer to Section 5.5.1 in the Connectors 1.5 specification (`http://java.sun.com/j2ee/connector`).

Managed application scenario

The following steps are performed when a *managed application* requests to obtain a connection to an EIS instance from a connection factory, as specified in the `res-type` variable:

1. The application assembler or component provider specifies the connection factory requirements for an application component by using a deployment descriptor:

   ```
   res-ref-name: shme/shmeAdapter
   res-type:javax.resource.cci.ConnectionFactory
   res-auth: Application|Container
   ```

2. The Resource Adapter deployer sets the configuration information for the Resource Adapter.

3. VisiConnect uses a configured Resource Adapter to create physical connections to the underlying EIS.

4. The application component performs a JNDI lookup of a connection factory instance in the component's environment:

   ```
   // obtain the initial JNDI Naming context
    javax.naming.Context ctx = new javax.naming.InitialContext();
    // perform the JNDI lookup to obtain the connection factory
    javax.resource.cci.ConnectionFactory cxFactory =
           (javax.resource.cci.ConnectionFactory)ctx.lookup(
               "java:comp/env/shme/shmeAdapterConnectionFactory");
   ```

5. The JNDI name passed in the context lookup is that same as that specified in the `res-ref-element` of the component's deployment descriptor. The JNDI lookup returns a connection factory instance of type `java.resource.cci.ConnectionFactory` as specified in the `res-type` element.

6. The application component invokes the `getConnection()` method on the connection factory to request to obtain an EIS connection. The returned connection instance represents an application level handle to an underlying physical connection. An application component requests multiple connections by invoking the `getConnection()` method on the connection factory multiple times.

   ```
   javax.resource.cci.Connection cx = cxFactory.getConnection();
   ```

7. The application component uses the returned connection to access the underlying EIS. This is specific to the Resource Adapter.

8. After the component finishes with the connection, it closes it using the `close()` method on the connection interface.

   ```
   cx.close();
   ```

9. If the application component fails to close an allocated connection after its use, that connection is considered an unused connection. The AppServer manages to cleanup of unused connections. When the container terminates a component instance, the container cleans up all the connections used by that component instance.

Non-managed application scenario

In the non-managed application scenario, a similar programming model must be followed in the application component. The non-managed application must lookup a connection factory instance, request to obtain an EIS connection, use the connection for EIS interactions, and close the connection when completed.

The following steps are performed when a *non-managed application* component requests to obtain a connection to an EIS instance from a connection factory:

1. The application component calls the `getConnection()` method on the `javax.resource.cci.ConnectionFactory` instance to get a connection to the underlying EIS instance.
2. The connection factory instance delegates the connection request to the default connection manager instance. The Resource Adapter provides the default connection manager implementation.
3. The connection manager instance creates a new physical connection to the underlying EIS instance by calling the `ManagedConnectionFactory.createManagedConnection()` method.
4. Invoking `ManagedConnectionFactory.createManagedConnection()` creates a new physical connection to the underlying EIS, represented by the `ManagedConnection` instance it returns. The `ManagedConnectionFactory` uses the security information from the JAAS `Subject` object, and `ConnectionRequestInfo`, and its configured set of properties (port number, server name, etc.) to create the new `ManagedConnection` instance.
5. The connection manager instance calls the `ManagedConnection.getConnection()` method to get an application-level connection handle. This method call does not necessarily create a new physical connection to the EIS instance; it produces a temporary handle that is used by an application to access the underlying physical connection, represented by the `ManagedConnection` instance.
6. The connection manager instance returns the connection handle to the connection factory instance; the connection factory in turn returns the connection to the requesting application component.

Code excerpts—programming to the CCI

The following code excerpts illustrate the application programming model based on the CCI requesting to obtain a connection, obtaining the connection factory, creating the interaction and interaction spec, obtaining a record factory and records, executing the interaction with the records, and performing the same using result sets and custom records.

```
// Get a connection to an EIS instance after lookup of a connection factory
// instance from the JNDI namespace. In this case, the component allows the
// container to manage the EIS sign-on
javax.naming.Context ctx = new javax.naming.InitialContext();
javax.resource.cci.ConnectionFactory cxFactory =
(javax.resource.cci.ConnectionFactory)ctx.lookup(
  "java:comp/env/shme/shmeAdapter" );
javax.resource.cci.Connection cx = cxFactory.getConnection();

// Create an Interaction instance
javax.resource.cci.Interaction ix = ct.createInteraction();

// Create a new instance of the respective InteractionSpec
com.shme.shmeAdapter.InteractionSpecImpl ixSpec = new
com.shme.shmeAdapter.InteractionSpecImpl();
ixSpec.setFunctionName( "S_EXEC" );
ixSpec.setInteractionVerb( javax.resource.cci.InteractionSpec.SYNC_SEND_RECEIVE
);
// ...
```

```
// Get a RecordFactory instance
javax.resource.cci.RecordFactory recFactory = // ... get a RecordFactory

// Create a generic MappedRecord using the RecordFactory instance. This record
// instance acts as an input to the execution of an interaction. The name of
the
// Record acts as a pointer to the metadata for a specific record type
javax.resource.cci.MappedRecord input = recFactory.createMappedRecord(
"ShmeExecRecord" );

// Populate the generic MappedRecord instance with input values. The component
// code adds values based on the metadata it has accessed from the metadata
// repository
input.put( "<key: element0>", new String( "S_APP01"     );
input.put( "<key: element1>", // ... );
// ...

// Create a generic IndexedRecord to hold output values that are set by the
// execution of the interaction
javax.resource.cci.IndexedRecord output =
    recFactory.createIndexedRecord( "ShmeExecRecord" );

// Execute the Interaction
boolean response = ix.execute( ixSpec, input, output );

// Extract data from the output IndexedRecord. Note that type mapping is done
// in the generic IndexedRecord by mean of the type mapping information in the
// metadata repository. Since the component uses generic methods on the
// IndexedRecord, the component code performs the required type casting
java.util.Iterator iter = output.iterator();

while ( iter != null && iter.hasNext() )
{
  // Get a record element and extract value ...
}

// Set up the requirements for the ResultSet returned by the execution of
// an Interaction. This step is optional. Default values are used if
// requirements are not explicitly set.
com.shme.shmeAdapter.InteractionSpecImpl rsIxSpec =
    new com.shme.shmeAdapter.InteractionSpecImpl();
rsIxSpec.setFetchSize( 20 );
rsIxSpec,setResultSetType( javax.resource.cci.ResultSet.TYPE_SCROLL_INSENSITIVE
);

// Execute an Interaction that returns a ResultSet
javax.resource.cci.ResultSet rSet =
    (javax.resource.cci.ResultSet)ix.execute( rsIxSpec, input );

// Iterate over the ResultSet. The example here positions the cursor on the
// first row and then iterates forward through the contents of the ResultSet.
// Appropriate get methods are then used to retrieve column values.
rSet.beforeFirst();

while ( rSet != null && rSet.next() )
{
// get the column values for the current row using the appropriate
// get methods
}

// This illustrates reverse iteration through the ResultSet
rSet.afterLast();

while ( rSet.previous() )
```

```
{
// get the column values for the current row using the appropriate
// get methods
}

// Extend the Record interface to represent an EIS-specific custom Record.
// The interface CustomerRecord supports a simple accessor/mutator design
// pattern for its field values. A development tool would generate the
// implementation class of the CustomerRecord
public interface CustomerRecord extends javax.resource.cci.Record,
    javax.resource.cci.Streamable
{
  public void setName( String name );
  public void setId( String custId );
  public void setAddress( String address );

  public String getName();
  public String getId();
  public String getAddress();
}

// Create an empty CustomerRecord instance to hold output from
// the execution of an Interaction
CustomerRecord customer = // ... create an instance

// Create a PurchaseOrderRecord instance as an input to the Interaction
// and set properties on this instance. The PurchaseOrderRecord is another
// example of a custom Record
PurchaseOrderRecord purchaseOrder = // ... create an instance
purchaseOrder.setProductName( "..." );
purchaseOrder.setQuantity( "..." );
// ...

// Execute an Interaction that populates the output CustomerRecord instance
boolean crResponse = ix.execute( rsIxSpec, purchaseOrder, customer  );

// Check the CustomerRecord
System.out.println( "Customer Name = [" + customer.getName() + "],
                     Customer ID = [" + customer.getId() + "],
                     Customer Address = [" + customer.getAddress() + "]" );
```

Deployment Descriptors for Application Components

The application component deployment descriptors need to specify connection factory information for the Resource Adapter which the component will use. Appropriate entries are required in:

1. In the component's Sun standard deployment descriptor. For example, in ejb-jar.xml, the following is required:
   - `res-ref-name: shme/shmeAdapter`
   - `res-type: javax.resource.cci.ConnectionFactory`
   - `res-auth: Application|Container`
2. In addition, any version specific entries can be included. For example, EJB 2.0's res-sharing-scope:
   - `res-sharing-scope: Shareable|Unshareable`

3 In the component's Borland-specific deployment descriptor. For example, in ejb-borland.xml, the following is required:

- `res-ref-name: shme/shmeAdapter`
- `res-type: javax.resource.cci.ConnectionFactory`

4 In addition, any version specific entries can be included. For example, EJB 1.1's cmp-resource:

- `cmp-resource: True|False`

The following details example deployment descriptors for two EJBs—the first written to the EJB 2.0 spec, the second written to the EJB 1.1 spec. Both the standard and Borland-specific deployment descriptors are shown. In these examples, a hypothetical Resource Adapter is referenced.

EJB 2.x example

ejb-jar.xml deployment descriptor

This example uses container-managed persistence

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <display-name>SHME Integration Jar</display-name>
    <enterprise-beans>
        <session>
            <description>Interface EJB for shmeAdapter Class /shme/test/
              shmeAdapter/schema/Customer</description>
            <display-name>customer_bean</display-name>
            <ejb-name>shme/customer_bean</ejb-name>
            <home>com.shme.test.shmeAdapter.schema.CustomerHome</home>
            <remote>com.shme.test.shmeAdapter.schema.CustomerRemote</remote>
            <ejb-class>com.shme.test.shmeAdapter.schema.CustomerBean
              </ejb-class>
            <session-type>Stateful</session-type>
            <transaction-type>Container</transaction-type>
            <env-entry>
                <description>SHME Repository URL for Connector configuration
                   </description>
                <env-entry-name>repositoryUrl</env-entry-name>
                <env-entry-type>java.lang.String</env-entry-type>
                <env-entry-value>s_repository://S_APP01</env-entry-value>
            </env-entry>
            <env-entry>
                <description>Location of Resource Adapter Configuration within
                   the SHME Repository</description>
                <env-entry-name>configurationUrl</env-entry-name>
                <env-entry-type>java.lang.String</env-entry-type>
                <env-entry-value>/shme/client</env-entry-value>
            </env-entry>
            <resource-ref>
                <description>Reference to SHME Resource Adapter</description>
                <res-ref-name>shme/shmeAdapter</res-ref-name>
                <res-type>com.shme.shmeAdapter.ConnectionFactory</res-type>
                <res-auth>Container</res-auth>
                <res-sharing-scope>Shareable</res-sharing-scope>
            </resource-ref>
    </session>
    </enterprise-beans>
```

```
    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>customer_bean</ejb-name>
                <method-intf>Remote</method-intf>
                <method-name>s_exec_customer_query</method-name>
                <method-params/>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

This corresponds to the ejb-jar.xml above.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Borland Software Corporation//DTD Enterprise
   JavaBeans 2.0//EN" "http://www.borland.com/devsupport/appserver/dtds/
   ejb-jar_2_0-borland.dtd">
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>shme/customer_bean</ejb-name>
            <bean-home-name>shme/customer_bean</bean-home-name>
            <resource-ref>
                <res-ref-name>shme/shmeAdapter</res-ref-name>
                <jndi-name>eis/shmeAdapter</jndi-name>
            </resource-ref>
        </session>
    </enterprise-beans>
</ejb-jar>
```

EJB 1.1 example

ejb-jar.xml deployment descriptor

This example uses bean-managed persistence.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
   1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>
<ejb-jar>
 <description />
     <display-name>ShmeAdapter Interface Jar</display-name>
     <small-icon />
  <large-icon />
     <enterprise-beans>
      <session>
  <description>Interface EJB for SHME Class /shme/test/shmeAdapter/schema/
      Customer</description>
   <display-name>customer_bean</display-name>
  <ejb-name>shme/customer_bean</ejb-name>
 <home>com.shme.test.shmeAdapter.schema.CustomerHome</home>
  <remote>com.shme.test.shmeAdapter.schema.CustomerRemote</remote>
   <ejb-class>com.shme.test.shmeAdapter.schema.CustomerBean</ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Bean</transaction-type>
   <env-entry>
         <description>SHME Repository URL for Connector configuration
           </description>
      <env-entry-name>repositoryUrl</env-entry-name>
```

```
   <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>s_repository://S_APP01</env-entry-value>
   </env-entry>
 <env-entry>
  <description>Location of Resource Adapter configuration within the SHME
Repository</description>
     <env-entry-name>configurationUrl</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>/shme/client</env-entry-value>
  </env-entry>
 <resource-ref>
  <description>Reference to SHME Resource Adapter</description>
 <res-ref-name>shme/shmeAdapter</res-ref-name>
 <res-type>com.shme.shmeAdapter.ConnectionFactory</res-type>
 <res-auth>Container</res-auth>
 </resource-ref>
   </session>
</enterprise-beans>
<ejb-client-jar />
</ejb-jar>
```

ejb-inprise.xml deployment descriptor

This corresponds to the ejb-jar.xml above.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE inprise-specific PUBLIC '-//Inprise Corporation//DTD Enterprise
   JavaBeans 1.1//EN' 'http://www.borland.com/devsupport/appserver/dtds/
   ejb-inprise.dtd'>
<inprise-specific>
    <enterprise-beans>
        <session>
            <ejb-name>shme/customer_bean</ejb-name>
            <bean-home-name>shme/customer_bean</bean-home-name>
            <timeout>0</timeout>
            <resource-ref>
                <res-ref-name>shme/shmeAdapter</res-ref-name>
                <jndi-name>eis/shmeAdapter</jndi-name>
                <cmp-resource>False</cmp-resource>
            </resource-ref>
        </session>
    </enterprise-beans>
</inprise-specific>
```

Other Considerations

Working with Poorly Implemented Resource Adapters

Some commercially available Resource Adapters may be poorly implemented. As there does not yet exist any mechanism to test a Resource Adapter for compliance to the Connectors specs (as the J2EE Compatibility Test Suite (CTS) tests a Connectors implementation for spec compliance), it is currently not a simple task to recognize, but among the symptoms, you will find:

1. The Resource Adapter will exhibit strange errors during deployment
2. The Resource Adapter will exhibit strange errors during method invocation on the connection factory.

As VisiConnect strictly implements J2EE 1.4 and Connectors 1.5 requirements, it is often the only Connector Container which will detect poorly implemented Resource Adapters and not ignore the problem.

Examples of Poorly Implemented Resource Adapters

Generally, poorly implemented Resource Adapters are not compliant with the Connectors 1.5 specification. Examples of such Resource Adapters include:

- The Resource Adapter with a connection factory implementing only `java.io.Serializable`, and not both `java.io.Serializable` and `javax.resource.Referenceable` as per the Connectors specification (Section 10.5 “JNDI Configuration and Lookup”). The local JNDI context handlers of application servers such as AppServer can only register objects if they implement both interfaces. If a Resource Adapter implements a connection factory as `Serializable`, and doesn't implement `Referenceable`, you will see exceptions thrown when the application server attempts to deploy the connection factory to JNDI.
- The Resource Adapter with a connection factory which poorly implements `javax.resource.Referenceable` (which inherits `getReference()` from `javax.naming.Referenceable`). The J2SE 1.3.x and 1.4.x specs specify that for `javax.naming.Referenceable`, `getReference()` either:

  a. Returns a valid, non-null reference of the `Referenceable` object, or

  b. Throws an exception (`javax.naming.NamingException`).

  If the Resource Adapter implements `Referenceable` such that `getReference()` can (and will) return `null`, you will see exceptions thrown when a client attempts to invoke a connection factory method such as `getConnection()`.
- The Resource Adapter with a connection factory correctly implementing `Referenceable`, but which does not provide an implementation of `javax.naming.spi.ObjectFactory` (which is required by the Connectors specification (Section 10.5 “JNDI Configuration and Lookup”)). Although such a Resource Adapter can be deployed to an application server without incident, it cannot be deployed to JNDI outside the aegis of an application server, as a non-managed Connector. Also, including a `javax.naming.spi.ObjectFactory` implementation source Adapter with backup mechanism for JNDI `Reference`-based connection factory lookup.

- The Resource Adapter which specifies an connection factory or connection interface while not implementing that interface in its connection factory or connection class, respectively. Section 10.6 “Resource Adapter XML DTD” in the Connectors spec discusses the related requirements. To illustrate, let's say that in the `ra.xml` of a particular Resource Adapter, you have the following elements:

  ```
  //...
  <connection-interface>java.sql.Connection</connection-interface>
  <connection-impl-class>com.shme.shmeAdapter.ShmeConnection</connection-impl-
  class>
  //...
  ```

  But your implementation of `ShmeConnection` is as follows:

  ```
  package shme;
  public class ShmeConnection
  {
  private ShmeManagedConnection mc;
    public ShmeConnection( ShmeManagedConnection mc )
    {
     System.out.println( "In ShmeConnection" );
     this.mc = mc;
    }
  }
  ```

  Any attempt to invoke `getConnection()` on this Resource Adapter's connection factory will result in a `java.lang.ClassCastException`, as you're indicating to the appserver in `ra.xml` that connection objects returned by the Resource Adapter are to be cast to `java.sql.Connection`.

Working with a Poor Resource Adapter Implementation

To work around a poor Resource Adapter implementation, perform the following:

Extend the connection factory and/or connection class of the Connector, and have the extension correctly implement the poorly implemented code. For example, when dealing with a connection factory which implements `Serializable`, and doesn't implement `Referenceable` the idea is to extend the original connection factory to implement `Referenceable`, which means implementing `getReference()` and `setReference()`.

To illustrate, if the connection factory is `com.shme.BadConnectionFactory`, extend the connection factory as `com.shme.GoodConnectionFactory`, and implement `Referenceable` as follows:

```
package com.shme.shmeAdapter;

public class GoodConnectionFactory
{
 private javax.naming.Reference ref;
// ...
 public javax.naming.Reference getReference()
{
// implement such that getReference() never returns null
// ...
  return ref;
    }
public javax.naming.Reference setReference( javax.naming.Reference ref )
 //  this.ref = ref;
  }
 //
```

Also, when dealing with a poorly behaving `getReference()`, there are various ways to accomplish this, but principally, the idea is to implement `getReference()` such that it never returns `null`. The best approach is to implement:

- A fallback mechanism in `getReference()` which sets the reference to be returned correctly if the connection factory's reference attribute is `null`—returning a registerable javax.naming.Reference object, and
- A helper class implementing `javax.naming.spi.ObjectFactory` to provide the fallback object to create the connection factory object from the valid Reference instance.

To illustrate, if the connection factory is `com.shme.BadConnectionFactory`, extend the connection factory as `com.shme.GoodConnectionFactory`, and override `getReference()` as follows:

```
package com.shme.shmeAdapter;

public class GoodConnectionFactory
{
  // ...

  public javax.naming.Reference getReference()
  {
    if ( ref == null )
    {
      ref = new javax.naming.Reference( this.getClass().getName(),
        "com.shme.shmeAdapter.GoodCFObjectFactory"
          /* object factory for GoodConnectionFactory references */,
            null );
      String value;
      value = managedCxFactory.getClass().getName();

      if ( value != null )
      {
        ref.add( new javax.naming.StringRefAddr(
          "managedconnectionfactory-class", value ) );
      }

      value = cxManager.getClass().getName();

      if ( value != null )
      {
        ref.add( new javax.naming.StringRefAddr(
          "connectionmanager-class", value ) );
      }
    }

    return ref;
  }

  // ...
}
```

Then implement the associated object factory class, in this case:

