Developer's Guide

Enterprise Server 6

Borland®
Excellence Endures™

Borland Software Corporation
100 Enterprise Way
Scotts Valley, California 95066-3249
www.borland.com

BES0060WW21002

0102030405-9 8 7 6 5 4 3 2

PDF

Contents

Tables

Figures

Chapter 1

Introduction to Borland Enterprise Server

The Borland Enterprise Server is a set of services and tools that enable you to build, deploy, and manage enterprise applications in your corporate environment. These applications provide dynamic content by using JSP, servlets, and Enterprise Java Bean (EJB) technologies.

BES Product overview

Borland provides the following different flavors of its Enterprise Server in order to better meet your specific deployment requirements:

- Web Edition: For those who do not require a full J2EE compliant application server, Borland provides the *Web Edition* which is designed for developing and deploying web applications using JavaServer pages and Servlets with a Java-based database.
- VisiBroker Edition: For the CORBA developer, Borland provides the *VisiBroker Edition*, which includes both *VisiBroker for Java* and *VisiBroker for C++* to leverage the industry-leading VisiBroker Object Request Broker (ORB). Both are complete implementations of the CORBA 2.6 specification. *VisiBroker Edition* also includes the *Web Edition* and it's feature set.
- VisiBroker Standalone (installation option): An installation option for those who purchase *VisiBroker Edition*, but prefer a smaller footprint - the standalone VisiBroker is comprised of only *VisiBroker for Java* and *VisiBroker for C++*.

- Team Edition: Provides a full J2EE 1.3 implementation and can service up to 25 concurrent users on a single box. The aim of the *Team Edition* is to support scalability within an architecture that delivers performance and reliability at an affordable price. In short, the *Team Edition* is the optimal solution for small to medium deployment scales that require full J2EE capabilities.
- Borland Enterprise Server: The complete *Borland Enterprise Server* product provides full J2EE support. On top of the *Web Edition*, *Team Edition*, and *VisiBroker Edition* features, BES supports unlimited concurrent users and Partitions (applications), adds enterprise-level clustering and other high-end features.

Each BES offering is built upon the same server core, and interact with each other seamlessly. You can choose the degree of functionality and services you need, and if your needs change, it is simple to upgrade your license. See the *BES Installation Guide* for details.

Web Edition

The Web Edition is designed for developing and deploying web applications using JavaServer Pages and Servlets with a Java-based database. The Web Edition includes:

- the open-source Apache Web Server version 2.0.
- the Borland web container based on the open-source Tomcat web container version 4.1.
- the Smart Agent for object referencing and directory service for server connection.
- Java Session Service (JSS) to store session information for recovery in case of container failure.
- Naming Service to associate one or more logical names with an object reference as well as hosting associations between serial names of data sources and the JNDI names.
- IIOP Connector that enables Apache to communicate with the Tomcat-based, Borland web container and CORBA servers via Internet Inter-ORB Protocol (IIOP). The IIOP Plug-in leverages the power of VisiBroker to allow for CORBA connectivity directly from Apache.
- IIOP redirector that enables Microsoft IIS to communicate with the Borland web container and CORBA servers.
- Borland's all Java relational database, JDataStore and support for JDBC datasources.

Web Edition features

The Web Edition offers the following features:

- a complete deployment platform for web applications.

- Web Services support, including Apache SOAP server integration, Borland XML toolkit, and development tools like the Deployment Descriptor Editor.
- industry-proven load balancing and fault tolerance.
- automatic session management.
- web-enabled CORBA servers.
- a homogeneous integration to an all-Java database with support for multiple connections.

VisiBroker Edition

VisiBroker is primarily for deployments that require CORBA to communicate with non-Java objects, and is comprised of both the VisiBroker for Java and the VisiBroker for C++ ORBs. Using VisiBroker's IIOP Connector, you can quickly move CORBA applications on-line with very little coding.