```
com.shme.shmeAdapter.GoodCFObjectFactory
package com.shme.shmeAdapter;

import javax.naming.spi.*;
import javax.resource.spi.*;

public class GoodCFObjectFactory implements ObjectFactory {
  public GoodCFObjectFactory() {};

  public Object getObjectInstance( Object obj,
                                   javax.naming.Name name,
                                   javax.naming.Context context,
                                   java.util.Hashtable env )
         throws Exception
  {
    if ( !( obj instanceof javinstance ofReference ) )
    {
      return null;
    }

    javax.naming.Reference ref = (javax.naming.Reference)obj;

    if ( ref.getClassName().equals(
"com.shme.shmeAdapter.GoodConnectionFactory" ) )
    {
      ManagedConnectionFactory refMcf = null;
      ConnectionManager refCm = null;

if ( ref.get( "managedconnectionfactory-class" ) != null )
      {
        String managedCxFactoryStr =
         (String)ref.get( "managedconnectionfactory-class" ).getContent();
        Class mcfClass = Class.forName( managedCxFactoryStr );
        refMcf = (ManagedConnectionFactory)mcfClass.newInstance();
      }

if ( ref.get( "connectionmanager-class" ) != null )
      {
        String cxManagerStr = (String)ref.get( "connectionmanager-class"
).getContent();
        Class cxmClass = Class.forName( cxManagerStr  );
        java.lang.ClassLoader cloader = cxmClass.getClassLoader();
        refCm = (ConnectionManager)cxmClass.newInstance();
      }

      GoodConnectionFactory cf = null;

      if ( refCm != null )
      {
        cf = new GoodConnectionFactory( refMcf, refCm  );
      }
      else
      {
        cf = new GoodConnectionFactory( refMcf );
      }

    return cf;
    }

    return null;
  }
}
```

Update the classes in the ra.xml standard deployment descriptor file. For example, before extending the implementation, the `ra.xml` may look something like this:

```
<managedconnectionfactory-class>com.shme.shmeAdapter.
LocalTxManagedConnectionFactory</managedconnectionfactory-class>
<connectionfactory-interface>javax.sql.DataSource</connectionfactory-interface>
<connectionfactory-impl-class>com.shme.shmeAdapter.BadConnnectionFactory</
connectionfactory-impl-class>
<connection-interface>java.sql.Connection</connection-interface>
<connection-impl-class>com.shme.Connection</connection-impl-class>
```

After extending the interfaces, the ra.xml may look something like this:

```
<managedconnectionfactory-class>com.shme.shmeAdapter.
LocalTxManagedConnectionFactory </managedconnectionfactory-class>
<connectionfactory-interface>javax.sql.DataSource</connectionfactory-interface>
<connectionfactory-impl-class>com.shme.shmeAdapter.GoodConnectionFactory</
connectionfactory-impl-class>
<connection-interface>java.sql.Connection</connection-interface>
<connection-impl-class>com.shme.shmeAdapter.Connection</connection-impl-class>
```

As this illustrates, this conversion only impacts the connection factory. No other Resource Adapter classes are affected by this conversion.

Compile the Java code for the extended implementation (and any helper classes) into class files.

Package these into the Resource Adapter's Java Archive (.jar) file.

Update the Resource Adapter Archive (.rar) file with this extended .jar.

Deploy the Resource Adapter Archive, or include it in an Enterprise Application Archive (.ear) file to be deployed as part of a J2EE application, to VisiConnect running standalone or as a Partition service in the AppServer.

You've now converted a badly behaving Resource Adapter into a well behaving one.

Sometimes the design of a Resource Adapter makes it impossible to extend the existing API implementation. In such cases you need to re-implement the offending class or classes, and set the elements in `ra.xml` to reference the re-implementation(s). Or better yet, choose another Resource Adapter, which is compliant with the Connectors specification to work with.

Chapter

33

Borland AppServer Ant tasks and running AppServer examples

Many of the Borland AppServer (AppServer) examples now employ the Ant build script system. In addition to Ant's core functionality, the Borland AppServer version of Ant includes customized tasks for several of the AppServer command line tools, including commands of the following:

- `appclient`
- `iastool`
- `idl2java`
- `java2iiop`

These customized Ant tasks have the following advantages over using `exec` or `apply` directives:

- Customized Ant tasks run under the VM used to launch the Ant script, hence they run faster and use less memory compared to spawning new JVM's with the `exec`/`apply` commands.
- Customized tasks have a much simpler command syntax than the exec/apply version.
- Ant features such as filesets and patternsets are available in a more natural way.

General syntax and usage

The following table shows the currently defined Ant tasks and their relationship to the equivalent command line tools.

Ant task name	Equivalent command line	Function
appclient	appclient	Runs a client application.
idl2java	idl2java	Converts IDL to Java classes.
java2iiop	java2iiop	Executes the java2iiop command.
iastool	iastool	Runs iastool.

Generally the AppServer Ant task uses the same pattern as the command-line tool equivalent.

Name-value pair transformation

All name-value pair command line arguments can be transformed into Ant task attributes. The name-value pair command-line arguments should be translated into equivalent XML attributes. For example, the command line:

```
iastool -verify -src cart_beans_client.jar -role DEVELOPER
```

translates into the Ant task:

```
<iastool option="verify" src="cart_beans_client.jar" role="DEVELOPER" />
```

Name-only argument transformation

All name-only command-line arguments can be transformed into boolean type Ant task attributes. For boolean-style attributes, such as -nowarn, use the default usage of the attribute, according to the usage documented for each of the command line tools. See "iastool command-line utility" on page 311 for more information about iastool command line attributes.

For example, the following command sets the warn attribute to false:

```
iastool -verify -src cart_beans_client.jar -role DEVELOPER -nowarn -nostrict
```

The equivalent Ant task is:

```
<iasverify src="cart_beans_client.jar" role="DEPLOYER" nowarn="true"
strict="false" />
```

Note It is **not** valid to use "warn" as an attribute in the Ant task. For instance, the following line causes a syntax error:

```
****** INCORRECT SYNTAX!!! ******
<iasverify src="cart_beans_client.jar" role="DEPLOYER" warn="false"
strict="false" />
```

Multiple File Arguments

Many commands either act on multiple files or have options which can point to multiple files. There are several ways to achieve this functionality in the equivalent Ant task. For example, the `iastool -merge` command:

```
iastool -merge -target build\client.jar -type lib client\build\local_client.jar
build\local_stubs.jar
```

has the Ant equivalent:

```
<iastool option="merge" target="${build.dir}/client.jar" type="lib"
 jars="client/build/local_client.jar ; build/local_stubs.jar" />
```

Note The files in the `jars` attribute must be separated by semi-colons (;) or colons (:)—spaces and commas are **not** valid separators.

Ant provides a convenient `<fileset>` task to include multiple files:

```
<iastool option="merge" target="build/client.jar" type="lib" >
    <fileset dir="client/build" includes="local_client.jar" />
    <fileset dir="build" includes="local_stubs.jar" />
</iastool>
```

The patternset feature of Ant can also be useful. The following alteration now includes all of the jar files contained in the build directory and all of its sub-directories:

```
<iastool option="merge" target="${build.dir}/client.jar" type="lib" >
    <fileset dir="${build.dir}" includes="**/*.jar" />
</iastool>
```

Class path attributes can include multiple paths separated by semicolons:

```
<iastool option="verify" src="cart_beans_client.jar" role="DEPLOYER"
classpath="alib.jar;blib.jar" />
```

or use the <classpath> element:

```
<iastool option="verify" src="cart_beans_client.jar" role="DEPLOYER" >
   <classpath>
       <pathelement location="alib.jar" />
       <pathelement location="blib.jar" />
   </classpath>
</iastool>
```

Syntax and usage for iastool

The iastool Ant tasks can use two different styles:

1 <iastool option="*myoption*" />

2 <ias*myoption* />

For example, the command line:

```
iastool -verify -src cart_beans_client.jar
```

translates into the Ant task:

```
<iastool option="verify" src="cart_beans_client.jar" />
```

or you can use the older Ant style for backward compatible Ant tasks:

```
<iasverify src="cart_beans_client.jar" />
```

The following table shows the Ant task styles for each of the iastool options.

iastool Ant task style 1	iastool Ant task style 2	Equivalent command line	Function	Fileset attribute
<iastool option= "compilejsp" />	<iascompilejsp />	iastool -compilejsp	Precompiles JSP's.	
<iastool option= "compress" />	<iascompress />	iastool -compress	Compresses a JAR file.	
<iastool option= "deploy" />	<iasdeploy />	iastool -deploy	Deploys a J2EE module.	jars
<iastool option= "dumpstack" />	<iasdumpstack />	iastool -dumpstack	Dumps the stack trace of a Partition process to the stdout.log, located in: <install_dir>/var/domains/ <domain-name>/configurations/ <configuration-name>/ mos/<partition-name>/adm/logs/ <partition_name>.stdout.log	
<iastool option= "genclient" />	<iasgenclient />	iastool -genclient	Generates a client library.	jars
<iastool option= "gendeployable" />	<iasgendeployable />	iastool -gendeployable	Generates a manually deployable module.	
<iastool option= "genstubs" />	<iasgenstubs />	iastool -genstubs	Generate a stub library.	

iastool Ant task style 1	iastool Ant task style 2	Equivalent command line	Function	Fileset attribute
<iastool option= "info" />	<iasinfo />	iastool -info	Displays system configuration information.	
<iastool option= "kill" />	<iaskill />	iastool -kill	Kills a Managed Object.	
<iastool option= "listhubs" />	<iaslisthubs />	iastool -listhubs	Lists available Hubs on a management port.	
<iastool option= "listpartitions" />	<iaslistpartitions />	iastool -listpartitions	Lists the Partitions running on a Hub.	
<iastool option= "listservices" />	<iaslistservices />	iastool -listservices	Lists the services running on a Hub.	
<iastool option= "manage" />	<iasmanage />	iastool -manage	Actively manage a Managed Object.	
<iastool option= "merge" />	<iasmerge />	iastool -merge	Merges a set of JAR files into a single JAR file.	jars
<iastool option= "migrate" />	<iasmigrate />	iastool -migrate	Migrates a module from J2EE 1.2 to J2EE 1.3.	
<iastool option= "patch" />	<iaspatch />	iastool -patch	Applies a patch to a JAR file.	
<iastool option= "ping" />	<iasping />	iastool -ping	Pings a Managed Object or Hub for its current state.	
<iastool option= "pservice" />	<iaspservice />	iastool -pservice	Enables, disables, or gets the state of a Partition service.	
<iastool option= "removestubs" />	<iasremovestubs />	iastool -removestubs	Removes stubs from a JAR.	
<iastool option= "restart" />	<iasrestart />	iastool -restart	Restarts a Managed Object.	
<iastool option= "setmain" />	<iassetmain />	iastool -setmain	Sets the main class of a client JAR or a JAR in an EAR.	
<iastool option= "stop" />	<iasstop />	iastool -stop	Stops a Managed Object.	
<iastool option= "uncompress" />	<iasuncompress />	iastool -uncompress	Uncompresses a JAR file.	
<iastool option= "undeploy" />	<iasundeploy />	iastool -undeploy	Undeploys a Managed Object.	
<iastool option= "unmanage" />	<iasunmanage />	iastool -unmanage	Removes a Managed Object from active management.	
<iastool option= "verify" />	<iasverify />	iastool -verify	Verifies a J2EE module.	

Note The Fileset attributes column indicate attributes which can accept multiple file names. Such attributes can employ the Ant `<fileset>` element to designate these files. Techniques for including multiple files is explained in the "Multiple File Arguments" on page 302.

Omitting attributes

Omitting an attribute from the Ant task call has the same effect as omitting the option from the command line tool. Since some attributes are `true` by default, omitting an attribute does **not** necessarily set the attribute to `false`.

For more information on the default values of these options, see "iastool command-line utility" on page 311.

Examples of iastool Ant tasks

The following are a few examples to illustrate the usage details for iastool Ant tasks. For more details about the function of each iastool option and attribute, see "iastool command-line utility" on page 311.

deploy

```
<target name="deploy" description="Deploys the example to the server">
<iastool option="deploy" hub="${hub.name}" cfg="${cfg.name}"
 partition="${partition.name}" mgmtport="${default.mgmtport}"
 jars="${build.dir}/hello.ear;${bes.lib.dir}/../var/repository/archives/wars/
   bank_form.war"
 realm="${realm.name}" user="${server.user.name}" pwd="${server.user.pwd}"/>
</target>
```

merge

```
<iastool option="merge" target="${build.dir}/helloclient.jar" type="lib">
    <fileset dir="${build.dir}" includes="hello_stubs.jar" />
</iastool>
```

ping

```
<target name="ping">
<iastool option="ping" hub="${hub.name}" cfg="${cfg.name}"
 partition="${partition.name}" mgmtport="${default.mgmtport}"
 realm="${realm.name}" user="${server.user.name}" pwd="${server.user.pwd}" />
</target>
```

restart

```
<target name="iastoolrestart">
<iastool option="-restart" hub="${hub.name}" cfg="${cfg.name}"
 partition="${partition.name}" mgmtport="${default.mgmtport}"
 realm="${realm.name}" user="${server.user.name}" pwd="${server.user.pwd}" />
</target>
```

Syntax and usage for java2iiop

The `java2iiop` Ant task is very different from its command line tool. It is an exception to the Borland Ant task usage pattern. Ant task `java2iiop` takes classes in a directory instead of an individual file. The classpath attribute points to the directory containing the classes that need to be compiled by `java2iiop`. That classpath is Path-Like structure in Ant, and the usage of it is very flexible, but to use classpath with the `java2iiop` task you can only use one of the following styles:

1. Used as an attribute, its value only accepts colon- or semicolon-separated lists of locations:

```
<java2iiop classpath="${path1}:${path2}"/>
```

2. Used as nested classpath element.
   This takes the general form of:

```
<java2iiop>
   <classpath>
      <pathelement path="${path1}"/>
      <pathelement location="lib/helper.jar"/>
   </classpath>
</java2iiop>
```

   The location attribute specifies a single file or directory relative to the project's base directory (or an absolute filename), while the path attribute accepts colon- or semicolon-separated lists of locations. The path attribute is intended to be used with predefined paths. In any other case, multiple elements with location attributes should be preferred.

For details on the equivalent command line arguments for java2iiop, see “Programmer tools for Java” in the *VisiBroker for Java Developer’s Guide*.

Example of java2iiop Ant task

```
<target name="create_ejb_stubs" depends="home">
    <java2iiop root_dir="${stubsPath}" list_files="true"
classpath="${outputPath}" />
</target>
```

Syntax and usage for idl2java

The `idl2java` Ant task is similar to its equivalent command tool. It tasks nested Path-Like structure filesets which are equivalent to command line file inputs.

```
<idl2java>
    <fileset dir="server" includes="*.idl" />
</idl2java>
```

For details on the equivalent command line arguments for idl2java, see “Programmer tools for Java” in the *VisiBroker for Java Developer’s Guide*.

Attribute	Type	Required
classpath	Path	Yes
back_Compat_Mapping	boolean	No
bind	boolean	No
boa	boolean	No
comments	boolean	No
compile	boolean	No

Attribute	Type	Required
compiler	String	No
destDir	Path	No
dynamic_Marshal	boolean	No
examples	boolean	No
export_All	boolean	No
exported	String	No
gen_Included_Files	boolean	No
idl2package	String	No
idl_Strict	boolean	No
import	String	No
imported	String	No
include	File	No
invoke_Handler	boolean	No
line_Directives	boolean	No
list_Files	boolean	No
list_Includes	boolean	No
map_Keyword	String	No
narrow_Compliance	boolean	No
obj_Wrapper	boolean	No
object_Method	boolean	No
package	String	No
retain_Comments	boolean	No
root_Dir	File	No
sealed	String	No
servant	boolean	No
srcDir	Path	No
srcFile	String	No
stream_Marshal	boolean	No
strict	boolean	No
tie	boolean	No
undefine	String	No
VBJclassPath	Path	No
VBJdebug	String	No
VBJjavaVM	File	No
VBJprop	String	No
VBJquoteSpaces	String	No
VBJtag	String	No
version	String	No
warn_Missing_Define	String	No
warn_Unrecognized_Pragmas	boolean	No

Example of idl2java Ant task

```
<target name="idl2java" depends="init">
    <idl2java package="com.borland.examples.webservices.visibroker"
root_dir="${server-skel-src}">
        <fileset dir="server" includes="*.idl" />
    </idl2java>
    <javac srcdir="${server-skel-src}" destdir="${server-classes}"
classpathref="classpath"/>
</target>
```

Syntax and usage for appclient

```
<!-- Execute the example. -->
<target name="execute" description="Executes the Hello World example">
    <appclient jar="${build.dir}/hello.ear" uri="helloclient.jar" args="World"/
>
</target>
```

Building and running the Borland AppServer examples

Note Many of the AppServer examples have their own `readme.html` files located in:

```
<install_dir>/examples
```

To build an AppServer example:

1. Open a command line window.
2. Set the current directory to an example directory. The “Hello World” example located at `<install_dir>/examples/j2ee/hello` is a good place to start.
3. On the command line, enter “ant”.

   The example should build automatically.

Note The server does not have to be running to build the example. However, deployment and undeployment require that the server be operational. Executing an example requires that the Partition be running.

Deploying the example

1. Make sure that a server is running.
2. On the command line, enter `ant deploy`.

   This will deploy the example to the Hub, Configuration, and Partition set in the <install_dir>\examples\deploy.properties file.

If you wish to deploy to a different combination of Hub/Configuration/Partition, you can either edit the `deploy.properties` file to change the settings, or use `-D` options on the command line to override the `deploy.properties` settings.

For example, to use a Hub named “myhub", use the command:

```
ant -Dhub.name=myhub deploy
```

This will override the default Hub name in `deploy.properties` with the value `myhub`.

Running the example

1. Make sure that the Partition is running.
2. On the command line, enter `ant execute`.

   The precise response depends on the particular example.

Undeploying the example

1. Make sure that a server is running.
2. On the command line, enter `undeploy`.

Troubleshooting

1. Make sure that the `<appserver_install_dir>/bin` directory is on your path and precedes the path to any alternative Ant installations.
2. Before calling the `ant execute` command, make sure that the server and the Partition are running.
3. The `<appserver_install_dir>\examples\deploy.properties` contains default settings for the Hub, Configuration, Partition, and Management Port. These default properties include:

   - `hub.name=`*`your_machine_name`*
   - `cfg.name=j2ee`
   - `partition.name=standard`
   - `realm.name=ServerRealm`
   - `server.user.name=admin`
   - `server.user.pwd=admin`

   where *your_machine_name* is the machine name designated at installation. You can reset these values as needed or specify them on the Ant command line using the `-D` option.

Chapter

34

iastool command-line utility

This section describes the `iastool` command-line utility that you can use to manage your managed objects.

Using the iastool command-line tools

The `iastool` utility is a set of command-line tools for manipulating managed objects. The following table shows the command-line tools provided with the `iastool` utility:

Table 34.1 iastool command-line utilities

Use...	To...
`-compilejsp`	Precompile JSPs in a standalone WAR or all WARs in an EAR. For more information, see “compilejsp” on page 312.
`-compress`	Compress a JAR file. For more information, see “compress” on page 314.
`-deploy`	Deploy a J2EE module to the specified Partition. For more information, see “deploy” on page 315.
`-dumpstack`	Dump the stack trace of a Partition process to the Partition stdout.log file. For more information, see “dumpstack” on page 316.
`-genclient`	Generate a library containing client stubs, EJB interfaces, and dependent classes. For more information, see “genclient” on page 317.
`-gendeployable`	Generate a manually deployable module. For more information, see “gendeployable” on page 318.
`-genstubs`	Generate a library containing client or server stubs only. For more information, see “genstubs” on page 318.
`-info`	Display system configuration information. For more information, see “info” on page 319.
`-kill`	Kill a managed object. For more information, see “kill” on page 320.
`-listpartitions`	List the partitions on a hub. For more information, see “listpartitions” on page 321.
`-listhubs`	List the available hubs on a management port. For more information, see “listhubs” on page 322.
`-listservices`	List the services on a hub. For more information, see “listservices” on page 322.
`-manage`	Actively manage a managed object. For more information, see “manage” on page 323.

Table 34.1 iastool command-line utilities (continued)

Use...	To...
`-merge`	Merge a set of JAR files into a single JAR file. For more information, see "merge" on page 324.
`-migrate`	Migrate a module from J2EE 1.2, 1.3, or 1.4 to an alternative target J2EE version. For more information, see "migrate" on page 325.
`-newconfig`	Create a new configuration from a configuration template. For more information see "newconfig" on page 325.
`-patch`	Apply one or more patches to a JAR file. For more information, see "patch" on page 326.
`-ping`	Ping a managed object or hub for its current state. For more information, see "ping" on page 327.
`-pservice`	Enables, disables, or gets the state of a partition service. For more information, see "pservice" on page 328.
`-removestubs`	Remove all stub files from a JAR file. For more information, see "removestubs" on page 329.
`-restart`	Restart a hub or managed object. For more information, see "restart" on page 330.
`-setmain`	Set the main class of a standalone Client JAR or a Client JAR in an EAR. For more information, see "setmain" on page 331.
`-start`	Start a managed object. For more information, see "start" on page 332.
`-stop`	Stop a hub or managed object. For more information, see "stop" on page 333.
`-uncompress`	Uncompress a JAR file. For more information, see "uncompress" on page 334.
`-undeploy`	Remove a J2EE module from a Partition. For more information, see "undeploy" on page 334.
`-unmanage`	Remove a managed object from active management. For more information, see "unmanage" on page 335.
`-usage`	Display the usage of command-line options. For more information, see "usage" on page 336.
`-verify`	Verify a J2EE module. For more information, see "verify" on page 336.

compilejsp

Use this tool to precompile JSP pages in a standalone WAR or in all WARs in an EAR. The JSP pages are compiled into Java servlet classes and saved in a WAR file. This operation enables the JSP pages to be served faster the first time they are accessed.

Note When compiling JSPs using the iastool, you may encounter an out-of-memory error. Increase the size of the virtual memory on your system to solve this issue.

Syntax