Visibroker includes:

- the VisiBroker ORB, the industry-leading Object Request Broker.
- the VisiNaming Service, a complete implementation of the Interoperable Naming Specification in the CORBA 2.6 specification from the OMG.
- the IIOP Plug-in for CORBA which routes Apache requests to Interface Definition Language (IDL) and CORBA via IIOP, and then routes them back as HTTP requests. Thus, Java and C++ CORBA objects can be used to service HTTP requests.

VisiBroker Edition features

VisiBroker features:

- "out-of-the-box" security and web connectivity.
- CORBA 2.6 compliance.
- a seamless migration to the J2EE Platform.
- CORBA support like naming and event services.

VisiBroker Standalone (installation option)

VisiBroker Standalone is a smaller footprint option for VisiBroker Edition, it is comprised only of *VisiBroker for Java* and *VisiBroker for C++*.

Team Edition

The *Team Edition* is a scaled down version of the full Borland Enterprise Server. This option is optimal for small to medium-sized deployments. Using the *Team Edition*, only one server can be installed per physical machine. Clustering is not supported.

Team Edition features

- provides a full Web Container.
- provides a full EJB Container.
- includes the Borland JMS services, the bundled database, JDataStore
- Fully J2EE 1.3 compliant.
- services up to 25 concurrent users on a single box.
- services 1 running Partition at a time.

Borland Enterprise Server “AppServer Edition”

The Borland Enterprise Server “AppServer Edition” allows you to deploy and manage your distributed Java and CORBA applications that implement the J2EE 1.3 platform standard. The AppServer Edition includes all the features and services of the Web Edition and Visibroker with the addition of:

- provides a full EJB Container.
- a Java Messaging Service (JMS).
- VisiConnect, Borland's J2EE Connector Architecture (Connectors) implementation for connecting to general Enterprise Information Systems (EIS).
- Security for the Borland Enterprise Server.
- support for the HP-UX and IBM AIX platforms.

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

Borland Enterprise Server “AppServer Edition” features

The AppServer Edition offers the following features:

- a complete implementation of J2EE 1.3 and EJB 2.0 standards.
- leading Java Messaging Solutions.
- "out-of-the-box" EIS integration through Connectors.
- integration with the Borland JBuilder integrated development environment.
- seamless integration with the VisiBroker ORB infrastructure.
- fully supported clustering.

Borland Enterprise Server (BES) Documentation

The Borland Enterprise Server documentation set includes the following:

- *BES Installation Guide* - describes how to install BES on your network. It is written for system administrators who are familiar with Windows or UNIX operating systems.

- *BES 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.
- BES *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, the Quality of Service, and the Interface Repository.
- BES *VisiBroker for C++ Developer's Guide* - describes how to develop VisiBroker applications in C++. 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, the Quality of Service, and the Interface Repository.
- BES *VisiBroker for C++ API Reference* - provides a description of the classes and interfaces supplied with VisiBroker for C++.
- BES *VisiBroker VisiNotify Guide* - describes Borland's implementation of the OMG standard, Notification Service, how to use the major features of the notification messaging framework, in particular, the Quality of Service (QoS) properties, Filtering, and Publish/Subscribe Adapter (PSA).
- BES *VisiBroker VisiTransact Guide* - describes Borland's implementation of the OMG standard, Transaction Service, discusses the components of the CORBA Transaction Service and transaction processing in a distributed environment.
- BES *VisiBroker GateKeeper Guide* - describes how to use the VisiBroker GateKeeper to enable VisiBroker clients to communicate with servers across networks, while still conforming to the security restrictions imposed by web browsers and firewalls.

Important The documentation in PDF format and updates to the product documentation are available on the web at http://www.borland.com/techpubs/bes.

Accessing the BES Standalone online Help Topics

To access the standalone online Help Topics 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 and type the following command:

```
help
```

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

```
help
```

Accessing online Help Topics from within BES

To access the online Help Topics when running the Management Console, use one of the following methods:

- From within the Borland Management Console, choose | Help
- From within the DDEditor, choose | Help
- From within the VisiBroker Console, choose | Help

Documentation conventions

The documentation for the Borland Enterprise Server 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 Borland Enterprise Server documentation uses the following symbols to indicate platform-specific information:

Windows: All supported Windows platforms.

Win2003: Windows 2003 only

WinXP: Windows XP only

Win2000: Windows 2000 only

UNIX: UNIX platforms

Solaris: Solaris only

Linux: Linux 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, please see our web site at: http://www.borland.com/devsupport and select your geographic region.

For Borland support worldwide information, visit: http://www.borland.com/devsupport/contacts.

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)
Listserv	To subscribe to electronic newsletters, use the online form at: http://www.borland.com/contact/listserv.html or, for Borland's international listserver, http://www.borland.com/contact/inlist.html

World Wide Web

Check http://www.borland.com regularly. The Borland Enterprise Server 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/products/downloads (updated software and other files)
- http://www.borland.com/techpubs/bes (documentation updates, html and PDFs)
- http://community.borland.com (contains our web-based news magazine for developers)

Borland newsgroups

You can participate in many threaded discussion groups devoted to the Borland Enterprise Server products.

You can find user-supported newsgroups for Enterprise Server and other Borland products at http://borland.com/newsgroups.

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

Chapter

2

Borland Enterprise Server overview and architecture

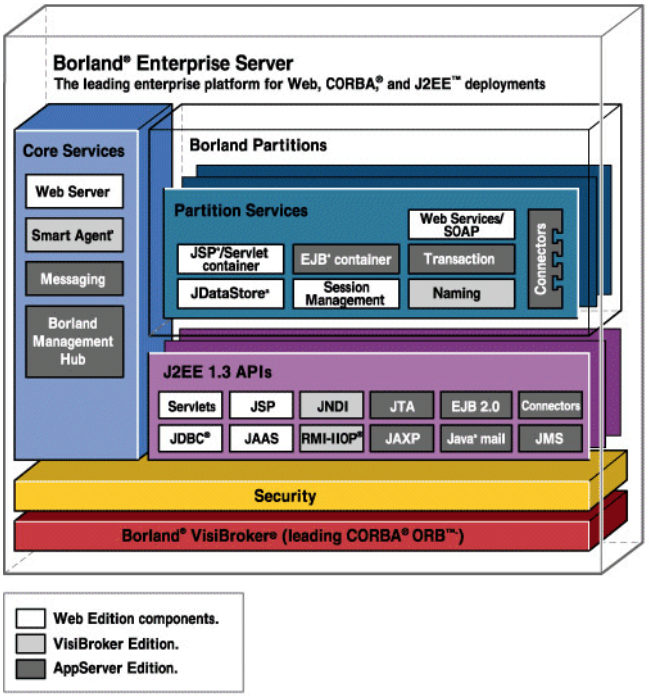
This section contains an overview of the Borland Enterprise Server products, Editions, and architecture.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

BES architecture overview

The Borland Enterprise Server is a CORBA-based, J2EE server that utilizes distributed objects throughout its architecture. With the Borland Enterprise Server, you can establish connectivity to platforms from corporate mainframes to simpler systems with small-business applications and remote databases. The Borland Enterprise Server 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 Borland Enterprise Server. An application server installation contains BES *core services* and *Partitions*.



BES services overview

BES services are those services available to all applications being hosted on the Borland Enterprise Server. They are:

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

Web Server

Borland Enterprise Server 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 ailing.

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

Borland Enterprise Server provides support for standard JMS pluggability, and currently bundles the Tibco messaging service. Additionally, BES is certified to support SonicMQ. Refer to Chapter 24, “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 BES. 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.

For more information, go to the *VisiBroker for Java Developer's Guide* Using the Smart Agent section.

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.

Management

The Borland Management Service encompasses a set of Management Agents which communicate with a Management Hub. The Hub is installed on a single host in your network from which you carry out management tasks such as clustering. The Management Hub lets you monitor and control resources installed on the local host on which Borland Enterprise Server is installed.

Note Borland Deployment Op-Center (purchased separately) provides the ability to manage distributed resources installed on remote hosts with network facing services.