```
-compilejsp -src <war_or_ear> -target <target_file> [-overwrite]
[-package <package_root>] [-excludefile <exclude_file>] [-loglevel <0-4>]
[-classpath <classpath>]
```

Default Output

By default, `compilejsp` reports if the operation was successful or not.

Options

The following table describes the options available when using the `compilejsp` tool.

Option	Description
`-src <war_or_ear>`	Specifies the WAR or EAR file you want to compile. The full or relative path to the file must be specified. There is no default.
`-target <target_file>`	Specifies the name of the target WAR or EAR archive file to be generated. If file name you specify already exists, use the `-overwrite` option to overwrite the previously existing target. The full or relative path to the file must be specified. There is no default.
`-overwrite`	Indicates that the <target_file> should be overwritten if it previously existed. If <target_file> exists but `-overwrite` is not used, you will get an error message saying that the target JAR must be different from the source JAR.
`-package <package_root>`	Specifies the base package name for the precompiled JSP servlet classes. The default is `com.bes.compiledjsp`.
`-excludefile <exclude_file>`	Specifies a text file containing a list of JSP files to exclude from the compile operation. See "Using the excludefile option" on page 313 for more details.
`-loglevel <0-4>`	Specifies the amount of output diagnostic messages to be generated. A value greater than 2 will also leave the temporary servlet Java files for further inspection. The default is 2.
`-classpath <classpath>`	Specifies any additional libraries that may be required for compiling the JSP pages. There is no default.

Example

To precompile the JSP pages contained in a WAR file called `proj1.war` located in the current directory into a WAR file called `proj1compiled.war` in the same location:

```
iastool -compilejsp -src proj1.war -target proj1compiled.war
```

To precompile the JSP pages contained in an EAR file called `proj1.ear` located in the directory `c:\myprojects\` into an EAR file called `proj1compiled.ear` in the same location and generate the maximum amount of diagnostic messages:

```
iastool -compilejsp -src c:\myprojects\proj1.ear -target
c:\myprojects\proj1compiled.ear -loglevel 4
```

Using the excludefile option

The `compilejsp excludefile` option allows you to specify a text file containing a list of JSPs to exclude from the compile operation. The following list details the usage rules.

- Comment lines (with a leading '#') and blank lines are ignored.
- Leading and trailing blank spaces and white spaces on each line are trimmed.
- Each line in the exclude file represents one exclude pattern entry, which can be a string for exact matching or a Java Pattern regular expression.
- Each JSP exclude entry is used to perform an exact match against a JSP URL first. If there is no match, the JSP exclude entry will be used as a regular expression to match against the JSP URL again.
- A JSP URL is compared against each of the JSP exclude entries using the above algorithm. As soon as there is a match, the JSP is excluded from compilation. If the JSP URL does not match any of the JSP exclude entries, the JSP will be compiled.
- If a pattern entry is not a valid Java regular expression, a warning is shown. It will still be used to compare against JSP URLs for exact match.

- If the `iastool -compilejsp -loglevel` option is set to 3 or higher, the exclude pattern entries, the number of excluded JSP pages, and the excluded JSP URLs are be displayed.
- If all of the JSP files in the archive are excluded the `-compilejsp` command will fail.

The following are some example exclude patterns:

```
# This pattern excludes a specific JSP: /jsp/test/test.jsp
/jsp/test/test[.]jsp

# This pattern excludes the JSP /jsp/test/test.jsp or /jsp/test/test2jsp, etc.
# because the regular expression "." represents any character
/jsp/test/test.jsp

# This pattern excludes all the files under the /include URL path
/include/.*

# This pattern excludes all the include.jsp files
.*/include[.]jsp

# This pattern excludes all the JSP files that start with "tmp_" in the /jsp
URL path
# and all JSP files in any URL path that starts with "tmp_" under /jsp
/jsp/tmp_.*[.]jsp

# This pattern excludes all the JSP files that start with "tmp_" in the /jsp
URL path
/jsp/tmp_[^/]*[.]jsp
```

compress

Use this tool to compress a JAR file.

Syntax

```
-compress -src <srcjar> -target <targetjar>
```

Default Output

By default, `compress` reports if the operation was successful or not.

Options

The following table describes the options available when using the `compress` tool.

Option	Description
`-src <srcjar>`	Specifies the JAR file that you want to compress. The full or relative path to the file must be specified. There is no default.
`-target <targetjar>`	Specifies the name of the compressed JAR file to be generated. The full or relative path to the file must be specified. There is no default.

Example

To compress a JAR file, called `proj1.jar` and located in the current directory, into a file called `proj1compress.jar` in the same location:

```
iastool -compress -src proj1.jar -target proj1compress.jar
```

To compress a JAR file called `proj1.jar` located in the directory `c:\myprojects\` into a file called `proj1compress.jar` in the same location:

```
iastool -compress -src c:\myprojects\proj1.jar
-target c:\myprojects\proj1compress.jar
```

deploy

Use this tool to deploy a J2EE module to a specified Partition on the specified hub and configuration.

Syntax

```
-deploy -jars <jar1,jar2,...> <-hub <hub>|-host <host>:listener_port>>
-cfg <configname> -partition <partitionname> [-force_restart] [-cp <classpath>]
[-args <args>] [-javac_args <args>] [-noverify] [-nostubs] [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

Default Output

By default, `deploy` reports if the operation was successful.

Options

The following table describes the options available when using the `deploy` tool.

Option	Description
`-jars <jar1,jar2...>`	Specifies the names of one or more JAR files to be deployed. To specify more than one JAR file, enter a comma (,) between each file name (no spaces). The full or relative path to the files must be specified. There is no default.
`-hub <hub>`	Specifies the name of the hub in which to deploy the JAR files.
`-host <host>:<listener_port>`	Specifies the host name and the listener port of the machine on which the partition you are interested in is running. This option enables the `iastool` utility to locate a partition on a different subnet than the machine on which `iastool` is running.
`-cfg <configname>`	Specifies the name of the configuration containing the partition in which you want to load the JAR file.
`partition <partitionname>`	Specifies the name of the Partition in which you want to load the JAR file.
`-force_restart`	Restarts the specified Partition after deploying the module. If this option is not specified, you will need to restart the Partition manually to initialize the module.
`-cp <classpath>`	Specifies the classpath containing the class dependencies of the JAR file(s) to be deployed.
`-args <args>`	Specifies any arguments that are needed by the JAR file. For details, see “Programmer tools for Java” in the *VisiBroker for Java Developer’s Guide*.
`-javac_args <args>`	Specifies any Java compiler arguments that are needed by the JAR file.
`-noverify`	Turns off verification of the active connections to a Partition on a specified management port.
`-nostubs`	Prevents creation of client or server-side stub files for the deployed module.
`-mgmtport <nnnnn>`	Specifies the management port number used by the specified hub. The default is 42424.
`-realm <realm>`	Specifies the realm used to authenticate a user when the user and password options are specified.
`-user <username>`	Specifies the user to authenticate against the specified realm.

Option	Description
-pwd <password>	Specifies the user's password to authenticate against the specified realm.
-file <login_file>	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See “Executing iastool command-line tools from a script file” on page 338 for more information.

dumpstack

Use this tool to obtain diagnostic information about the threads running in a Partition. This tool causes the Partition to generate a stack trace of all threads, and the output is stored in the Partition's stdout.log, located in:

```
<install_dir>/var/domains/<domain-name>/configurations/<configuration-name>/
     mos/<partition-name>/adm/logs/<partition_name>.stdout.log
```

. The stack trace may be useful for diagnosing problems with the Partition. The log file is located in the directory:

```
<install_dir>\var\domains\<domain_name>\configurations\<config_name>\
     <partition_name>\adm\logs\partition_log.xml
```

Syntax

```
-dumpstack <-hub <hub>|-host <host>:<listener_port>> -cfg <configname>
-partition <partitionname> [-mgmtport <nnnnn>] [-realm <realm>]
[-user <username>] [-pwd <password>] [-file <login_file>]
```

Options

The following table describes the options available when using the `dumpstack` tool.

Option	Description
-hub <hub>|-host <hostname>:<listener_port>	Specifies the name of the hub or the host name and the listener port of the machine on which the partition process you are interested in is running. You must specify either a hub name or a host name and listener port. Specifying a listener port enables the `iastool` utility to locate a hub on a different subnet than the machine on which `iastool` is running.
-cfg <configname>	Specifies the name of the configuration containing the specified partition.
-partition <partitionname>	Specifies the name of the Partition that you want to diagnose. The name of a valid Partition must be specified.
-mgmtport <nnnnn>	Specifies the management port number used by the specified hub. The default is 42424
-realm <realm>	Specifies the realm used to authenticate a user when the user and password options are specified.
-user <username>	Specifies the user to authenticate against the specified realm.
-pwd <password>	Specifies the user's password to authenticate against the specified realm.
-file <login_file>	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See “Executing iastool command-line tools from a script file” on page 338 for more information.

Examples

The following example shows how to perform a thread dump of the `standard` Partition in the `j2ee` configuration on the `BES1` hub:

```
iastool -dumpstack -hub BES1 -cfg j2ee -partition standard
```

The following example shows how to perform a thread dump of the `standard` Partition on a computer host on a specific listener port. Note that the `-host` option can be used regardless of whether `iastool` is executed on the same or a different host machine on which the partition is running.

```
iastool -dumpstack -host mymachine:1234 -cfg j2ee -partition standard
```

genclient

Use this tool to generate a library containing client stubs files, EJB interfaces, and dependent class files for one or more EJB JAR files, and to package them into one or more client JAR files. The client JAR is not an EJB, but is an EJB client.

If `genclient` fails for one of the EJB JARs in the argument list, an error is displayed and the `genclient` tool will continue to attempt to generate a client JAR on the remainder of the specified list.

The `genclient` tool will exit `0` (zero), for 100% success, or `1`, for any failure.

Syntax

```
-genclient -jars <jar1,jar2,...> -target <client_jar> [-cp <classpath>]
[-args <java2iiop_args>] [-javac_args <args>]
```

Default Output

The default output returns nothing to standard output (stdout).

Options

The following table describes the options available when using the `genclient` tool.

Option	Description
`-jars <jar1,jar2,...>`	Specifies one or more JAR files for which you want to generate one or more client JAR files. To specify more than one JAR file, enter a comma (,) between each file name (no spaces). The full or relative path to the JAR files must be specified. There is no default.
`-target <client_jar>`	Specifies the client-JAR files to be generated on the localhost. The full or relative path to the JAR files must be specified. There is no default.
`-cp <classpath>`	Specifies the classpath containing the class dependencies of the JAR file for which you want to generate a client JAR file. The default is none.
`-args <java2iiop_args>`	Specifies any arguments that are needed by the file. For details, see “Programmer tools for Java” in the *VisiBroker for Java Developer’s Guide*.
`-javac_args <args>`	Specifies any Java compiler arguments that are needed by the JAR file.

Example

The following example shows how to generate a manually deployable module client JAR file from each of the EJB JAR files: `proj1.jar`, `proj2.jar`, and `proj3.jar` into the EJB JAR `myproj.jar`.

```
iastool -genclient -jars proj1.jar,proj2.jar,proj3.jar -target myproj.jar
```

gendeployable

Use this tool to create a manually deployable server-side module. Server-side deployable JAR files are archives (EAR, WAR, or JAR beans only) that have been compiled to resolve all external code references by using stubs and are, therefore, ready for deployment.

For example, first use `gendeployable` to create the server-side deployable JAR file on a local machine, then use the `deploy` tool to copy and load it on the hub. The hub is advised of the presence of the new JAR file and loads it automatically. Using the command-line tools lets you script a creation and deployment to several servers quite simply. You can also manually copy the server-side deployable JAR file to the correct location on each hub, but this requires restarting each hub to cause it to be recognized and loaded.

Syntax

```
-gendeployable -src <input_jar> -target <output_jar> [-cp <classpath>]
[-args <java2iiop_args>] [-javac_args <args>]
```

Default Output

The default output returns nothing to standard output (stdout).

Options

The following table describes the options available when using the `gendeployable` tool.

Option	Description
`-src <input_jar>`	Specifies the JAR file (or the directory of an expanded JAR) that you want to use to generate a new deployable JAR file. The full or relative path to the JAR file must be specified. There is no default.
`-target <output_jar>`	Specifies the deployable JAR files to be generated on the localhost. The full or relative path to the JAR files must be specified. There is no default.
`-cp <classpath>`	Specifies the classpath containing the class dependencies of the JAR file for which you want to generate a client JAR file. The default is none.
`-args <java2iiop_args>`	Specifies any arguments that are needed by the file. For details, see “Programmer tools for Java” in the *VisiBroker for Java Developer’s Guide*.
`-javac_args <args>`	Specifies any Java compiler arguments that are needed by the JAR file.

Example

The following example shows how to generate a server-side deployable module JAR file for `proj1.jar` into the file `server-side.jar`.

```
iastool -gendeployable -src proj1.jar -target serverside.jar
```

genstubs

Use this tool to create a stubs library file containing client or server stubs.

Syntax

```
-genstubs -src <input_jar> -target <output_jar> [-client] [-cp <classpath>]
[-args <java2iiop_args>] [-javac_args <args>]
```

Default Output

The default output returns nothing to standard output (stdout).

Options

The following table describes the options available when using the `genstubs` tool.

Option	Description
`-src <input_jar>`	Specifies the JAR file (or the directory of an expanded JAR) for which you want to generate a stubs library. The full or relative path to the JAR file must be specified. There is no default.
`-target <output_jar>`	Specifies the name of the JAR file that will be generated on the localhost. The full or relative path to the JAR file(s) must be specified. There is no default.
`-client`	Specifies that you want to generate client-side stubs. If this option is not specified, the `genstubs` tool will generate server-side stubs.
`-cp <classpath>`	Specifies the classpath containing the class dependencies of the JAR file for which you want to generate a client JAR file(s). The default is none.
`-args <java2iiop_args>`	Specifies any arguments that are needed by the file. For details, see "Programmer tools for Java" in the *VisiBroker for Java Developer's Guide*.
`-javac_args <args>`	Specifies any Java compiler arguments that are needed by the JAR file.

Examples

The following example shows how to generate a server-side stubs of the EJB JAR `proj1.jar` into the EJB JAR `server-side.jar`.

```
iastool -genstubs -src proj1.jar -target serverside.jar
```

The following example shows how to generate a client-side stub file for the EJB JAR `myproj.jar` into the EJB JAR `client-side.jar`.

```
iastool -genstubs -src c:\dev\proj1.jar -target
-client c:\builds\client-side.jar
```

info

Use this tool to display the Java system properties for the JVM the `iastool` is running in.

Syntax

```
-info
```

Default Output

The default output is the current Java system properties for the JVM the `iastool` is running in. For example, the first few lines of output look like the following partial listing:

```
application.home              : C:\Program Files\AppServer
awt.toolkit                   : sun.awt.windows.WToolkit
file.encoding                 : Cp1252
file.encoding.pkg             : sun.io
file.separator                : \
java.awt.fonts                :
java.awt.graphicsenv          : sun.awt.Win32GraphicsEnvironment
java.awt.printerjob           : sun.awt.windows.WPrinterJob
java.class.path               : C:\Program Files\AppServer\jdk\lib\tools.jar
.
.
.
```

Example
The following example shows how to display configuration information.

```
iastool -info|more
```

kill

Use this tool to kill a managed object on a specified hub and configuration.

Syntax

```
-kill <-hub <hub>|-host <host>:listener_port>> -cfg <configname>
-mo <managedobjectname> -moagent <managedobjectagent> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

Default Output
By default, the `kill` tool lists the managed object that has been killed.

Options
The following table describes the options available when using the `kill` tool.

Option	Description
`-hub <hub>`	Specifies the name of the hub on which you want to kill a managed object.
`-host <host>:<listener_port>`	Specifies the host name and the listener port of the machine on which the managed object you are interested in is running. The option is enables the `iastool` utility to locate a hub on a different subnet than the machine on which `iastool` is running.
`-cfg <configname>`	Specifies the name of the configuration containing the specified managed object.
`-mo <managedobjectname>`	Specifies the name of the managed object.
`-moagent <managedobjectagent>`	Specifies the managed object agent name. Use this option if the specified hub has more than one agent.
`-mgmtport <nnnnn>`	Specifies the management port number used by the specified hub. The default is 42424.
`-realm <realm>`	Specifies the realm used to authenticate a user when the user and password options are specified.
`-user <username>`	Specifies the user to authenticate against the specified realm.
`-pwd <password>`	Specifies the user's password to authenticate against the specified realm.
`-file <login_file>`	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See “Executing iastool command-line tools from a script file” on page 338 for more information.

Examples
The following example kills the managed object `j2ee-server` using the default management port:

```
iastool -kill -hub AppServer1 -cfg j2ee -mo j2ee-server
```

The following example kills the partition naming service running on the configuration `j2ee` using the management port 24410:

```
iastool -kill -hub AppServer1 -cfg j2ee -mo standard_visinaming -mgmtport 24410
```

listpartitions

Use this tool to list the partitions running on a specified hub, and optionally on a specified configuration or management port.

Syntax

```
-listpartitions <-hub <hub>|-host <host>:<listener_port>>
[-cfg <configname>] [-mgmtport <nnnnn>] [-bare] [-realm <realm>]
[-user <username>] [-pwd <password>] [-file <login_file>]
```

Default Output

By default, the `listpartitions` tool displays the partitions running on a specified hub, or displays the partitions running on a specified hub on a specified configuration or management port.

Options

The following table describes the options available when using the `listpartitions` tool.

Option	Description
`-hub <hub>`	Specifies the hub name for which you want to list the running partitions.
`-host <host>:<listener_port>`	Specifies the host name and the listener port of the machine on which the partitions you are interested in are running. The option is enables the `iastool` utility to locate a hub on a different subnet than the machine on which `iastool` is running.
`-cfg <configname>`	Specifies the name of the configuration on which to list partitions.
`-mgmtport <nnnnn>`	Specifies the management port number used by the specified hub. The default is 42424.
`-bare`	Suppresses the output information, other than the names of the running partitions.
`-realm <realm>`	Specifies the realm used to authenticate a user when the user and password options are specified.
`-user <username>`	Specifies the user to authenticate against the specified realm.
`-pwd <password>`	Specifies the user's password to authenticate against the specified realm.
`-file <login_file>`	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See “Executing iastool command-line tools from a script file” on page 338 for more information.

Examples

The following example lists the partitions running on the hub `AppServer1` using the default management port:

```
iastool -listpartitions -hub AppServer1
```

The following example lists the partitions running on the hub `AppServer1` using the management port 24410:

```
iastool -listpartitions -hub AppServer1 -mgmtport 24100
```

listhubs

Use this tool to list hubs running on a particular management port located on the same local area network.

Syntax

```
-listhubs [-mgmtport <nnnnn>] [-bare] [-realm <realm>] [-user <username>]
[-pwd <password>] [-file <login_file>]
```

Default Output

By default, the `listhubs` tool displays the hubs running in the default management port or on a specified management port.

Note If a particular hub that is queried is down, it is not listed.

Options

The following table describes the options available when using the `listhubs` tool.

Option	Description
`-mgmtport <nnnnn>`	Specifies a management port number of the running hub you want to list. The default is 42424.
`-bare`	Suppresses output information, other than the names of the running hubs.
`-realm <realm>`	Specifies the realm used to authenticate a user when the user and password options are specified.
`-user <username>`	Specifies the user to authenticate against the specified realm.
`-pwd <password>`	Specifies the user's password to authenticate against the specified realm.
`-file <login_file>`	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See “Executing iastool command-line tools from a script file” on page 338 for more information.

Examples

The following example lists the hubs running in the default management port:

```
iastool -listhubs
```

The following example lists the hubs running in the management port 24410:

```
iastool -listhubs -mgmtport 24100
```

listservices

Use this tool to list one or more services running on a hub.

Syntax

```
-listservices  <-hub <hub>|-host <host>:<listener_port>> [-cfg <configname>]
[-mgmtport <nnnnn>] [-bare] [-realm <realm>] [-user <username>]
[-pwd <password>] [-file <login_file>]
```

Default Output

By default, `listservices` displays a list of all partition services registered for the specified hub on a particular management port.

Options

The following table describes the options available when using the `listservices` tool.

Option	Description
`-hub <hub>`	Specifies the name of the hub for which you want to list the running services.
`-host <host>:<listener_port>`	Specifies the host name and the listener port of the machine on which the services you are interested in are running. The option is enables the `iastool` utility to locate a hub on a different subnet than the machine on which `iastool` is running.
`-cfg <configname>`	Specifies the name of the configuration on which to list services.
`-mgmtport <nnnnn>`	Specifies the management port number used by the specified hub. The default is 42424.
`-bare`	Suppresses output of information other than the names of the running services.
`-realm <realm>`	Specifies the realm used to authenticate a user when the user and password options are specified.
`-user <username>`	Specifies the user to authenticate against the specified realm.
`-pwd <password>`	Specifies the user's password to authenticate against the specified realm.
`-file <login_file>`	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See “Executing iastool command-line tools from a script file” on page 338 for more information.

Example

The following example lists all services running on the `salsa` hub:

```
iastool -listservices -hub salsa
```

manage

Use this tool to actively manage a managed object in a configuration.

Syntax

```
-manage  (-hub <hub>|-host <host>:<listener_port>) [-cfg <configname>] -mo
<managedobjectname> [-moagent <managedobjectagent>] [-mgmtport <99999>] [-realm
<realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

Default Output

The default output returns nothing to standard output (stdout).

Options

The following table describes the options available when using the `manage` tool.

Option	Description
`-hub <hub>`	Specifies the name of the hub on which the managed object is running.
`-host <host>:<listener_port>`	Specifies the host name and the listener port of the machine on which the managed object is running. This option enables the iastool utility to locate a hub on a different subnet than the machine on which iastool is running.
`-cfg <configname>`	Specifies the name of the configuration with which the managed object is associated.
`-mo <managedobjectname>`	Specifies name of the managed object.
`-moagent <managedobjectagent>`	Specifies the agent with which the managed object is associated.

Option	Description
`-mgmtport <99999>`	Specifies the port with which the managed object's agent is associated.
`-realm <realm>`	Specifies the realm used to authenticate a user when the user and password options are specified.
`-user <username>`	Specifies the user to authenticate against the specified realm.
`-pwd <password>`	Specifies the user's password to authenticate against the specified realm.
`-file <login_file>`	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See "Executing iastool command-line tools from a script file" on page 338 for more information.

Example

The following example puts the managed object j2ee-server into the actively managed mode using the default management port:

```
iastool -manage -hub AppServer1 -cfg j2ee -mo j2ee-server
```

merge

Use this tool to produce a single, new Java Archive file (EJB-JAR) containing the contents of a specified list of EJB-JARs. Multiple EJB 1.1 and EJB 2.0 deployment descriptors (if any) will be consolidated into a single deployment descriptor. If merging fails for one of the EJB-JARs in the argument list an error is displayed and the merge command will exit indicating failure.

Syntax

```
-merge -jars <jar1,jar2,...> -target <new_jar> -type <valid_type>
```

Default Output

The default output returns nothing to standard output (stdout).

Options

The following table describes the options available when using the `merge` tool.

Option	Description
`-jars <jar1,jar2,..>`	Specifies the JAR files to merge, comma separated and no spaces. The full or relative path to the JAR files must be specified. There is no default.
`-target <new_jar>`	Specifies the name of the new JAR file to be created containing the merged contents of the specified list of JAR files. The full or relative path to the new JAR file must be specified. There is no default.
`-type` *`valid_type`*	Specifies the type of the new archive file using one of the following supported formats: ■ `ejb2.0` – Version 2.0 Enterprise Java Bean ■ `ejb1.1` – Version 1.1 Enterprise Java Bean ■ `ear1.3` – Version 1.3 Enterprise Application Resource ■ `ear1.2` – Version 1.2 Enterprise Application Resource ■ `lib` – Library file ■ `war2.3` – Version 2.3 Web Application Archive ■ `war2.2` – Version 2.2 Web Application Archive ■ `rar1.0` – Version 1.0 Resource Adapter Archive ■ `client1.2` – Version 1.2 Client JAR ■ `client1.3` – Version 1.3 Client JAR ■ `jndi1.2` – Version 1.2 Java Naming and Directory Interface

Example

The following example merges the EJB-JAR files `proj1.jar`, `proj2.jar`, and `proj3.jar` into a new version 2.0 EJB-JAR file named `combined.jar`:

```
iastool -merge -jars proj1.jar,proj2.jar,proj2.jar
-target combined.jar -type ejb2.0
```

migrate

Use this tool to convert a JAR or XML file from one version of J2EE to another, for example, J2EE version 1.2 to J2EE version 1.3 or J2EE 1.4.

Note The `migrate` command only converts the deployment descriptor for an EJB; as such, code changes may also be required to implement the conversion properly in your deployment.

If the conversion fails, an error is displayed.

Syntax

```
-migrate [-to[1.2|1.3|1.4]] -src <src-archive> -target <target-archive>
```

Default Output

The default returns nothing to standard output (stdout).

Options

The following table describes the options available when using the `migrate` tool.

Option	Description
`-to[1.2|1.3|1.4]`	Specifies the target version that the source J2EE module should be migrated to. For instance, a J2EE 1.3 source module can be migrated to J2EE 1.2 or J2EE 1.4, and a J2EE 1.2 module can be migrated to a target J2EE 1.3 or J2EE 1.4 module. If you do not specify this option then it defaults to J2EE 1.4.
`-src <src-archive>`	Specifies the J2EE module to convert. The full or relative path to the archive file (or the directory of an expanded JAR) must be specified. There is no default.
`-target <target-archive>`	Specifies the name of the J2EE module to be created. The full or relative path to the archive file must be specified. There is no default.

Example

The following example migrates the file `myj1_2.jar` from J2EE version 1.2 to J2EE version 1.3 into new file called `myj1_3.jar`:

```
iastool -migrate -src myj1_2.jar -target myj1_4.jar //here -to 1.4 is implied
iastool -migrate -to 1.3 -src myj1_2.jar -target myj1_3.jar //here a 1.2 module
is converted to 1.3
```

newconfig

Use this tool to create a new configuration from a configuration template. The command takes the name of a new configuration, the filename and path of a template file relative to the installation's configuration template directory, and an optional property file that overrides the properties in the template used to create the new configuration.

Syntax

```
-newconfig (-hub <hub>|-host <host>:<listener_port>) -cfg <configname>
   -template <template_path> [-property <property_path>] [-mgmtport <99999>]
   [-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

Default Output

The default output returns nothing to standard output (stdout).

Options

Option	Description
`-hub <hub>`	Specifies the name of the hub in which to create the new configuration.
`-host <host>:<listener_port>`	Specifies the host name and the listener port of the machine on which the hub is running to create the new configuration. This option enables the iastool utility to locate a hub on a different subnet than the machine on which iastool is running.
`-cfg <configname>`	Specifies the name of the new configuration.
`-template <template_path>`	Specifies the path where the configuration template is located. `template_path` can be either a full path of a configuration template XML file or a path relative to the configuration template directory (i.e. <install_dir>/var/templates/configurations/).
`-property <property_path>`	Specifies the path to an optional property file that overrides the properties used in the template to create the new configuration.
`-mgmtport <99999>`	Specifies the management port number used by the specified hub. The default is 42424.
`-realm <realm>`	Specifies the realm used to authenticate a user when the user and password options are specified.
`-user <username>`	Specifies the user to authenticate against the specified realm.
`-pwd <password>`	Specifies the user's password to authenticate against the specified realm.
`-file <login_file>`	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See “Executing iastool command-line tools from a script file” on page 338 for more information.

Example

```
iastool -newconfig -hub myhub -cfg SimpleProcessConfig -template native.xml
   -property c:\simple.properties
```

patch

Use this tool to apply one or more patches to a JAR file and produce a new JAR file with the applied patches.

Syntax

```
-patch -src <original_jar> -patches <patch1_jar,...> -target <new_jar>
```

Default Output

The default output displays the patches that were applied.

Options

The following table describes the options available when using the `patch` tool.

Option	Description
`-src <original_jar>`	Specifies the JAR file to which you want to apply one or more patches. The full or relative path to the JAR file must be specified. There is no default.
`-patches <patch1_jar,...>`	Specifies one or more JAR files that contain the patches you want to apply. To specify more than one file, enter a comma (,) between each file name (no spaces). The full or relative path to the files must be specified. There is no default.
`-target <new_jar>`	Specifies the name of the new JAR file to be created. The full or relative path to the JAR file must be specified. There is no default.

Example

The following example applies the patches contained in the files `mypatch1.jar` and `mypatch2.jar` to the file `myold.jar` which are all located in the current directory and creates a new file called `mynew.jar` in the same location:

```
iastool -patch -src myold.jar -patches mypatch1.jar,mypatch2.jar
-target mynew.jar
```

ping

Use this tool to verify the current state of a hub or a managed object. The ping command will return nothing for a hub that is not running.

Syntax

```
-ping <-hub <hub>|-host <host>:<listener_port>> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

or

```
-ping <-hub <hub>|-host <host>:<listener_port>> -cfg <configname>
-mo <managedobjectname> -moagent <managedobjectagent> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

Default Output

The default output shows the name and status of the hub (and optionally the managed object) if the process is pinged and running. For example:

```
Pinging Hub xyz_corp1: Running
```

The `ping` tool returns one of the following states:

- Running
- Starting
- Stopping
- Not Running
- Restarting
- Cannot Load
- Cannot Start
- Terminated
- Unknown

Options

The following table describes the options available when using the `ping` tool.

Option	Description
`-hub <hub>`	Specifies the hub to ping or whose services to ping. There is no default.
`-host <host>:<listener_port>`	Specifies the host name and the listener port of the machine on which the hub or managed object you are interested in is running. The option is enables the `iastool` utility to locate a managed object on a different subnet than the machine on which `iastool` is running.
`-cfg <configname>`	Specifies the name of the configuration on which to ping for managed objects.
`-mo <managedobjectname>`	Specifies the name of the managed object.
`-moagent <managedobjectagent>`	Specifies the managed object agent name. Use this option if the specified hub has more than one agent.
`-mgmtport <nnnnn>`	Specifies the management port number used by the specified hub. The default is 42424.
`-realm <realm>`	Specifies the realm used to authenticate a user when the user and password options are specified.
`-user <username>`	Specifies the user to authenticate against the specified realm.
`-pwd <password>`	Specifies the user's password to authenticate against the specified realm.
`-file <login_file>`	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See "Executing iastool command-line tools from a script file" on page 338 for more information.

Examples

The following example pings the hub `AppServer1` in the default management port:

```
iastool -ping -hub AppServer1
```

The following example pings the partition naming service running on the hub AppServer1 in the management port 24410:

```
iastool -ping -hub AppServer1 -cfg j2ee -mo standard_visinaming -mgmtport 24410
```

pservice

Use this tool to enable, disable, or get the state of a partition service.

Syntax

```
-pservice <hub <hub>|-host <host>:<listener_port>> -cfg <configname>
-partition <partitionname> -moagent <managedobjectagent>
-service <servicename> <-enable|-disable|-status> [-force_restart]
[-mgmtport <nnnnn>] [-realm <realm>] [-user <username>] [-pwd <password>]
[-file <login_file>]
```

Default Output

The default output returns nothing to standard output (stdout).

Options

The following table describes the options available when using the `pservice` tool.

Option	Description
`-hub <hub>`	Specifies the hub where the partition service you are interested in is located. There is no default.
`-host <host>:<listener_port>`	Specifies the host name and the listener port of the machine on which the partition service you are interested in is running. The option is enables the `iastool` utility to locate a partition service on a different subnet than the machine on which `iastool` is running.
`-partition <partitionname>`	Specifies the name of the partition.
`-moagent <managedobjectagent>`	Specifies the managed object agent name. Use this option if the specified hub has more than one agent.
`-service <servicename>`	Specifies the name of the service.
`-enable|-disable|-status`	Specifies the operation you would like to perform on the partition service.
`-force_restart`	Restarts the specified Partition after completing the enable, disable, or status operation. If this option is not specified, you will need to restart the Partition manually to initialize the module.
`-mgmtport <nnnnn>`	Specifies the management port number used by the specified hub. The default is 42424.
`-realm <realm>`	Specifies the realm used to authenticate a user when the user and password options are specified.
`-user <username>`	Specifies the user to authenticate against the specified realm.
`-pwd <password>`	Specifies the user's password to authenticate against the specified realm.
`-file <login_file>`	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See “Executing iastool command-line tools from a script file” on page 338 for more information.

Example

The following example shows how to enable the partition naming service on the standard partition.

```
iastool -pservice -hub AppServer1 -cfg j2ee -partition standard
-service standard_visinaming -enable -force_restart -mgmtport 24431
```

removestubs

Use this tool to remove all stub files from a JAR file.

Syntax

```
-removestubs -jars <jar1,jar2,...> [-targetdir <dir>]
```

Default Output

The default output returns nothing to standard output (stdout).

Options

The following table describes the options available when using the `removestubs` tool.

Option	Description
`-jars <jar1,jar2...>`	Specifies the JAR file(s) from which you want to remove one or more stub files. To specify more than one JAR file, enter a comma(,) between each JAR file (no spaces). The full or relative path to the JAR file(s) must be specified. There is no default.
`-targetdir <dir>`	Specifies the directory in which the stub files that were removed will be stored. A full or relative path must be specified, if this option is specified. There is no default. If no target directory is specified, the stub files will be removed, but not saved.

Example

The following example shows how to remove stub files located from the EJB JAR files `proj1.jar`, `proj2.jar`, and `proj3.jar` located in the current directory and copy them to `c:\examples\proto`:

```
iastool -removestubs -jars proj1.jar,proj2.jar,proj3.jar
-targetdir c:\examples\proto
```

restart

Use this tool to restart a hub or managed object. The hub must already be running in order for the `restart` tool to work with a hub.

Syntax

```
-restart <-hub <hub>|-host <host>:<listener_port>> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

or

```
-restart <-hub <hub>|-host <host>:<listener_port>> [-cfg <configname>]
-mo <managedobjectname> -moagent <managedobjectagent> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

Default Output

The default output displays the hub or managed object that has been restarted.

If the `restart` tool fails (for example, when a managed object cannot be shutdown or restarted), an error is displayed with a status code which is returned to standard error output (stderr).

Options

The following table describes the options available when using the `restart` tool.

Option	Description
`-hub <hub>`	Specifies the name of the hub that you want to restart. Also used to locate a managed object on a particular hub.
`-host <host>:<listener_port>`	Specifies the host name and the listener port of the machine on which the managed object you are interested in is running. The option is enables the `iastool` utility to locate a managed object on a different subnet than the machine on which `iastool` is running.
`-cfg <configname>`	Specifies the name of the configuration on which to locate managed objects.
`-mo <managedobjectname>`	Specifies the name of the managed object.
`-moagent <managedobjectagent>`	Specifies the managed object agent name. Use this option if the specified hub has more than one agent.

Option	Description
`-mgmtport nnnnn`	Specifies the management port number used by the specified hub. The default is 42424.
`-realm realm`	Specifies the realm used to authenticate a user when the user and password options are specified.
`-user username`	Specifies the user to authenticate against the specified realm.
`-pwd password`	Specifies the user's password to authenticate against the specified realm.
`-file <login_file>`	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See "Executing iastool command-line tools from a script file" on page 338 for more information.

Examples

The following example restarts the hub `AppServer1` on the default management port:

```
iastool -restart -hub AppServer1
```

The following example restarts the partition naming service running on the hub `AppServer1` on the management port 24410:

```
iastool -restart -hub AppServer1 -cfg j2ee -mo standard_visinaming  -mgmtport
24410
```

setmain

Use this tool to set the main class of a standalone Client JAR or a Client JAR in an EAR file. Once the main class is set, the `java -jar jarfile` command will automatically invoke the main class that has been set for the JAR file.

Syntax

```
-setmain -jar <jar_or_ear> [-uri <client_jar_in_ear>] -class <main_classname>
```

Default Output

The default output displays the main class that has been set for the specified JAR file.

Options

The following table describes the options available when using the `setmain` tool.

Option	Description
`-jar <jar_or_ear>`	Specifies the name of the JAR or EAR file on which you want to set the main class.
`-uri <client_jar_in_ear>`	If you are setting the main class for an EAR file, you must use the `-uri` option to identify the URI (Uniform Resource Identifier) path of the client JAR in the EAR.
`-class <main_classname>`	Specifies the class name that will be set as the main class in the specified Client JAR. The class must exist in the client JAR file and contain a `main()` method.

Examples

The following example sets a main class for a standalone Client JAR:

```
iastool -setmain -jar myclient.jar -class com.bes.myjclass
```

The following example sets a main class for a Client JAR contained in an EAR file:

```
iastool -setmain -jar myapp.ear -uri base/myapps/myclient.jar
-class com.bes.myjclass
```

start

Use this tool to start a managed object on a specified hub and configuration.

Syntax

```
-start <-hub <hub>|-host <host>:<listener_port>> -cfg <configname>
-mo <managedobjectname> -moagent <managedobjectagent> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

Default Output

The default output displays the managed object that has been started.

Options

The following table describes the options available when using the `start` tool.

Option	Description
`-hub <hub>`	Specifies the name of the hub on which the managed object you want to start is located.
`-host <host>:<listener_port>`	Specifies the host name and the listener port of the machine on which the managed object you are interested in is running. The option is enables the `iastool` utility to locate a hub on a different subnet than the machine on which `iastool` is running.
`-cfg <configname>`	Specifies the name of the configuration containing the managed object you are interested in.
`-mo <managedobjectname>`	Specifies the name of the managed object you are interested in.
`-moagent <managedobjectagent>`	Specifies the managed object agent name. Use this option if the specified hub has more than one agent.
`-mgmtport <nnnnn>`	Specifies the management port number used by the specified hub. If not specified, the default is 42424.
`-realm <realm>`	Specifies the realm used to authenticate a user when the user and password options are specified.
`-user <username>`	Specifies the user to authenticate against the specified realm.
`-pwd <password>`	Specifies the user's password to authenticate against the specified realm.
`-file <login_file>`	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See “Executing iastool command-line tools from a script file” on page 338 for more information.

Example

The following example starts the partition naming service running on the hub `AppServer1` in the `j2ee` configuration on management port 24410:

```
iastool -start -hub AppServer1 -cfg j2ee -mo standard_visinaming  -mgmtport
24410
```

stop

Use this tool to shut down a hub or managed object.

Syntax

```
-stop <-hub <hub>|-host <host>:<listener_port>> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

or

```
-stop <-hub <hub>|-host <host>:<listener_port>> [-mgmtport <nnnnn>]
-cfg <configname> -mo <managedobjectname> -moagent <managedobjectagent>
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

Default Output

The default output displays the process or processes that have been shut down.

If the `stop` tool fails (for example when a managed object cannot be shutdown), an error is displayed with a status code, which is returned to standard error output (stderr).

Options

The following table describes the options available when using the `stop` tool.

Option	Description
`-hub <hub>`	Specifies the name of the hub that you want to shut down, or the hub on which resides the managed object you want to shut down.
`-host <host>:<listener_port>`	Specifies the host name and the listener port of the machine on which the hub or managed object you are interested in is running. The option is enables the `iastool` utility to locate a hub on a different subnet than the machine on which `iastool` is running.
`-cfg <configname>`	Specifies the name of the configuration containing the managed object you are interested in.
`-mo <managedobjectname>`	Specifies the name of the managed object you are interested in.
`-moagent <managedobjectagent>`	Specifies the managed object agent name. Use this option if the specified hub has more than one agent.
`-mgmtport <nnnnn>`	Specifies the management port number used by the specified hub. The default is 42424.
`-realm <realm>`	Specifies the realm used to authenticate a user when the user and password options are specified.
`-user <username>`	Specifies the user to authenticate against the specified realm.
`-pwd <password>`	Specifies the user's password to authenticate against the specified realm.
`-file <login_file>`	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See "Executing iastool command-line tools from a script file" on page 338 for more information.

Example

The following example stops the partition naming service running on the hub `AppServer1` in the `j2ee` configuration on the management port 24410:

```
iastool -stop -hub AppServer1 -cfg j2ee -mo standard_visinaming -mgmtport 24410
```

uncompress

Use this tool to uncompress a JAR file.

Syntax

```
-uncompress -src <srcjar> -target <targetjar>
```

Default Output

By default, `uncompress` reports if the operation was successful or not.

Options

The following table describes the options available when using the `uncompress` tool.