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 BES 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 Enterprise Server. Each Partition instance provides:

- “Connector Service” on page 13
- “EJB Container” on page 13
- “JDataStore Server” on page 13
- “Lifecycle Interceptor Manager” on page 13
- “Naming Service” on page 13
- “Session Storage Service” on page 14
- “Session Storage Service” on page 14
- “Web Container” on page 14

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 Borland Enterprise Server. 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 26, “VisiConnect overview”.

EJB Container

The Borland Enterprise Server 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/bes.
```

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 25, “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. For more information refer to the *VisiBroker for Java Developer's Guide* Using the Smart Agent section.

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, go to Chapter 6, “Java Session Service (JSS) configuration”.

Transaction Manager

A Partition Transaction Manager exists in each Borland Enterprise Server 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, go to Chapter 19, “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). BES 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, go to Chapter 4, “Web components”.

Borland Enterprise Server and J2EE APIs

Since Borland Enterprise Server is fully J2EE 1.3 compliant, it supports the use of the following J2EE 1.3 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 2.0: the Enterprise JavaBeans 2.0 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: 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 version 2.0 from Sun Microsystems. JDBC 2.0 provides APIs for writing database drivers and a full Service Provider Interface (SPI) for those looking to develop their own drivers. JDBC 2.0 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, go to the *VisiBroker for Java Developer's Guide*,Using RMI over IIOP section.

Other Technologies

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

OptimizeIt Profiler

Borland's OptimizeIt Profiler (purchased separately) helps you track memory and CPU usage issues during the development of Java applications. Borland Enterprise Server runs OptimizeIt 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.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

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 BES 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.

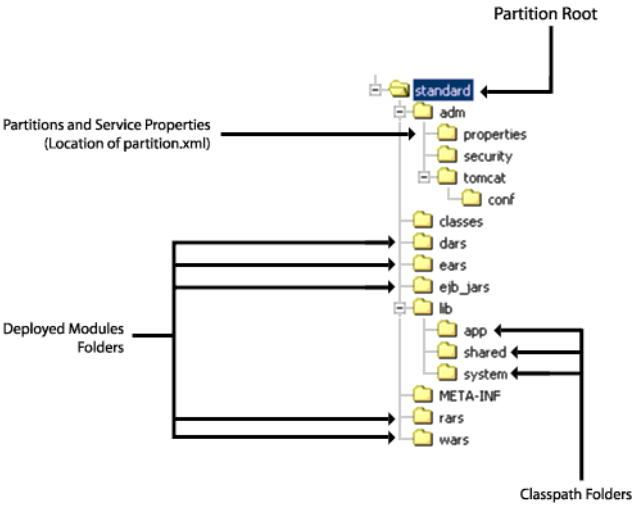
Creating Partitions

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 the *Management Console User's Guide* Using Partitions section. Most configuration data for the Partition and its services is captured in its Partition XML reference file described in Chapter 30, “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 the *Management Console User's Guide*, Using Partitions section, 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 that 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.
`-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.

Table 3.1 Partition command options

Option	Description
-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.

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 an XML 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 18.

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> element 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 in Chapter 30, “Partition XML reference” 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 30, “Partition XML reference”.

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 30, “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 30, “Partition XML reference”.

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 Chapter 25, “Implementing Partition Interceptors”.

Chapter 4

Web components

This section provides information about the web components which are included in all Borland Enterprise Server product offerings with the exception of the *VisiBroker Standalone* installation option.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

Apache web server implementation

BES includes an implementation of the open-source Apache web server version 2.0 (an httpd server) with all product offerings except in the case of the *VisiBroker Standalone* installation option. The Apache web server 2.0 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 BES 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/apache2/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>/mos/
```

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

```
httpd-conf=
```

For more information about the Apache Managed Object elements and attributes, go to the *BDOC Reference*, Managed Object elements and attributes, apache-process Managed Object type section.

For information about configuring the `httpd.conf` file for the IIOP connector/redirector, go to Chapter 5, “Web server to web container connectivity”.

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.

Note For additional information on the Apache web server configuration options and general directive usage, go to the Apache Software Foundation web site at:Apache2 `www.apache.org`, or go to the online Help Topics, *Java API Reference*, Apache2 APIs.