Option	Description
`-src <srcjar>`	Specifies the JAR file that you want to uncompress. The full or relative path to the file must be specified. There is no default.
`-target <targetjar>`	Specifies the name of the uncompressed JAR file to be generated. The full or relative path to the file must be specified. There is no default.

Examples

The following example converts the compressed JAR file called `small.jar` located in the current directory into an uncompressed file called `big.jar` in the same location:

```
iastool -uncompress -src small.jar -target big.jar
```

The following example uncompresses a JAR file named `small.jar` located in the directory `c:\myprojects\` into a file named `big.jar` in the same location:

```
iastool -uncompress -src c:\myprojects\small.jar -target c:\myprojects\big.jar
```

undeploy

Use this tool to undeploy a J2EE module from a specified Partition on a specified hub and configuration.

Syntax

```
-undeploy -jar <jar> <-hub <hub>|-host <host>:<listener_port>>
-cfg <config_name> -partition <partitionname> [-mgmtport <nnnnn>]
[-realm <realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

Default Output

By default, the `undeploy` tool reports if the operation was successful or not.

Options

The following table describes the options available when using the `undeploy` tool.

Option	Description
`-jar <jar>`	Specifies the name of the JAR file to be undeployed. The full or relative path to the file must be specified. There is no default.
`-hub <hub>`	Specifies the name of the hub from which to undeploy the JAR file.
`-host <host>:<listener_port>`	Specifies the host name and the listener port of the machine on which the deployed module you are interested in is located. The option is enables the `iastool` utility to locate a module on a different subnet than the machine on which `iastool` is running.

Option	Description
-cfg <configname>	Specifies the configuration name under which the partition is configured.
-partition <partitionname>	Specifies the name of the Partition that contains the JAR file.
-mgmtport <nnnnn>	Specifies the management port number used by the specified hub. If not specified, the default is 42424.
-realm <realm>	Specifies the realm used to authenticate a user when the user and password options are specified.
-user <username>	Specifies the user to authenticate against the specified realm.
-pwd <password>	Specifies the user's password to authenticate against the specified realm.
-file <login_file>	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See "Executing iastool command-line tools from a script file" on page 338 for more information.

unmanage

Use this tool to remove a managed object from the actively managed mode.

Syntax

```
-unmanage  (-hub <hub>|-host <host>:<listener_port>) [-cfg <configname>] -mo
<managedobjectname> [-moagent <managedobjectagent>] [-mgmtport <99999>] [-realm
<realm>] [-user <username>] [-pwd <password>] [-file <login_file>]
```

Default Output

The default output returns nothing to standard output (stdout).

Options

The following table describes the options available when using the `unmanage` tool.

Option	Description
-hub <hub>	Specifies the name of the hub on which the managed object is running.
-host <host>:<listener_port>	Specifies the host name and the listener port of the machine on which the managed object is running. This option enables the iastool utility to locate a hub on a different subnet than the machine on which iastool is running.
-cfg <configname>	Specifies the name of the configuration with which the managed object is associated.
-mo <managedobjectname>	Specifies name of the managed object.
-moagent <managedobjectagent>	Specifies the agent with which the managed object is associated.
-mgmtport <99999>	Specifies the port with which the managed object's agent is associated.
-realm <realm>	Specifies the realm used to authenticate a user when the user and password options are specified.
-user <username>	Specifies the user to authenticate against the specified realm.
-pwd <password>	Specifies the user's password to authenticate against the specified realm.
-file <login_file>	Specifies a login script file containing the realm, user name , and password used to authenticate a user. The full or relative path to this file must be specified. See "Executing iastool command-line tools from a script file" on page 338 for more information.

Example

The following example removes the managed object j2ee-server from the actively managed mode using the default management port:

```
iastool -unmanage -hub AppServer1 -cfg j2ee -mo j2ee-server
```

usage

When invoked without arguments `usage` displays a list of recognized command-line options and a brief description of each. Invoking `usage` with one or more arguments provides a more detailed description of the specified commands and their arguments.

Syntax

```
-usage
-usage <tool>
-usage <tool1 tool2 tool3>
```

Note Arguments to the `usage` command do not require a leading hyphen.

Default Output

By default, the `usage` tool displays a list with a brief description of each command-line tool.

Examples

The following example displays a list and a brief description of each the command-line tools:

```
iastool -usage
```

The following example displays detailed information on the `compress` tool:

```
iastool -usage compress
```

The following example displays detailed information on the `-start`, `-stop`, and `-restart` tools:

```
iastool -usage start stop restart
```

verify

Use this tool to check an archive file for correctness and consistency, and to check if all of the elements required for deploying your application are in place.

The verification process supports the following roles that correspond to a phase in the application's life cycle and the appropriate level of verification (similar to the J2EE role definitions):

- **Developer:** This is the lowest verification level. All xml is checked for syntax as well as standard and proprietary keywords relevant to the current archive type. Consistency across the archive is checked, but no external resources are verified at this level.
- **Assembler:** Once the archives are individually verified and are correct, other resources built into an application will start to be verified. For example, this level will verify the existence and correctness of URIs (Uniform Resource Identifiers), but not EJB or JNDI links.
- **Deployer:** (the default) All checks are turned on. EJB and JNDI links are checked at this level as well as the operational environment in which the application is to be deployed.

Supported archive types are EAR, EJB, WAR, JNDI, and Client JARs. The typical archive verification process includes the following checks:

- A pass over the XML, checking for correct XML syntax.
- Verification of the semantics of the standard and proprietary XML descriptors, and the compliance with their required descriptors for each supported archive type.

Verification always occurs in a hierarchical fashion, starting with the top module, then recursively working through its submodules, and finally checking for inter-archive links.

Syntax

```
-verify -src <srcjar> [-role <DEVELOPER|ASSEMBLER|DEPLOYER>] [-nowarn]
[-strict] [-classpath <classpath>]
```

Default Output

By default, `verify` reports nothing (for example, if no errors are found in the specified module).

Options

The following table describes the options available when using the `verify` tool.

Option	Description
`-src <srcjar>`	Specifies the JAR file (or the directory of an expanded JAR) that you want to verify. The full or relative path to the file must be specified. There is no default.
`-role <DEVELOPER|ASSEMBLER|DEPLOYER>`	Specifies the level of error checking to perform: ■ `DEVELOPER` ■ `ASSEMBLER` ■ `DEPLOYER` (default) For details, see the role descriptions above.
`-nowarn`	Specifies that the tool should only report errors that preclude deployment and not report warnings.
`-strict`	Specifies that the tool should report the most minute inconsistencies, many of which do not affect the overall integrity of the application.
`-classpath <classpath>`	Specifies the search path for application classes and resources. To enter more than one directory, ZIP, or JAR file entry, separate each entry with a semicolon (;).

Example

The following example performs a *developer* level verification of the JAR file `soap-client.jar` located in the `c:\examples\soap` directory:

```
-verify -src c:\examples\soap\soap-client.jar -role DEVELOPER
```

Executing iastool command-line tools from a script file

Several `iastool` utility tools require that you supply login information (realm, username, and password). You may, however, want to run `iastool` commands from a script file, but doing so would expose the realm, username, password information to anyone who has access to the script file. There are two methods you can use to protect this information:

- Enter the realm, username, and password information into a file and pipe that file to the command.
- Enter the realm, username, password information into a file and pass the file to the command with the -file option.

Piping a file to the iastool utility

The following example shows how to ping a hub named `east1` by piping the file `mylogin.txt` (located in the default Borland Deployment Platform installation directory) to the `iastool` utility:

```
iastool -ping -hub east1 < c:\AppServer\mylogin.txt
```

where the file `mylogin.txt` contains three lines that correspond to what you would enter for the realm, username, and password:

```
2
username
password
```

Note The contents of the file are exactly what you would enter on the command-line. The first entry in the file is the `realm` option—not the realm name, but the number you would choose from the list presented to you if you run the `ping` tool without the `realm` option. The second line is the `username` and the third line is the `password`. This file can then be secured in such a way that it is readable by the `iastool` utility, but not by unauthorized users.

Passing a file to the iastool utility

The following command shows how to ping a hub named `east1` by passing a file to the `iastool` utility using the `-file` option:

```
iastool -ping -hub east1 -file c:\AppServer\mylogin.txt
```

where `mylogin.txt` has the following format:

```
Default Login
Smart Agent port number
username
password
false
ServerRealm
```

The `-file` option requires that you supply a fully qualified file name (the file name plus a relative or absolute path). When passing a file to the `iastool` utility, only the third, fourth, and sixth lines are used, which are the `username`, `password`, and `realm name`, respectively. The other lines must be present, but the information they contain is ignored by the `iastool` utility. For example:

```
Default Login
12448
myusername
mypassword
false
ServerRealm
```

Chapter

35

Partition XML reference

This section describes the XML definition of a Partition's `partition.xml` configuration file that contains the core meta-data for a Partition's configuration.

<partition> element

The `partition` element is the root node of the schema which contains the attributes and sub-elements that define the settings that control the configuration of a Borland AppServer (AppServer) Partition.

Attribute	Description
`version`	Product version of the Partition.
`name`	The name of the Partition.
`description`	A description of the Partition.

Syntax

```
<partition version="version number" name="partition name"
description="description">
   ⋮
</partition>
```

The `partition` element contains the following sub-elements:

- `<jmx>`
- `<statistics.agent>`
- `<security>`
- `<container>`
- `<user.orb>`
- `<management.orb>`
- `<shutdown>`
- `<services>`
- `<archives>`

<jmx> element

The `jmx` element contains the sub-elements for configuring the JMX agent. See JMX support in Partitions for more information about the AppServer JMX implementation.

The `jmx` element contains the following sub-elements:

- `<mbean.server>`
- `<mlet.service>`
- `<http.adaptor>`
- `<rmi-iiop.adaptor>`

<mbean.server> element

The `mbean.server` element is used to enable or disable the JMX agent's MBean Server. The MBean Server is an interface and a factory object defined by the agent specification level of the JMX.

Attribute	Description
`enable`	Enables or disables the MBean Server. Valid values are `true` (default) or `false`.

<mlet.service> element

The `mlet.service` element configures the JMX agent's MLet service. The MLet service is useful for loading MBean classes and resources inside an MBean Server's JVM from a remote host and registering the MBeans in a single action.

Attribute	Description
`enable`	Enables or disables the MLet service. Valid values are `true` or `false` (default).

<http.adaptor> element

The `http.adaptor` element configures the JMX agent's HTTP adaptor. The HTTP adaptor is the adaptor for the HTTP protocol through which the Partition can be managed through any HTML 3.2 compliant browser or application.

Attribute	Description
`enable`	Enables or disables the HTTP adaptor. Valid values are `true` or `false` (default).
`port`	The port number at which the HTTP adaptor is listening. 8082 is the default port number.
`host`	Defines the host name which the server will be listening to. If left with the default setting (`localhost`) you cannot access the server from another computer. This is good for security reasons, forcing you to explicitly open the server to the outside. You can also use 0.0.0.0 which will open the server to all local interfaces. Default is `localhost`.
`authentication.method`	Sets the authentication method. Valid values are `none`, `basic`, and `digest`. Default is `none`.
`socket.factory.name`	Replaces the default socket factory with another using an ObjectName, the pointed MBean has to have a `public ServerSocket createServerSocket(int port, int backlog, String host) throws IOException` method.
`processor.name`	This sets the MBean's ObjectName to be used as XML processor. The MBean has to implement the `mx4j.tools.adaptor.http.ProcessorMBean` interface.

The `http.adaptor` element contains the following sub-element:

- `<xslt.processor>`

<xslt.processor> element

The `xslt.processor` element configures the XSLT processor for the HTTP adaptor. The XSLT processor transforms the raw XML into presentable HTML for the Web browser. If this property is not enabled and you try to use the MX4J Web console you will get raw XML in your Web browser.

Attribute	Description
`enable`	Enables or disables the XSLT processor. Valid values are `true` (default) or `false`.
`File`	Determines where to look for XSL files. If the target file is a directory, it assumes that XSL files are located in the directory. Otherwise if it points to a JAR or ZIP file, it assumes that the files are located inside. Pointing to a file system is especially useful for testing.
`PathInJar`	Sets the directory in the JAR file where XSL files reside.
`LocaleString`	Sets the locale by using a String. Default is `en`.
`UseCache`	Indicates whether to cache the transformation objects. This speeds up the transformation process. It is usually set to `true`, but you can set it to `false` for easier testing. Default is `true`.

<rmi-iiop.adaptor> element

The `rmi-iiop.adaptor` element configures the JMX agent's RMI-IIOP adaptor. The RMI-IIOP adaptor is based on the client framework, which helps managers or managing applications to communicate with the MBean Server through RMI.

Attribute	Description
`enable`	Enables or disables the RMI-IIOP adaptor. Valid values are `true` (default) or `false`.
`port`	The port number to be allocated for the RMI-IIOP adaptor port. If you do not specify a port number or specify 0 as the port number, a random port number is assigned.

<statistics.agent> element

The `statistics.agent` element configures the Partition's statistics agent. The Partition statistics agent consists of two components:

- A statistics collector that periodically collects statistics data on the Partition and saves that data onto disk. These periodic data samples build up on disk enabling the product tools to provide current, and historical current statistical data on a Partition.
- A statistics reaper that periodically *reaps* (cleans up) the historical data from disk.

The Partition statistics agent is intended for collecting short term statistical data. However, it is only physically limited by the amount of disk space it is allowed to consume.

Attribute	Description
`enable`	Enables or disables the statistics agent. A disabled statistics agent will not collect or reap statistics data. Valid values are `true` (default) or `false`.
`level`	Sets the level of detail of statistics collected from a Partition. Valid values are: `none`, `minimum` (default), and `maximum`.
`snapshot.period_secs`	Specifies how often (in seconds) Partition statistics are collected and written to the disk. The default is `10` seconds.
`reap.enable`	Enables or disables the reaping (clean up) of Partition statistics data on disk. Valid values are `true` (default) or `false`.
`reap.older_than_secs`	If `reap_enable` is `true`, sets the threshold for the age (in seconds) of statistics data kept on disk before being deleted. The default is `600` seconds (10 minutes).
`reap.period_secs`	If `reap_enable` is `true`, sets the time period (in seconds) between sweeps to clean up statistics data older than `reap.older_than_secs` from disk. The default is `60` seconds (1 minute).

<security> element

The security element lets you configure the security settings for a given Partition. This empty element contains the attributes described in the following table.

Attribute	Description
enable	Enables or disables security for a Partition. Valid values are true (default) or false.
manager	Specifies the name of the security manager used by a Partition. Valid values are any available security provider names, for example com.borland.security.provider.CertificateWallet.
policy	Specifies the name of the security policy file that defines the security rules of a Partition. Valid values are any fully qualified security policy file names, for example <install_dir>/va/\security/profile/\management/java_security.policy

<container> element

The container element specifies how the Partition works with classloading.

Attribute	Description
system.classload.prefixes	This is a comma separated list of resource prefixes that the custom classloader will delegate to the system classloader prior to attempting to load itself.
verify.on.load	When true runs verify on JARs as they are loaded. Default is true.
classloader.policy	Determines the type of classloader to be used by the Partition. Valid values are per_module or container. The per_module classloader policy will create a separate application classloader for each deployed module. This policy is required if you want to be able to hot deploy. The container policy will load all deployed modules in the shared classloader. You cannot hot deploy if this policy is selected.
classloader.classpath	Contains a semicolon (;) separated list of JAR files to be loaded by each instance of the application classloader. This has the same logical effect as bundling these jars in every module.

<user.orb> element

The user.orb element controls the VisiBroker configuration used for the Partition's user domain ORB.

Attribute	Description
orb.propstorage	Path to the Partition's user ORB properties file. Relative paths are relative to the Partition's properties directory (the directory partition.xml is in).
use.default.smartagent.port	This property defines whether the Partition will use the SCU Smart Agent configuration to determine the Smart Agent port value.
use.default.smartagent.addr	This property defines whether the Partition will use the SCU Smart Agent configuration to determine the Smart Agent host address value.

<management.orb> element

The `management.orb` element controls the VisiBroker configuration for the Partition's management domain ORB.

Attribute	Description
`orb.propstorage`	Path to the Partition's management domain ORB properties file.
`required_roles.propstorage`	Path to the Partition's management domain ORB required roles configuration file.
`runas.propstorage`	Path to the Partition's management domain ORB `runas` configuration file.

All the paths are relative to the Partition's properties directory (the directory `partition.xml` is in).

<shutdown> element

The `shutdown` element determines the actions taken when a Partition stops. This empty element has no attributes.

Attribute	Description
`dump_threads`	Flag that causes the Partition to dump diagnostic information on threads still running late in Partition shutdown.
`dump_threads.count`	If defined, the value indicates the number of times to dump the thread states during shutdown. It is useful if you are trying to see if some threads are simply taking a long time to quit, but do quit eventually.
`delay.1`	Reserved for support use.
`garbage_collection.1`	Reserved for support use.
`delay.2`	Reserved for support use.
`runfinalizersonexit`	Reserved for support use.
`delay.3`	Reserved for support use.
`garbage_collection.2`	Reserved for support use.
`delay.4`	Reserved for support use.
`runfinalization`	Reserved for support use.

<services> element

The services element lets you configure the Partition's services. Each Partition service has a service sub-element with its specific configuration, the services element itself has the following attributes:

Attribute	Description
autostart	List of Partition's services to be started at Partition startup. The value is a space separated list of Partition service names.
startorder	The startup order to be imposed on the Partition services configured to be started by the autostart attribute. Partition services that are not specified are started after those specified. A valid value is a space separated list of Partition service names in their start order (left to right).
shutdownorder	The shutdown order to be imposed on the Partition services that are running at Partition shutdown. Partition services that are not specified are stopped before those specified. A valid value is a space separated list of Partition service names in their shutdown order (left to right).
administer	List of Partition services that are visible to the user. They appear in the tools when Partition services are listed.

The <services> element contains the following sub-element:

- service

<service> element

The <service> element provides the configuration for a Partition service. It contains attributes that govern the Partition's management of the service and a properties sub-element that contains the service's configuration metadata.

Attribute	Description
name	The Partition service's name.
version	The version of the Partition service.
description	The description for the Partition service.
vendor	The description of the vendor for the Partition service.
class	The class that implements the Partition's service plugin architecture and provides the management and control interface for the service.
in.management.domain	Flag that indicates if the service runs in the Partition's management domain or in the Partition's user domain.
startup.synchronization	The type of synchronization to be performed when the service is started. Valid values are: ■ service_ready—wait for the service to be ready for up to startup.service_ready.max_wait milliseconds. ■ delay—always wait for startup.delay milliseconds, do not monitor the service for it to become ready. Default is no synchronization.
startup.service_ready.max_wait	Limits the maximum time, in milliseconds, that the Partition waits for the service to start when the startup.synchronization value is service_ready. A value of 0 (zero) means no time limit is imposed. The default value is 0 (zero).
startup.delay	Defines the time, in milliseconds, that the Partition waits in order to give the service a chance to start when the startup.synchronization value is delay. A value of 0 (zero) means wait forever. Default is 0 (zero) .

Attribute	Description
shutdown.synchronization	The type of synchronization to be performed when the service is shutdown. Valid values are: ■ service_shutdown—wait for the service to stop for up to shutdown.service_shutdown.max_wait milliseconds. ■ delay—always wait for shutdown.delay milliseconds, do not monitor the service for it to stop. Default is no synchronization.
shutdown.service_shutdown.max_wait	Limits the maximum time, in milliseconds, that the Partition waits for the service to stop when the shutdown.synchronization value is service_shutdown. A value of 0 (zero) means no time limit is imposed. Default value is 0 (zero).
shutdown.delay	Defines the time, in milliseconds, that the Partition waits in order to give the service a chance to stop when the shutdown.synchronization value is delay. A value of 0 (zero) means wait forever. Default is 0 (zero).
shutdown.phase	This property governs which Partition shutdown phase the service is shutdown in. A Partition shuts down in 2 phases. In the first phase all services and components providing user facility are shutdown, and in the second phase the Partition's own infrastructure is shutdown. Valid values are 1 (default) and 2. It is not typical for any Partition service to be shutdown in phase 2.

<properties> element

The properties element lets you supply the specific service's configuration metadata.

<archives> element

The archives element contains configuration metadata for the archives that the Partition can host. A specific archive can have an archive sub-element with attributes specific to that archive. An archive does not have to have an archive sub-element.

Attribute	Description
ear.repository.path	Path to the Partition's EARs directory. All EARs found in that directory are loaded by the Partition on startup, unless specifically disabled with an archive element.
war.repository.path	Path to the Partition's WARs directory. All WARs found in that directory are loaded by the Partition on startup, unless specifically disabled with an archive element.
ejbjar.repository.path	Path to the Partition's EJB jars directory. All EJB jars found in that directory are loaded by the Partition on startup, unless specifically disabled with an archive element.
rar.repository.path	Path to the Partition's RARs directory. All RARs found in that directory are loaded by the Partition on startup, unless specifically disabled with an archive element.
dar.repository.path	Path to the Partition's DARs directory. All DARs found in that directory are loaded by the Partition on startup, unless specifically disabled with an archive element.
lib.repository.path	Path to the Partition's lib directory. All JAR files found in that directory are placed on the Partition's system classpath.
classes.repository.path	Path to the Partition's classes directory. All classes found in that directory are placed on the Partition's system classpath.

All the paths are relative to the Partition's root directory.

<archive> element

The `archive` element contains configuration metadata specific to an archive. Archives that are found in the Partition's archive repository directories do not need an archive element unless there is non-default configuration that need to be applied to them.

Attribute	Description
`name`	Name of the archive to which this element pertains. Is the filename of the archive.
`disable`	Flag for disabling the hosting of that archive in the Partition at startup. Valid values are `true` or `false` (default).
`path`	Path to an archive that exists outside of the Partition repositories. Use to get the Partition to host an archive from a specified path.

All the paths are relative to the Partition's root directory.

Chapter 36

EJB, JSS, and JTS Properties

EJB Container-level Properties

Set EJB container properties in `partition.xml` (each Partition has its own properties file). This file is located in the following directory:

```
<install_dir>/var/domains/base/configurations/configuration_name/mos/
partition_name/adm/properties
```

Property	Description	Default
`ejb.copy_arguments=true|false`	This flag causes arguments to be copied in intra-bean in-process calls. By default, intra-bean calls use pass-by-reference semantics. Enable this flag to cause intra-bean calls to use pass-by-value semantics. **Note:** A number of EJBs will run significantly slower using pass-by-value semantics.	`false`
`ejb.use_java_serialization=true|false`	If set it overrides use of IIOP serialization with Java serialization for things like session passivation, and so forth.	`false`

Property	Description	Default
ejb.useDynamicStubs=true|false	This property is only relevant for CMP 2.0 entity beans that provide local interfaces. If set, the Container, which otherwise uses CORBA to dispatch calls, uses a dynamic proxy-based scheme to dispatch calls (creating custom lightweight, non-CORBA references). These local dynamic stubs provide many optimizations, especially due to the callers and callees being in the same VM, making a direct dispatch to the beans without going through the CORBA layer. Also, since the dynamic stubs are aware of the EJB container data structures, they access the target beans more quickly. Note that currently the stub generator, java2iiop (called using the iastool or directly) still generates the stubs for all the interfaces in the archive. When ejb.useDynamicStubs is active, the subset of stubs that correspond to the selected CMP 2.0 beans are ignored. This feature, when used, makes the whole dispatch mechanism dynamic, providing dynamic stubs for the client side as well as dynamic skeletons on the server side. Any statically generated stub and skeleton classes in the archive are ignored. You set the property in the bean. However, if there isn't an issue with using the property in all the entity beans, the easiest way is to set it at the EAR level in the deployment descriptor. **Important:** You must use this property in conjunction with ejb.usePKHashCodeAndEquals.	true
ejb.usePKHashCodeAndEquals=true|false	Data structures that support Active Cache (TxReady cache) and Associated Cache (Ready beans cache) use java.util.Hashtable, and java.util.HashMap. The values (entity bean instances) pooled in these data structures are keyed on the primary key values of the cached entity beans. As we know, the implementation of Hashtable relies on computing hashCode() and calling equals() methods of the keys to place and locate the values. These data structures are in the critical code path and are accessed frequently by the container while dispatching calls to methods in entity beans The default in Borland AppServer (AppServer) is a reflection-based computation. When this property is set, the container uses a user supplied implementation of the equals() and hashCode() methods.	true
ejb.no_sleep=true|false	Typically set from a main program that embeds a Container. Setting this property prevents the EJB container from blocking the current thread, thereby returning the control back to user code.	false
ejb.trace_container=true|false	Turns on useful debugging information that tells the user what the Container is doing. Installs debugging message interceptors.	false
ejb.xml_validation=true|false	If set, the XML descriptors are validated against its DTD at deployment time.	true
ejb.xml_verification=true|false	If set, J2EE archive is verified at deployment time.	false
ejb.classload_policy=per_module|container|none	Defines class loading behavior of standalone EJB container. Not applicable to the Partition. If set to per_module, the container uses a new instance of custom class loader with each J2EE archive deployed. If set to none, the container uses the system class loader. Hot-deployment and deployment of EARs does not work in this mode. If set to container, container uses single custom class loader. This enables deployment of EARs, but disables hot-deployment feature.	per_module
ejb.module_preload=true|false	Loads the entire J2EE archive into memory at deployment time, so the archive can be overwritten or rebuilt. This option is required by JBuilder running a standalone ejb container.	false
ejb.system_classpath_first=true|false	If set to true, the custom classloader will look at the system classpath first.	false

Property	Description	Default
ejb.sfsb.keep_alive_timeout=<num>	Defines the default value of the <timeout> element used in the ejb-borland.xml descriptor. This property affects an EJB whose <timeout> element is skipped or set to 0. The purpose of this property is to define a time interval in seconds how long to keep an inactive stateful session bean alive in the persistent storage (JSS) after it was passivated. After the time interval ends, JSS deletes the session's state from the persistent storage, so it becomes impossible to activate it later.	86400 (=24 hours)
ejb.cacheTimeout=<integer>	This property hints the container to invalidate the data fields of entity beans after a specified time-out period. Use the property by specifying the interval for which the container will not load a bean's state from the database, but uses the cached state instead. At the end of the expire period specified, the container marks the bean as dirty (but keeps its association with the primary key), forcing the instance to load its state from the database (not the cache) before it can be used in any new transactions. The property is expected to be used by entity beans that are not frequently modified. The property is a positive integer representing cache intervals in seconds. This is only valid for commit mode A. It is ignored if specified for any other commit mode.	0 (no timeout).
ejb.sfsb.aggressive_passivation=true|false	If set to true, stateful session bean is passivated no matter when it was used last time. This enables fail-over support, so if an EJB container fails, the session can be restored from the last saved state by one of EJB containers in the cluster. If set to false, only the beans which were not used since the last passivation attempt, are passivated to JSS. This makes the fail-over support less deterministic, but speeds things up. Use this setting, to trade performance for high-availability.	true
ejb.sfsb.factory_name=<string>	If set, makes the stateful session beans use a different JSS from the one that is running within the same EJB container or Partition. Specify the factory name of JSS to use. This is the name under which JSS is registered with Smart Agent (osagent).	none
ejb.logging.verbose=true|false	If set to true, the EJB container logs messages about unexpected situations which potentially could require user's attention. The messages are marked with >>>> EJB LOG <<<< header. Set it to false, to suppress these messages.	true
ejb.logging.doFullExceptionLogging=true|false	If set, the container logs all unexpected exceptions thrown in an EJB implementation.	false
ejb.jss.pstore_location=<path>	Use this to override the default name and location of the file used as a JSS backend storage. Applicable only for standalone ejb containers. This option is deprecated, use jss.pstore and jss.workingDir instead.	none
ejb.jdb.pstore_location=<path>	Use this to override the default name and location of the file used by the database service. Applicable only for standalone ejb containers.	none
ejb.interop.marshal_handle_as_ior=true|false	If set to true, each instance of javax.ejb.Handle is marshaled as a CORBA IOR. Otherwise, it is marshaled as a CORBA abstract interface. See CORBA IIOP spec for details.	false
ejb.finder.no_custom_marshal=true|false	When a multi-object finder returns a collection of objects, by default the EJB container does the following: ■ creates and returns a custom Vector implementation to the caller. ■ creates IORs (from the primary keys) lazily as the caller of the finder browses/iterates over the Vector returned. ■ compute IORs for the whole Vector, when result is to leave the JVM where it was created. If this property is set to true, the EJB container does not do any of the above.	false

Property	Description	Default
ejb.collect.stats_gather_ frequency=<num>	The time interval in seconds between printouts of container statistics. If set to zero, this disables stats gathering and no stats are displayed (since they are not collected). This means that a zero setting overrides whatever may be the value of ejb.collect.display_statistics, ejb.collect.statistics or ejb.collect.display_detail_statistics properties.	5
ejb.collect.display_ statistics=true|false	This flag turns on timer diagnostics, which allow the user to see how the Container is using the CPU.	false
ejb.collect.statistics= true|false	Same as the ejb.collect.display_statistics property, except this property does not write the timer value to the log.	false
ejb.collect.display_detail_ statistics=true|false	This flag turns on the timer diagnostics, as ejb.collect.display_statistics option does. In addition, it prints out method level timing information. This allows the developer to see how different methods of the bean are using CPU. Please note, that the console output of this flag will require you to widen your terminal to avoid wrapping of long lines.	false
ejb.mdb.threadMaxIdle=<num>	There is a VM wide thread pool maintained by the EJB container for message-driven bean execution. This pool has the same configurability as the ORB dispatcher pool for handling RMI invocations. This particular property controls the maximum duration in seconds a thread can idle before being reaped out.	300
ejb.mdb.threadMax=<num>	Maximum number of threads allowed in the MDB thread pool.	0 (no limit)
ejb.mdb.threadMin=<num>	Minimum number of threads allowed in the MDB thread pool.	0
ejb.allowNullsInFinders= true|false	This property is applicable only to CMP 2.x. If you set this property to true, it will allow the presence of NULLs as the return value of a finder or as part of a finder Collection. By default, this property is set to False.	False

EJB Customization Properties: Deployment Descriptor level

These properties customize the behavior of a particular EJB. Some of them are applicable only to a particular type of EJB (such as session or entity), others are applicable to any kind of bean. There are several places where these properties can be set. Below are these places in the order of precedence:

1 Property element defined on the EJB level in the ejb-borland.xml deployment descriptor of a JAR file. This setting affects this particular EJB only. For example, the following XML sets the ejb.maxBeansInPool property to 99 for the EJB named data:

```
<ejb-jar>
   ⋮
   <enterprise-beans>
      <entity>
         <ejb-name>data</ejb-name>
         <bean-home-name>data</bean-home-name>
         <property>
            <prop-name>ejb.maxBeansInPool</prop-name>
            <prop-type>Integer</prop-type>
            <prop-value>99</prop-value>
         </property>
      </entity>
   </enterprise-beans>
   ⋮
</ejb-jar>
```

2 Property element defined on the <ejb-jar> level in the ejb-borland.xml deployment descriptor of a JAR file. This setting affects all EJBs defined in this JAR. For example, the following XML sets the `ejb.maxBeansInPool` property to 99 for all EJBs in the particular JAR file:

```
<ejb-jar>
   ⋮
   <property>
      <prop-name>ejb.maxBeansInPool</prop-name>
      <prop-type>Integer</prop-type>
      <prop-value>99</prop-value>
   </property>
   ⋮
</ejb-jar>
```

3 Property element defined at the `<application>` level in the `application-borland.xml` deployment descriptor of an EAR file. This setting affects all EJBs defined in the all JARs located in this EAR file. For example, the following XML sets the `ejb.maxBeansInPool` property to 99 on the EAR level:

```
<application>
   ⋮
   <property>
      <prop-name>ejb.maxBeansInPool</prop-name>
      <prop-type>Integer</prop-type>
      <prop-value>99</prop-value>
   </property>
   ⋮
</application>
```

4 EJB property defined as an EJB container level property. This affects all EJBs deployed in this EJB container. For example, the following command sets the `ejb.maxBeansInPool` property to 99 for all beans deployed in the EJB container started standalone:

```
vbj -Dejb.maxBeansInPool=99 com.inprsie.ejb.Container ejbcontainer hello.ear
-jns -jss -jts
```

Complete Index of EJB Properties

Properties common for any kind of EJB

Property	Type	Description	Default
`ejb.default_transaction_attribute`	`Enumeration (NotSupported, Supports, Required, RequiresNew, Mandatory, Never)`	This property specifies a transaction attribute value for the methods which have no trans-attribute defined in the standard deployment descriptor. Note, that if this property is not specified, the EJB container does not assume any default transaction attribute. Thus, specifying this property, may simplify porting J2EE applications created with other application servers which assume some default transaction attribute.	None

Entity Bean Properties (applicable to all types of entities—BMP, CMP 1.1 and CMP 2)

Property	Type	Description	Default
ejb.maxBeansInPool	Integer	This option specifies the maximum number of beans in the ready pool. If the ready pool exceeds this limit, entities will be removed from the container by calling unsetEntityContext.	1000
ejb.maxBeansInCache	Integer	This option specifies the maximum number of beans in the cache that holds on to beans associated with primary keys, but not transactions. This is relevant for Option “A” and “B” (see ejb.transactionCommitMode below). If the cache exceeds this limit, entities will be moved to the ready pool by calling ejbPassivate.	1000
ejb.maxBeansInTransactions	Integer	A transaction can access any/large number of entities. This property sets an upper limit on the number of physical bean instances that EJB container will create. Irrespective of the number of database entities/rows accessed, the container will manage to complete the transaction with a smaller number of entity objects (dispatchers). The default for this is calculated as ejb.maxBeansInCache/2. If the ejb.maxBeansInCache property is not set, this translates to 500.	Calculated

Property	Type	Description	Default
ejb.transactionCommitMode	Enumeration (A|Exclusive, B|Shared, C|None)	This flag indicates the disposition of an entity bean with respect to a transaction. The values are: A or Exclusive: This entity has exclusive access to the particular table in the DB. Thus, the state of the bean at the end of the last committed transaction can be assumed to be the state of the bean at the beginning of the next transaction. For example, to cache the beans across transactions. B or Shared: This entity shares access to the particular table in the DB. However, for performance reasons, a particular bean remains associated with a particular primary key between transactions, to avoid extraneous calls to ejbActivate and ejbPassivate between transactions. This means the bean stays in the active pool. This setting is the default. C or None: This entity shares access to the particular table in the DB. A particular bean does not remain associated with a particular primary key between transactions, but goes back to ready pool after every transaction. This is generally not a useful setting.	Shared
ejb.transactionManagerInstanceName	String	Use this property to specify by name a particular transaction manager for driving the transaction started for method calls. This option is useful in cases where you need 2PC completion of a particular transaction but want to avoid the RPC overhead of using a 2PC transaction manager for all other transactions in the system. This is also supported for MDBs.	None

Property	Type	Description	Default
ejb.findByPrimaryKeyBehavior	Enumeration (Verify, Load, None)	This flag indicates the desired behavior of the findByPrimaryKey method. The values are: Verify: This is the standard behavior, for findByPrimaryKey to simply verify that the specified primary key exists in the database. Load: This behavior causes the bean's state to be loaded into the container when findByPrimaryKey is invoked, if the finder call is running in an active transaction. The assumption is that found objects will typically be used, and it is optimal to go ahead and load the object's state at find time. This setting is the default. None: This behavior indicates that findByPrimaryKey should be a no-op. Basically, this causes the verification of the bean to be deferred until the object is actually used. Since it is always the case that an object could be removed between calling find and actually using the object, for most programs this optimization will not cause a change in client logic.	ejb.checkExistenceBeforeCreate

Property	Type	Description	Default
Boolean	Most tables to which entity beans are mapped have a Primary Key Constraint. If the CMP engine attempts to create a bean that already exists, this constraint is violated and a DuplicateKeyException is thrown. Some tables, however, do not define Primary Key Constraints. In these cases, the checkExistanceBeforeCreate property can be used to avoid duplicate entities. When set to True, the CMP engine checks the database to see if the entity exists before attempting the insert operation. If the entity exists then the DuplicateKeyException is thrown.	False	

Message Driven Bean Properties

Property	Type	Description	Default
ejb.mdb.use_jms_threads	Boolean	Option to switch to using the JMS providers dispatch thread rather than the Container managed thread to execute the onMessage() method. For OpenJMS this value will be true, since the message will be delivered in the JMS providers dispatch thread.	false **Note:** This value will be true by default for OpenJMS.
ejb.mdb.local_transaction_optimization	Boolean	This property is currently used only with OpenJMS. It is used to attain atomicity without using the XAConnectionFactory. The same database is used for message persistence and application data.	true
ejb.mdb.maxMessagesPerServerSession	Integer	For JMS providers that support the option to batch load a ServerSession with multiple messages, use this property to tune performance.	5
ejb.mdb.max-size	Integer	This is the maximum number of connections in the pool.	None
ejb.mdb.init-size	Integer	When the pool is initially created, this is the number of connections AppServer populates the pool with.	None

Property	Type	Description	Default
ejb.mdb.wait_timeout	Integer	The number of seconds to wait for a free connection when maxPoolSize connections are already opened. When using the maxPoolSize property and the pool is at its max and can't serve any more connections, the threads looking for JDBC connections end up waiting for the connection(s) to become available for a long time if the wait time is unbounded (set to 0 seconds). You can set the waitTimeout period to suit your needs.	30
ejb.mdb.rebindAttemptCount	Integer	This is the number of times the EJB Container attempts to re-establish a failed JMS connection or a connection that was never established for the MDB. To make the Container attempt to rebind infinitely you need to explicitly specify ejb.mdb.rebindAttemptCount=0.	5
ejb.mdb.rebindAttemptInterval	Integer	The time in seconds between successive retry attempts (see above property) for a failed JMS connection or a connection that was never established.	60
ejb.mdb.maxRedeliverAttemptCount	Integer	This is the number of times a message will be re-delivered by the JMS service provider should the MDB fail to consume a message for any reason. The message will only be re-delivered five times. After five attempts, the message will be delivered to a dead queue (if one is configured).	5
ejb.mdb.unDeliverableQueueConnectionFactory	String	Should an MDB fail to consume a message for any reason, the message will be re-delivered by the JMS service. The message will only be re-delivered five times. After five attempts, the message will be delivered to a dead queue (if one is configured). This property looks up the JNDI name for the connection factory to create a connection to the JMS service. This property is used in conjunction with ejb.mdb.unDeliverableQueue.	None
ejb.mdb.unDeliverableQueue	String	Should an MDB fail to consume a message for any reason, the message will be re-delivered by the JMS service. The message will only be re-delivered five times. After five attempts, the message will be delivered to a dead queue (if one is configured). This property looks up the JNDI name of the queue. This property is used in conjunction with ejb.mdb.unDeliverableQueueConnectionFactory.	None
ejb.transactionManagerInstanceName	String	This property is currently supported only for MDBs that have the “Required” transaction attribute. Use this property to specify by name a particular transaction manager for driving the transaction started for the onMessage() call. This option is useful in cases where you need 2PC completion of this particular transaction but desire to avoid the RPC overhead of using the 2PC transaction manager for all other transactions in the system, for example, in entity beans. Please refer to the MDB chapter for more details.	None

Stateful Session Bean Properties

Property	Type	Description	Default
ejb.sfsb.passivation_timeout	Integer	Defines the time interval (in seconds) when to passivate inactive stateful session beans into the persistent storage (JSS).	5
ejb.sfsb.instance_max	Integer	Defines the maximum number of instances of a particular stateful session bean allowed to exist in the EJB container memory at the same time. If this number is reached and a new instance of a stateful session needs to be allocated, the EJB container throws an exception indicating lack of resources. 0 is a special value. It means no maximum set. Note, that this property is applicable only if the ejb.sfsb.passivation_timeout property is set to non-zero value.	0
ejb.sfsb.instance_max_timeout	Integer	If the max number of stateful sessions defined by the ejb.sfsb.instance_max property is reached, the EJB container blocks a request for an allocation of a new bean for the time defined by this property waiting if the number goes lower before throwing an exception indicating lack of resources. This property is defined in ms (1/1000th of second). 0 is a special value. It means not to wait and throw an exception indicating lack of resources immediately.	0
ejb.jsec.doInstanceBasedAC	Boolean	If set to true, the EJB container checks if the principal invoking an EJB's method is the same principal that created this bean. If this check fails, the method throws a java.rmi.AccessException (or javax.ejb.AccessLocalException) exception. This is applicable to stateful session beans only.	True

EJB Security Properties

Property	Type	Description	Default
ejb.security.transportType	Enumeration (CLEAR_ONLY, SECURE_ONLY, ALL)	This property configures the Quality of Protection of a particular EJB. If set to CLEAR_ONLY, only non-secure connections are accepted from the client to this EJB. This is the default setting, if the EJB does not have any method permissions. If set to SECURE_ONLY, only secure connections are accepted form the client to this EJB. This is the default setting, if the EJB has at least one method permission set. If set to ALL, both secure and non-secure connections are accepted from the client. Setting this property controls a transport value of the ServerQoPConfig policy.	None
ejb.security.trustInClient	Boolean	This property configures the Quality of Protection of a particular EJB. If set to true, the EJB container requires the client to provide an authenticated identity. By default, the property is set to false, if there is at least one method with no method permissions set. Otherwise, it is set to true. Setting this property controls a transport value of the ServerQoPConfig policy.	

Java Session Service (JSS) Properties

JSS can run as part of standalone EJB container (-jss option) or as part of a Partition.

As a “Partition service", JSS Configuration information is located in each Partition's data directory in the partition.xml file. By default, this file is located in the following directory:

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>
/mos/<partition_name>/adm/properties/
```