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
`conf`	Contains all configuration files.
`htdocs`	Contains all HTML documents and web pages.
`logs`	Contains all log files.
`CGI-bin`	Contains all CGI scripts.
`proxy`	Contains the proxies for your web application.
`icons`	Contains the icon images in `.gif` format.

Borland web container implementation

The Borland web container supports development and deployment of web applications. All BES products (with the exception of *VisiBroker Standalone* installation option) provide the Borland web container which is based on Tomcat 4.1. The Borland web container is a sophisticated and flexible tool that provides support for Servlets 2.3 and JSP 1.2 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=
```

For more information about the Partition type Managed Object and its elements and attributes, go to the *BDOC Reference*, Managed Object elements and attributes, `partition` Managed Object type section.

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 BES Deployment Descriptor Editor (DDE) to create a Web ARchive (WAR) file, go to the *Management Console User's Guide*, Using the Deployment Descriptor Editor, Adding WAR information section.

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

The `web.xml` file contains the standard deployment descriptor facilities for web applications. However, the `web-borland.xml` file contains some Borland-specific extensions. The following tables describes the Borland-specific elements and how to use them. Some of these augment the standard constructs and some are new constructs.

Note All attributes listed for each element are required.

Table 4.2 Borland-specific new elements

Element	Required	Description	Default Behavior	DDEditor Pane
`context-root`	no	Specifies a user-defined name for the web application. To designate the application as the root web application, type "`!ROOT!`".	By default, the WAR name (without the .war extension) is used for the application if there is no context-root at the EAR level.	General
`web-deploy-path(service, engine, host)`	no	Specifies exactly where to deploy the web application (service, engine, host). The Borland web container (based on Tomcat) has a notion of a host being part of an engine which in itself is a part of a service. There can be multiple hosts under an engine and there can be multiple engines under a given service. A given web application can be deployed to one or more of these hosts. The service, engine, and host you specify using this element, override the defaults. However, this element does accept multiple entries.	By default, the `web-deploy-path` is defined in the following file: `<install_dir>\var\domains\<domain_name>\configurations\<configuration_name>\<partition_managed-object_name>\adm\tomcat\conf\web-borland.xml` If no web-deploy-path is defined in this file, then the default is: `service=HTTP, engine=HTTP, and host=* (deploy to all hosts available under the specified engine)`	Web Deploy Paths

Table 4.2 Borland-specific new elements

Element	Required	Description	Default Behavior	DDEditor Pane
`authorization-domain`	no	Specifies which authorization domain is used for the web application. Because multiple authorization domains can be defined in an application server, you must specify the one for the web application. The authorization domain specified must be one of the domains previously defined. For more information, see the *VisiSecure Guide*, Security Authorization section.	If not defined in the EAR, the domain specified in the WAR is used. If not specified in the WAR also, then the domain specified for the Partition is used.	General
`security-role (role-name, deployment-role?)`	no	Maps the roles used in the web application to the real roles defined in the application server by specifying the (role name, deployment role).	n/a	Open up `.war`, expand Security roles node, select a defined security role to access the Security Roles pane.

Table 4.3 Borland additional attributes on existing elements

Element	Additional Attribute	Description	DDEditor Pane
`resource-ref`	`(res-ref-name, jndi-name)`	Specifies a JNDI name to associate with the resource reference. At runtime, when a servlet looks up the specified resource reference name, the web application looks up the JNDI name in the JNDI. **Note:** The `res-ref-name` that the additional `jndi-name` element modifies is required.	Resource References
`resource-env-ref`	`(resource-env-ref-name, jndi-name)`	Specifies a JNDI name to associate with the resource environment reference. At runtime, when a servlet looks up the specified resource environment reference name, the web application looks up the JNDI name in the JNDI. **Note:** The `resource_env-ref-name` that the additional jndi-name element modifies is required.	Resource Env Refs

Table 4.3 Borland additional attributes on existing elements

Element	Additional Attribute	Description	DDEditor Pane
ejb-ref	(ejb-ref-name, jndi-name)	Specifies a JNDI name to associate with the EJB reference name. At runtime, when a servlet looks up the specified EJB reference name, the web application looks up the JNDI name in the JNDI. **Note:** The ejb-ref-name that the additional jndi-name element modifies is required.	EJB References
ejb-local-ref	(ejb-local-ref-name, jndi-name)	Specifies a JNDI name to associate with the EJB local reference name. At runtime, when a servlet looks up the specified EJB local reference name, the web application looks up the JNDI name in the JNDI. **Note:** The ejb-local-ref-name that the additional jndi-name element modifies is required.	EJB Local References

This is the DTD for the web-borland.xml file:

Note "*" means you can specify more than one, "?" means you can only specify one.