For example, for a Partition named “standard", by default the JSS configuration information is located in:

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>
/mos/standard/adm/properties/partition.xml
```

For more information, go to the *partition.xml reference*, “<service> element” on page 344.

Otherwise, for the location of a Partition data directory, go to the configuration.xml file located in:

```
<install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/
```

and search for the Partition Managed Object directory attribute:

```
<partition-process directory=
```

The JSS supports two kinds of backend storage: JDataStore or a JDBC datasource. For more information, go to the Java Session Service (JSS) configuration section.

Property	Console Property Name	Description	Default
jss.workingDir=<path>	Working directory	The directory where the backend database (JDataStore) file is located. **Note:** this property is applicable only if the jss.pstore property is configured to use a JDataStore file as backend storage.	If not specified and the JSS runs in the Partition, then the Partition's working directory <install_dir>/var/domains/<domain_name>/configurations/<configuration_name>/mos/<partition_name> is used. If not specified and the JSS runs as part of a standalone EJB container, then the current directory where the container started is used.
jss.factoryName=<char_string>	Factory name	Name given to the JSS factory created by this service. The service gets registered with this name in the Smart Agent (osagent).	If not specified and the JSS runs in the Partition, the default value is: <server_name>:file:<install_dir>/ var/domains/<domain_name>/configurations/<configuration_name>/ mos/<partition_name>/ . If not specified and the JSS runs in a standalone EJB container, the default value is EJB/JSS[<container_name>].
jss.softCommit=true|false	Soft commit	If true, the JSS uses the JDataStore backend database with the Soft Commit mode enabled. Setting this property improves the performance of the Session Service, but can cause recently committed transactions to be rolled back after a system crash. **Note:** this property is applicable only if the jss.pstore property is configured to use a JDataStore file as backend storage. For more details, see the JDataStore documentation at: http://info.borland.com/techpubs/jdatastore/.	true
jss.maxIdle=<numeric value>	Max idle	The time interval in seconds between runs of JSS garbage collection job. The JSS garbage collection job is responsible for removing the state of expired sessions from the backend database. If set to 0, the garbage collection job never starts.	1800 (=30min)
jss.debug=true|false		Print debug information. If set to true, the JSS prints out debug traces.	false

Property	Console Property Name	Description	Default
jss.pstore=<char_string>	Persistent store	Specifies the JDatastore file to use for backend storage. If the file does not exist, JSS creates the file with the .jds extention, for example jss_factory.jds. For any compatible database supporting JDBC, specifies the JNDI name with the serial: prefix, for example serial://datasources/OracleDB to use for backend storage. In this case, JSS uses a datasource that is deployed in the Naming Service under the JNDI name specified.	If the JSS runs in the Partition, the JDataStore file named is used, such as jss_factory.jds. If the JSS runs in a standalone ejb container, the <container_name>_jss.jds is used.
jss.backingStoreType=<Dx|JDBC>		Specifies the type of persistent backend storage to use. Acceptable values are: Dx (indicating a local JDatastore database, or JDBC (indicating a JDBC datasource that is resolved from the JNDI name provided using the jss.pstore property value).	For a JSS that runs in a Partition, the default is Dx (local JDataStore database). If the JSS runs in a standalone ejb container, the default is Dx.
jss.userName=<char_string>	User name	User name JSS uses to open a connection with the JDataStore backend database. **Note:** this property is applicable only if the jss.pstore property is configured to use a JDataStore file as backend storage.	<default-user-name>
jss.passWord=<char_string>		When JSS persistent storage is defined through jss.backingStoretype=DX, use this property to specify the password value required for jss.userName access to the local JDataStore database. **Note:** This property is applicable only if jss.backingStoretype=Dx (configured to use JDataStore for persistent backend storage).	masterkey

Partition Transaction Service (Transaction Manager)

Listed below are properties that influence the behavior of the Partition Transaction Service (Transaction Manager). The properties can be specified when hosted by either a standalone EJB container or a Partition.

When configuring the Partition Transaction Service for a Partition, set the properties in the `partition.xml` file which is located in the `<install_dir>/var/domains/base/configurations/<configuration_name>/mos/<partition_name>/adm/properties`.

If running an EJB container standalone, they must be specified using system property names described below in section titled JTS System Properties. For example, when JTS is hosted by a standalone EJB Container property `jts.allow_unrecoverable_completion` must be specified using its system property equivalent:

```
prompt% vbj -DEJBAllowUnrecoverableCompletion com.inprise.ejb.Container
ejbcontainer beans.jar -jns -jts
```

Property	Description	Default
`jts.allow_unrecoverable_completion=true|false`	If set to `true`, this instructs the Container built-in JTS implementation to do a non-recoverable (that is, non two-phase) completion when there are multiple Resource registrations. Use at your own risk. It is provided only as a developer friendly feature. For OpenJMS, this property is set to true by default.	`False`
`jts.no_global_tids=true|false`	By default, JTS generates X/Open XA compatible transaction identifiers. By setting this property to `true`, the transaction key generation behavior changes to generate non-XA compliant tids. By generating XA compliant properties out of the box, the EJB container can work with JDBC2/XA drivers seamlessly.	`False`
`jts.no_local_tids=true|false`	There is an optimization where the EJB container detects that a transaction was started in the transaction service that lives in the same VM, and make the transaction comparison faster. Setting this property to `true` turns that off. This local transaction identifier (local tid) is a subset of the global transaction id hence makes the transaction comparisons faster.	`False`
`jts.timeout_enable=true|false`	By default, JTS transaction timeout facility is disabled. When enabled, each new transaction created by JTS will registered with a timeout in the JTS Timeout Manager. If the timeout expires before completion of the transaction, JTS will automatically rollback the transaction.	`False`
`jts.timeout_interval=<num>`	The JTS Timeout Manager examines registered transactions for timeout expiration at intervals in seconds controlled by the value of this property. Setting it to a value of `0` causes the interval to occur every 9999 seconds.	5

Property	Description	Default
jts.default_timeout=<num>	The timeout period for a Bean Managed transaction can be configured using JTA UserTransaction setTransactionTimeout() method. If not used or if transaction is a Container Managed transaction, the default transaction timeout value is applied. This can be configured upon JTS startup using the jts.default_timeout property value. The granularity of this property is 1 second.	600
jts.default_max_timeout=<num>	To prevent specification of an excessive timeout value for the jts.default_timeout property, the jts.default_max_timeout property controls the maximum time a transaction can be active before its expired. The granularity of this property is 1 second.	3600
jts.trace=true|false	Set this property to generate JTS debug messages.	False
jts.transaction_debug_timeout=<num>	If set, this property displays a list of active transactions maintained by JTS. Its value dictates the interval in seconds at which transactions are displayed.	None

Chapter

37

Using LifeRay Portal 3.6.0 with AppServer 6.6

This document describes the steps to prepare the liferay ear for deployment, create a liferay configuration using the Borland AppServer (AppServer) console, deploy the LifeRay portal, and deploy any custom portlets.

Liferay is an open-source portal that is designed to deploy portlets. It provides personalization, user/group management, web mail, message boards, content management all rolled into one package. It comes bundled with many portlet applications which are compliant with Java Portlet Specification, JSR-168.

In order to use the LifeRay Portal with AppServer do the following:

1. Download the LifeRay 3.6.0 EAR file from `http://www.liferay.com`.
2. Open a Borland Management Console and log in.
3. Create a configuration for the LifeRay Portal. When you create the configuration using the Management Console the configuration comes preconfigured with a LifeRay partition, JDataStore, and JMS.
   a. Right-click on Configurations node in the left pane and select Add Configuration... from the menu.
   b. Click on Portals in the Template Gallery.
   c. Select LifeRay Portal Configuration from the right pane in the Template Gallery and click on the Select button. The Create New Configuration dialog box will open.
   d. Enter a name for the new liferay configuration in the Name field.
   e. Change the Smart Agent Port in the Configuration Properties box by double-clicking on the value in the Value column.
   f. Click OK.
4. Run the configuration by right-clicking on the configuration name and selecting Start from the resulting menu.

5 Create a LifeRay server on which to host the LifeRay EAR file. Deploy the EAR file to the server.

   To create a LifeRay server:

   a Right-click on the Hosted Modules node under the LifeRay partition in the left pane of the Borland Management Console.

   b Select Host LifeRay module... from the menu. The Host LifeRay Portal dialog will open.

   c Enter the path to the LifeRay EAR file in the Liferay Ear Path box.

   d Enter the path to the directory where you want to host the LifeRay module in the Host Target Directory field. This directory must be on the same machine as the agent. Make sure that this directory already exists.

   e Make sure that the Generate Stub checkbox is checked.

   f Enter a name in the Module Name text box if you want to change the default module name. By default the module gets the same name as the directory in which you host the LifeRay module.

   g Click OK. You will see a status box. The AppServer will first generate the stub then extract the contents of the EAR file into the directory that you entered in step 3.

6 Test whether the LifeRay module has been deployed correctly by going to `http://localhost:8080` in a browser. This should open the LifeRay portal in the browser. The default login name for the LifeRay Portal is test@liferay.com and the default password is test.

Note To change any of the LifeRay partition properties, right-click on the LifeRay partition name in the left pane of the Management Console and select Properties from the menu. Click on the desired tab to bring it forward and change the properties associated with that tab.

Using Other Databases

By default, LifeRay uses the JDataStore database to store data. You can use a database other than JDataStore. Refer to the `http://www.liferay.com/web/guest/documentation/development/databases` site for information on which databases are supported. If you would like to use a database other than JDataStore, you must do the following:

1 Create a new `liferay.dar` file with the JNDI information for the database you want to use in the `jndi-definitions.xml` file.

   For information on how to edit the `jndi-definitions.xml` file for the database you want to use, see `http://www.liferay.com/web/guest/documentation/development/databases` site.

   For information on how to create a DAR file, see “Creating a JNDI definitions archive (DAR)” in the *Management Console User's Guide*.

2 Replace the default `liferay.dar` that is included with the LifeRay portal configuration with the new one you created.

Deploying Portlet or J2EE modules to LifeRay module

You can add an EJB JAR, WAR, RAR or a library JAR to a hosted liferay module. To deploy any of these to a LifeRay module:

1. Open the Borland Management Console.
2. Expand the LifeRay partition node in the left pane.
3. Right-click on the LifeRay hosted module under the Hosted Modules node, and select Deploy Portlet from the menu. The LifeRay Portal Deployment Wizard will open.
4. Click on the Add button to point the wizard to the portlet (WAR file) that you want to deploy.
5. Click on the Finish button.

To check whether the portlet is deployed successfully:

1. Open a web browser.
2. Go to the LifeRay portal by typing `http://localhost:8080` in a web browser
3. Log in to the portal using the default login which is test@liferay.com and password which is test.
4. Scroll down until you see the Add Portlet to Wide Column field. You should see the newly added portlet in the drop-down menu for field.
5. Select the portlet and click on the Add button to add the portlet to your portal.

Chapter

38

Integrating Borland AppServer 6.6 with JBuilder 2006

This chapter explains how to install the Borland AppServer 6.6 plug-in for JBuilder 2006, configure the plug-in, and use Borland AppServer 6.6 with JBuilder 2006.

Note JBuilder 2006 also supports Borland Enterprise Server AppServer Edition versions 6.0RP1 and 6.5 (Patch 11). For more information on these plug-in versions, see "Using JBuilder with Borland servers" in *Developing J2EE Applications* in the JBuilder online Help for more information.

Installing the Borland AppServer 6.6 plug-in

The Borland AppServer 6.6 plug-in for JBuilder 2006 is installed to the `<APPSERVER_HOME>/etc` folder of the Borland AppServer installation. The plug-in is not installed with JBuilder 2006. In order to access the Borland AppServer 6.6 plug-in and the J2EE 1.4 supported features, you need to copy the plug-in to the JBuilder `<JBUILDER_HOME>/patch` folder and restart JBuilder.

To install the JBuilder 2006 plug-in,

1. Save your project and exit JBuilder.
2. Copy `jbuilder2006_bas66_plugin.jar` from the `<APPSERVER_HOME>/etc/jbuilder` folder to the `<JBUILDER_HOME>/patch` folder.
3. Restart JBuilder.

Configuring JBuilder 2006 for Borland AppServer 6.6

After you have installed the plug-in and restarted JBuilder, you need to configure JBuilder to use the plug-in.

To configure JBuilder settings for Borland AppServer 6.6,

1. Choose Enterprise|Configure Servers to display the Configure Servers dialog box. The right side of the dialog box displays the default settings for the server. The General page displays common fields, while the Custom page displays server-specific fields. In some cases, modifying a Custom setting will update a setting on the General page.
2. Select Borland Enterprise Server AppServer Edition 6.x from the User Home folder in the left pane.

Note 6.x refers to AppServer versions Borland Enterprise Server AppServer Edition 6.0RP1, Borland Enterprise Server AppServer Edition 6.5 (Patch 11), and Borland AppServer 6.6.

3. Select the Enable Server option at the top of the dialog box.
   Checking this option enables the fields for Borland AppServer 6.6. You won't be able to edit any fields until it is checked. The Enable Server check box also determines whether this server will appear in the list of servers when you select a server for your project using Project|Project Properties|Servers.
4. View and change (if required) fields on the General tab:
   - Home Directory: The directory where Borland AppServer 6.6 is installed. The default is `Borland/AppServer`. If the default directory is not correct, use the ellipsis (…) button to browse to the correct directory.
   - Native Executable Launcher: The native executable used to run this server. The default is partition.exe in the `<APPSERVER_HOME>/bin` folder. If JBuilder is able to locate the native executable, this field is automatically filled in for you.
   - VM Parameters: The parameters you want to pass to the virtual machine.
   - Server Parameters: The parameters you want to pass to the server.
   - Working Directory: The location of a working directory.
5. Click the Custom tab to view and change (if required) fields unique to the server. Change or fill in these fields:
   - JDK Installation Directory: The directory where JDK v 1.5.0 is located. For Borland AppServer 6.6, this is automatically set to the `<APPSERVER_HOME>/jdk/jdk1.5.0` folder. Your project will use this JDK to run the AppServer partition.
   - Server Name: The Borland AppServer 6.6 hub name.
   - Configuration Name: The name of the configuration that manages the partition. By default, this is set to jbuilder.
   - Partition Name: The name of the partition in which the module is run. By default, this is set to jbpartition.
   - Add A Management Agent Item To The Enterprise Menu: Adds a Management Agent item to the JBuilder Enterprise menu, so you can start the Management Agent quickly from the JBuilder IDE.
   - Server Realm: The server realm name. For more information, see the *Borland Management Console User's Guide*. The default setting is ServerRealm.
   - User Name: The name you use to identify yourself to the server. The default value is admin.
   - User Password: The password you use to identify yourself to the server. The default value is admin.

- Advanced Settings: Click this button to display the Advanced Settings dialog box. Use this dialog box to change the port number used by the Management Agent and to select the Use Security option. The management port is used in JBuilder to detect the server during startup and deployment. Change the management port only if you are deploying to a remote server with a management port that is different from the default. When you change the port number, make sure that you entered the same port number as the server: the server won't start up without the correct port number. The port and security settings you choose must match the settings of your server. The values are read from the Borland AppServer property files, but the values will be changed automatically if you change the Home Directory setting. Note that changing the port number while you have the Management Agent started in JBuilder automatically shuts down the Management Agent.

6 Click OK to close the dialog box and save settings.
Any projects that were targeted for Borland Enterprise Server AppServer Edition 6.x are automatically updated to the new Borland AppServer home.

Displaying the Borland Management Console in JBuilder

After you have installed and configured JBuilder 2006 for Borland AppServer 6.6, you can display the Borland AppServer Management Console in the JBuilder message pane. To do so, you need to edit the `jbuilder.config` configuration file.

1 Save your project and exit JBuilder.

2 Open `jbuilder.config` in a text editor. This file is located in the `<JBUILDER_HOME>/bin` folder.

3 Add the following VM parameter to the configuration file:
`vmparam -Djava.endorsed.dirs=<APPSERVER_HOME>/lib/endorsed`

4 Restart JBuilder.

5 Choose View|Panes|BAS Console 6.6.
The Borland Management Console is displayed in the message pane.

VisiBroker development with JBuilder

You use the CORBA node of the Enterprise Setup dialog box (Enterprise|Enterprise Setup) to set up Borland AppServer for use with VisiBroker 7.0 in JBuilder 2006.

To make the ORB available to JBuilder,

1 Choose Enterprise|Enterprise Setup to display the Enterprise Setup dialog box. Select the CORBA page. The parameters in this dialog box allow a JBuilder user to develop CORBA applications.

2 Select the VisiBroker (Borland Enterprise Server AppServer Edition 6.x) option from the Configuration drop-down list. This option is automatically updated to point to the `<APPSERVER_HOME>/bin` folder.

3 To start the Smart Agent from the Tools menu, check the Add The VisiBroker Smart Agent Item To The Tools Menu option.

4 Enter the number of the SmartAgent port in the SmartAgent Port field.

5 Click OK to save the configuration.

Important In the runtime configuration for your CORBA application, you need to add the following parameters to the VM Parameters field in the Edit Runtime Configuration dialog box (Run|Configurations|Edit):

```
-Dvbroker.agent.port=<your_osagent_port>
-Dborland.enterprise.licenseDir=<APPSERVER_HOME>/var
-Dborland.enterprise.licenseDefaultDir=<APPSERVER_HOME>/license
```

You've now completed setting up your system to use VisiBroker 7.0, installed with Borland AppServer 6.6. Before running your application, choose Tools|VisiBroker Smart Agent to start the Smart Agent.

Using the JBuilder Deployment Descriptor Editor to develop J2EE 1.4 applications

Borland AppServer 6.6 supports J2EE 1.4 applications. The JBuilder Deployment Descriptor Editor provides new editors for J2EE 1.4 applications targeted for Borland AppServer 6.6.

J2EE 1.4 requires that each deployment descriptor be validated against an XML schema, instead of a DTD. The Borland AppServer 6.6 plug-in for JBuilder 2006 supports these updates. When you right-click a J2EE deployment descriptor in the project pane and choose Validate, the descriptor is validated against an XML schema file. The J2EE modules for Borland AppServer deployment are validated against the following schema files:

- Application module: `application_1_4-borland.xsd`
- Application Client module: `application-client_1_4-borland.xsd`
- Connector module: `connector_1_5.xsd`
- EJB module: `ejb-jar_2_1-borland.xsd`
- Web module: `web-app_2_4-borland.xsd`

To view the standard Java commented J2EE 1.4 XML schemas, go to Java 2 Platform, Enterprise Edition (J2EE) : XML Schemas for J2EE Deployment Descriptors at:

```
http://java.sun.com/xml/ns/j2ee/
```

To support Borland AppServer 6.6, the JBuilder Deployment Descriptor Editor now contains updated editors or new editors for the following entities:

- Message Destinations page: Web module, EJB module, Application Client module
- Message Destination Reference page: Web module, Application Client module, session bean, entity bean, message-driven bean
- Message-Driven Bean page: Message-driven bean
- Resource Environment References page: Message-driven bean
- Admin Object and Admin Object Properties page: Message-driven bean
- Resource Adapter page: Connector module
- BES Connection Definition page: Connector module

Note JBuilder provides a Deployment Descriptor editor for editing standard and server-specific J2EE 1.3 modules and deployment descriptors. For information on the editor, see “Editing EJB deployment descriptors” in *Developing Applications with Enterprise JavaBeans* in the JBuilder online Help.

Message Destinations page

The Messages Destinations page is a new DD Editor page for a Web module, EJB module, and Application Client module.

Use the Message Destinations page to set the new J2EE 1.4 deployment descriptor element `<message-destination-name>` for a Web module, EJB module, or Application Client module. This element specifies the name of a message destination reference. The Borland AppServer 6.6 value is a JNDI name that represents the message destination reference name used in the web module, EJB module, or application client module.

To display the Message Destinations page and set the standard and Borland AppServer-specific deployment descriptor elements,

1. Select the Web module, EJB module, or Application Client module in the project pane.
   Select the DD Editor tab at the bottom of the content pane.
2. Open the structure pane.
3. Expand the module node and select the Message Destinations node.
4. To add a message destination, right-click the node and choose Add.
5. Click the Standard tab in the DD Editor.
   a. Enter the name of the message destination in the Name field. The name must be unique among the names of message destinations within the deployment file.
   b. Enter the language with which to associate the Display Name, Description, and icons in the Language field. You can have one Display Name, Description, and large and small icon for each language. Use the Add and Remove buttons to add and remove languages.
   c. Enter the name to be displayed the Display Name field.
   d. Enter the description in the Description field.
   e. Enter the location of a large icon (32 x 32 pixels) in the Large Icon field. The icon must be contained in the module tree.
   f. Enter the location of a small icon (16 x 16 pixels) in the Small Icon field. The icon must be contained in the module tree.
6. Click the BES tab in the DD Editor. In the JNDI Name field, enter the JNDI name for the message destination. See Chapter 22, “Using JMS” for more information.

Message Destination Reference page

The Message Destinations Reference page is a new DD Editor page for a Web module and Application Client module, as well as for entity beans, session beans, and message-driven beans.

Use the Message Destination Reference page to set the new J2EE 1.4 deployment descriptor element `<message-destination-ref>` for a Web module and Application Client module, as well as for entity beans, session beans, and message-driven beans. The message destination reference contains a declaration of the reference associated with a resource.

To display the Message Destination Reference page and set the Borland AppServer 6.6-specific deployment descriptor element,

1. Select the Web module, EJB module, or Application Client module in the project pane.

   Select the DD Editor tab at the bottom of the content pane.
2. Open the structure pane.
3. Expand the module or bean node and select the Message Destination References node.
4. To add a message destination reference, right-click the node and choose Add.
5. Click the Standard tab in the DD Editor. Set the following attributes:
   a. Enter the name of the message destination reference in the Name field. This is the name used in deployment component code.
   b. Specify the Java type of the message destination in the Type field. The type specifies the Java interface to be implemented by the destination.
   c. Choose how the message destination will be used in the Usage field. Choose Consumes if messages are consumed from the message destination; choose Produces if messages are produced for the destination, or choose ConsumeProduces if messages are both consumed and produced. The Assembler makes use of this information in linking producers of a destination with its consumers.
   d. Specify the message destination link in the Link field. This element links a message destination reference or message-driven bean to a message destination. The Assembler sets the value to reflect the flow of messages between producers and consumers in the application. The value must be the name of a message destination in the same deployment file or in another deployment file in the same J2EE application unit. Alternatively, the value may be composed of a path name specifying a deployment file containing the referenced message destination with the name of the destination appended and separated from the path name by `#`. The path name is relative to the deployment file containing a deployment component that is referencing the message destination. This allows multiple message destinations with the same name to be uniquely identified.
   e. Enter the language to which the description applies in the Description Language field. Use the Add and Remove buttons to add and remove languages. You can have one description per language.
   f. Enter the description of the message destination reference in the Description field.
6. Click the BES tab in the DD Editor.
7. Enter the JNDI name for the message destination. See Chapter 22, “Using JMS” for more information.

Message-Driven Bean page

The Borland-specific Messages-Driven Bean page was updated for message-driven beans.

Use the Message-Driven Bean page of the DD Editor to set the deployment descriptor for a message-driven bean. Both the standard and Borland AppServer 6.6-specific pages have been updated for the J2EE 1.4 implementation.

To display the Message-Driven Bean page and set standard and BAS-specific deployment descriptor elements,

1. Select the EJB module in the project pane.
   The DD Editor is displayed in the content pane.
2. Open the structure pane.
3. Expand the EJB module, then the Message Driven Beans node. Select a message-driven bean.
   The Message-Driven Bean page is displayed in the DD Editor.
4. Click the Standard tab in the DD Editor. Set the following attributes:
   a. Enter the name of the message-driven bean in the Name field.
   b. Enter the fully-qualified name of the Java class that implements the bean's business methods in the EJB Class field. This information must be specified.
   c. Choose the bean's messaging type in the Messaging Type field.
   d. Select how the bean's transactions are managed in the Transaction Type drop-down list. Transactions can be managed by the bean itself or by the container.
   e. Select the message destination type in the Message Destination Type drop-down list. This is the actual topic or queue the message-driven bean is listening to.
   f. Enter the bean's message destination in the Message Destination Link field.
   g. Enter the language for the activation configuration description in the Description Language field. Use the Add and Remove buttons to add and remove languages. You can have one description per language.
   h. Enter a description of the bean in the Description field.
   i. Enter the language for the display information in the Language field. Use the Add and Remove buttons to add and remove languages. You can have one display description per language.
   j. Enter the name you want used to identify the bean for display purposes in the Display Name field.
   k. Enter a description for display purposes in the Description field.
   l. Enter the name of a large icon (32 x 32 pixels) you want associated with the bean in the Large Icon field.
   m. Enter the name of a small icon (16 x 16 pixels) you want associated with the bean in the Small Icon field.
5. Click the BES tab in the DD Editor. Use this page to set the Borland AppServer `<message-source>` element.
   a. Choose the message source type from the Message Source Type drop-down list. Choose jms-provider-ref to activate the message source through a JMS provider, using EJB 2.0 implementation. Choose adapter-ref to activate the message source through a JCA 1.5 resource adapter.
   b. Enter the message-driven bean destination in the Destination Name field. This is the actual topic or queue the message-driven bean is listening to. This field is available if jms_provider_ref is selected as the Message Source Type.

c Enter the resource connection factory used to connect to the JMS broker in the Connection Factory Name field. This field is available if jms_provider_ref is selected as the Message Source Type.

d Enter the initial number of connections in the Initial Pool Size field. This field is available if jms_provider_ref is selected as the Message Source Type.

e Enter the maximum number of connections in the Maximum Pool Size field. This field is available if jms_provider_ref is selected as the Message Source Type.

f Enter the length of time (in seconds) to wait for a connection in the Wait Timeout field. This field is available if jms_provider_ref is selected as the Message Source Type.

g Enter the name of the resource adapter instance that connects to a J2EE resource in the Instance Name field. This field is available if resource_adapter_ref is selected as the Message Source Type.

Resource Environment References page

The Borland-specific Resource Environment References page was updated for message-driven beans.

Use the Resource Environment References page to set the `<resource-environment-ref>` element for a message-driven bean. The resource environment reference can be set to either a JNDI name or an administered object. A resource environment reference maps a logical name used by the client application to the physical name of an object.

To display the Resource Environment Reference page and set the Borland AppServer 6.6-specific deployment descriptor element,

1 Select the EJB module in the project pane.
The DD Editor is displayed in the content pane.

2 Open the structure pane.

3 Expand the EJB module and select the Message Driven Beans node. Select a message-driven bean.

4 Right-click the Resource Environment References page and choose Add.

5 Click the BES tab in the DD Editor.

a Choose the type of reference from the Resource Environment References Type drop-down list. Choose JNDI name to select a JNDI reference. Choose Admin Object to select an administered object. If you select Admin Object, you need to set properties for the object. See “Admin Object and Admin Object Properties page” on page 375 for more information.

b Enter the name of the JNDI bean that maps the logical name to the object name in the JNDI Name field. This field is only available if JNDI Name is selected as the Resource Environment Type. See Chapter 22, “Using JMS” for more information.

Admin Object and Admin Object Properties page

The Admin Object and Admin Object Properties pages are new for resource environment references on message-driven beans.

Use the Admin Object page to add an administered object for a resource environment reference. Use the Admin Object Properties page to set the object properties. This page is only available if you select Admin Object as the Resource Environment References Type. Administered objects are specific to a messaging style or message provider.

To display the Admin Object page and set the Borland AppServer 6.6-specific deployment descriptor element,

1. Select the EJB module in the project pane.
   The DD Editor is displayed in the content pane.
2. Open the structure pane.
3. Expand the EJB module and select the Message Driven Beans node. Select a message-driven bean.
4. Right-click the Resource Environment References page and choose Add.
5. Click the BES tab in the DD Editor.
6. Choose Admin Object from the Resource Environment Reference Type drop-down list.
7. Expand the Resource Environment References node in the structure pane until you see the entry you just added.
8. Expand the node and select the Admin Object Properties node. Right-click the node and choose Add.
   The BES Admin Object Properties page is displayed in the DD Editor.
9. Enter properties:
   a. Enter the property name in the Name field.
   b. Choose the type of property from the Type drop-down list. Select one of java.lang.String, java.lang.Boolean, or Integer. You can also select <Unspecified>.
   c. Enter a value for the property in the Value field. The value must match the type of property.

Resource Adapter page

The Borland-specific Resource Adapter page is new for a Connector module.

Use the Resource Adapter page to set the Borland-specific JCA 1.5 deployment descriptor `<resourceadapter>` element for a Connector module. This element describes a resource adapter for a connector.

To display the Resource Adapter page and set the Borland AppServer 6.6-specific deployment descriptor element,

1. Select the Connector module in the project pane.
   The DD Editor is displayed in the content pane.
2. Open the structure pane.
3. Expand the Connector module and choose the Resource Adapter node.

4 Click the BES tab in the DD Editor.

   a Enter the name of the connection factory in the Instance Name field.

   b Enter the resource adapter link reference in the Resource Adapter Link Reference field. This allows you to associate multiple deployed resource adapters with a single deployed resource adapter. The link provides for linking and reusing resources already configured in a base resource adapter to another resource adapter, modifying only a subset of attributes. Using this field avoids duplication of resources where possible. Any values defined in the base resource adapter deployment are inherited by the linked resource adapter unless otherwise specified.

   c Enter the directory where all shared libraries should be copied in the Resource Adapter Library Directory field.

   d Enter the authorization domain for the connection in the Authorization Domain field.

BES Connection Definition page

The BES Connection Definition is new for a Connector module.

Use the BES Connection Definition page to set the Borland-specific JCA 1.5 deployment descriptor `<outbound-resourceadapter>` element for a Borland Connector Module. The information includes the fully qualified names of classes and interfaces that are required as part of the connector architecture, the number of managed connections, and the connection timing intervals.

To display the BES Connection Definition page and set the Borland AppServer 6.6-specific deployment descriptor element,

1 Select the Connector module in the project pane.
The DD Editor is displayed in the content pane.

2 Open the structure pane.

3 Expand the Connector module and the Resource Adapter node.

4 Right-click the BES Connection Definitions node and choose Add.

5 Set attributes for the connection definition:

   a Enter the name of the factory interface in the Factory Interface field.

   b Enter the name of the factory class used to connect to the JMS broker in the Factory Name field.

   c Enter the connection description in the Description field.

   d Enter the name of the JNDI class to the connection factory in the JNDI Name field. See Chapter 22, “Using JMS” for more information.

   e Check the Enable Logging option to require logging for the ManagedConnectionFactory or ManagedConnection classes.

   f Enter the name and location of the file where the logging results are to be written in the Log File Name field.

   g Enter the initial number of managed connections the server attempts to allocate at deployment time in the Initial Capacity field.

   h Enter the maximum number of managed connections the server allows to be allocated at any one time in the Maximum Capacity field.

   i Enter the length of time in seconds to wait if the connection is busy in the Busy Timeout field.

j Enter the length of time in seconds to wait before timing out a connection in the Idle Timeout field.

k Enter the length of time in seconds to wait for a connection in the Wait Timeout field.

l Enter the number of managed connections the server attempts to allocate when fulfilling a request for a new connection in the Capacity Delta field.

m Select the Enable Cleanup option to force the server to attempt to claim unused managed connections to save system resources.

n Enter the time in seconds the server waits between attempts to claim used managed connections in the Cleanup Interval field.

6 To add connection definition properties, expand the node for the definition you just added, right-click the Properties node and choose Add. Enter properties:

a Enter the property name in the Name field.

b Choose the type of property from the Type drop-down list. Select one of java.lang.String, java.lang.Boolean, or Integer. You can also select <Unspecified>.

c Enter a value for the property in the Value field. The value must match the type of property.

Creating a run configuration for Borland AppServer 6.6 targeted projects

JBuilder uses a default Borland AppServer configuration called jbuilder and a default partition called jbpartition as a deployment target for Borland AppServer 6.6. If the configuration or partition doesn't exist, one will be created automatically when you configure the plug-in. The server name is the same as the hub name.

You can start multiple partitions using multiple JBuilder run configurations. To create multiple run configurations, follow these steps:

1 Choose Run|Configurations, then click New.

2 Change the Run Type to Server. The displayed server is the server selected for the project in Project Properties|Server.

3 In the Category list, select Server|Command Line.

4 Change the Partition and Configuration fields to the ones you want to use.

5 Click OK to close the New Runtime Configuration dialog box and save the configuration.

6 Repeat these steps to create additional run configurations to run other partitions.

7 To run multiple partitions, use the Management Agent (Enterprise|Borland Enterprise Server Management Agent).

To avoid naming service conflicts, make sure that you have the naming service enabled for only one of the partitions. To disable the naming service for a partition, edit the run configuration for the partition (Edit Runtime Configuration dialog box) and uncheck the Naming/Directory service in the Category list.

Before starting up the partition, make sure you have configured unique port numbers for the Tomcat and JDataStore services.

Important You cannot change the Tomcat port setting from the JBuilder run configuration. You need to start the server, open the Borland AppServer Management Console, and set the port on the server side. To do this,

1. Start the Borland Management Agent from the command line: `<APPSERVER_HOME>/bin/scu.exe`
2. Open the Borland AppServer Management Console: `<APPSERVER_HOME>/bin/console.exe`
3. Log into the console.
4. Choose the Management Hubs node.
5. Expand the Management Hubs node until you see the running partition. Expand the partition node.
6. Right-click the Web Container node and choose Properties.
   The Configure Web Container dialog box is displayed.
7. Expand the Service: HTTP node. Choose Connector.
8. Scroll through the Connector page until you see the Port Number field.
9. Change the port number to the one you want to use.
10. Choose File|Save.

Changing the management port

The Management Agent manages all partitions. The default management port, set on the Advanced Settings dialog box (Tools|Configure Servers|Advanced Settings) is 42424. You can change the management port used by JBuilder, or the one used by the server.

To change the port used by JBuilder,

1. Choose Enterprise|Configure Servers and chose Borland Enterprise Server AppServer Edition 6.x from the User Home Folder on the left.
2. Click the Custom tab, then the Advanced Settings button.
3. Change the port in the Management Port field. (The default is 42424.)
4. Click OK two times.

To change the management port used by the server,

1. Open the Borland Management Console embedded in JBuilder 2006 (View|Panes|BAS 6.6 Console).

Note The embedded console has to be enabled first. See "Displaying the Borland Management Console in JBuilder" on page 369.

2. Double-click Installations.
3. Expand the server location node and select the server node.

Note The server is also the machine ID.

4. Expand the server node until you see the server displayed as a sub-node of the Agents node.
5. Right-click the server node and choose Properties.
6. Change the port number in the Management Port field.
7. Click OK to save the settings.

If you change the management port, the management agent will shutdown if it was started in JBuilder.

Launching the partition in JBuilder 2006

When you launch the partition in JBuilder, the Management Agent is started by default and the server is launched. After the partition has been started, all deployable archives are automatically deployed. Startup output is displayed in the message pane.

If you want to start multiple partitions, or quickly redeploy partitions, you can start the Management Agent (Enterprise|Borland Enterprise Server Management Agent). The Management Agent refers to scu.

To launch the partition and configuration for the server, right-click the module in the project pane you want to run. Select Run Using <Configuration_Name>. Typically, this will be the name of the server runtime configuration.

Running the partition in JBuilder will:

- Create the partition and configuration, if not already present. The partition and configuration name are derived from the run configuration used to start up the server. If you are starting the configuration or server using the default configuration, the partition name as configured in the application server properties will be used.
- Deploy any resources defined in `jndi-definitions.xml` (if present) to the root of the partition directory, `<APPSERVER_HOME>\var\domains\base\configurations\<CONFIGURATION_NAME>\mos\<PARTITION_NAME>\dars\jbuilder.dar`. The `jndi-definitions.xml` file is packaged into this `.dar` file and deployed.

Note The `jndi-definitions.xml` file is created if your project contains an EJB 2.0 module with data sources/messaging resources defined. This action can be turned off by unchecking the Deploy jndi-definitions.xml option on the Deployment|EJBs Service Properties page on the Server node of the Project Properties dialog box (Project Properties|Server|Services|Deployment|EJBs).

- Remove any archives deployed to the partition, if the Remove Archives Already Deployed To Server option is selected. You can set this option in the Server|Archives category on the Edit Runtime Configuration dialog box for the server run configuration (Run|Configurations|<Server_Config_Name>|Edit|Run page|Category list).
- Deploy the selected archives. By default, all deployable archives in the project are selected. You can choose the archives to deploy in the Server|Archives category on the Edit Runtime Configuration dialog box for the server run configuration (Run|Configurations|<Server_Config_Name>|Edit|Run page|Category list).
- Start the partition. When partition startup is complete, you should see the partitions listed in the message pane for Borland AppServer 6.6. Archives deployed at startup should be loaded and accessible.

Note By default, all services associated with a partition are started.

Deploying

To deploy EJBs, WARs, and EAR modules to Borland AppServer 6.6, follow these steps.

1. Choose Enterprise|Configure Servers.
2. Select Borland Enterprise Server AppServer Edition 6.x from the left side of the dialog box.
3. Click the Custom tab and set the Server, Configuration, and Partition names to match that of the server. (The server can be running remotely or on the local machine.)
4. Click the Advanced Settings button and make sure the Management Port setting matches the port set for the server.
5. Click OK two times.

Now you are ready to deploy. There are two ways to deploy EJBs, WARs, and EAR modules using JBuilder. You can deploy using the Deployment wizard or the context menu.

To deploy using the Deployment wizard,

1. Build your project (Project|Make).
2. Start the Management Agent (Enterprise|Borland Enterprise Server Management Agent).
3. Open the Server Deployment wizard (Enterprise|Server Deployment).
4. On Page 1 of the wizard, select the modules to deploy. Set the Restart Partitions On Deploy (Cold Deploy) option if the partition has already been started. Click Next.
5. On Page 2, select the partition you want to deploy modules to from the list.

Important If the selected partition has already been started, restart it to access the deployed modules.

6. Click Finish to deploy.

To deploy from the context menu,

1. Build your project (Project|Make).
2. Start the Management Agent (Enterprise|Borland Enterprise Server Management Agent).
3. In the project pane, right-click any deployable node.
4. Choose Deploy Options|Deploy.

Note To deploy more than one module, you can select multiple deployable nodes in the project pane, right-click them, and use the Deploy Options context menu.

Remote debugging

Before you can debug your application remotely, you need to configure the partition. Choose one of these topics for more information:

- Preparing to remote debug partitions that are not managed in JBuilder
- Preparing to remote debug partitions with JBuilder

Once the configuration, partition, and server are started, follow the instructions in "Remote debugging from JBuilder" on page 382. For a remote debugging tutorial, see "Tutorial: Remote debugging with the Borland Enterprise Server AppServer Edition 6.0" in *Developing Enterprise JavaBeans* in the JBuilder online help. The steps are the same for both the 6.x and 6.6 versions of the Borland application server.

Preparing to remote debug partitions that are not managed in JBuilder

To prepare to remote debug partitions that are not managed in JBuilder,

1. Start the Borland Management Agent from the command line: `<APPSERVER_HOME>/bin/scu.exe`
2. Open the Borland AppServer Management Console: `<APPSERVER_HOME>/bin/console.exe`
   Log into the console.
3. Choose the Management Hubs node. Expand the node until you see the partition you want to debug.
4. Right-click the partition name and choose Properties.
   The Partition Properties dialog box is displayed.
5. Select the Partition Process Settings tab.
6. Check the Enable JPDA Remote Debugging option.
7. Set the JPDA Debugging Transport Address to 3999.
8. Uncheck the Suspend Partition Until Debugger Attaches option.
9. Click OK.

Preparing to remote debug partitions with JBuilder

To prepare to remote debug partitions with JBuilder,

1. Shut down the server.
2. Open the file: `<APPSERVER_HOME>/var/domains/base/configurations/<CONFIGURATION_NAME>/configuration.xml`
3. Look for the JPDA element and edit attribute values as follows:

```
 enable-jpda-debug="true"
jpda-transport-address="3999"
jpda-suspend="false"
```

Remote debugging from JBuilder

Once the server, partition, and Management Agent have been started, follow these steps from the JBuilder IDE:

1. In the project from which you want to launch the remote debug session, choose Run|Configurations.
2. Select the Server run configuration and choose Edit.
3. Select the Debug|Connection node.
4. Select the Remote Attach option.
5. Set the Transport Type to dt_socket and the localhost value to 3999.
6. Click OK two times to close the Run Configuration dialog boxes.
7. Set a breakpoint in the process you want to debug.
8. Click the down arrow next to the Debug Project button on the toolbar and select the Server configuration you just created or edited. The debugger launches, attaches to the partition running remotely, and stops at the breakpoint.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X