```
<!ELEMENT web-app(context-root?, resource-env-ref*, resource-ref*,
        ejb-ref*, ejb-local-ref*, property*, web-deploy-path*,
        authorization-domain?, security-role*)>

<!ELEMENT ejb-ref (ejb-ref-name, jndi-name)>
<!ELEMENT ejb-local-ref (ejb-ref-name, jndi-name?)>
<!ELEMENT resource-ref (res-ref-name, jndi-name)>
<!ELEMENT resource-env-ref (resource-env-ref-name, jndi-name)>
<!ELEMENT web-deploy-path (service, engine, host)>
<!ELEMENT context-root (#PCDATA)>
<!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)>
<!ELEMENT resource-env-ref-name (#PCDATA)>
<!ELEMENT service (#PCDATA)>
```

```
<!ELEMENT engine (#PCDATA)>
<!ELEMENT host (#PCDATA)>
<!ELEMENT authorization-domain (#PCDATA)>
<!ELEMENT security-role (role-name, deployment-role?)>
<!ELEMENT role-name (#PCDATA)>
<!ELEMENT deployment-role (#PCDATA)>
```

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.

The `configuration.xml` file is 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>
```

For more information, go to the *BDOC Reference*, Managed Object elements and attributes, `process` sub-elements section.

Microsoft Internet Information Services (IIS) web server

The Microsoft Internet Information Services (IIS) web server is not included with any BES product offerings. However, BES 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, go to Chapter 5, “Web server to web container connectivity”.

IIS/IIOP redirector directory structure

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

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

Table 4.4 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, go to the *VisiBroker for Java Developer's Guide*, Using the Smart Agent section, or the *VisiBroker for C++ Developer's Guide*, Using the Smart Agent section.

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.

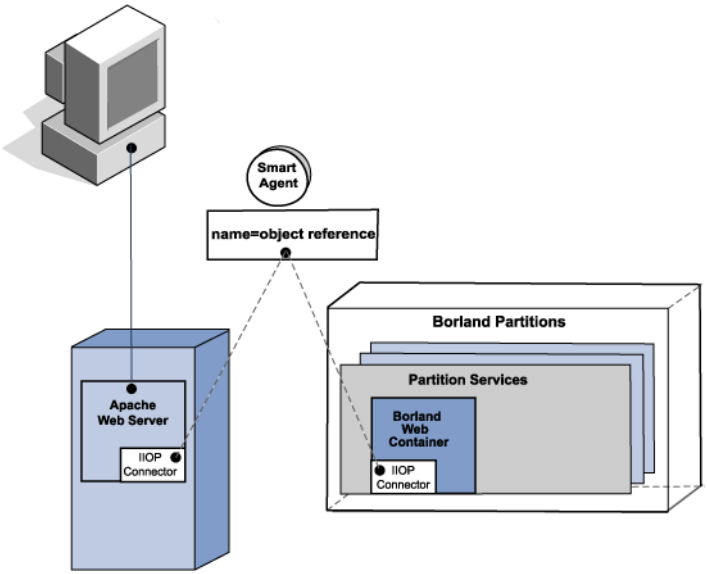
The Smart Agent must be started on at least one host within your local network. When your client program invokes an object (using the `bind` method), the Smart Agent is automatically consulted. The Smart Agent locates the specified object implementation so that a connection can be established between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

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).

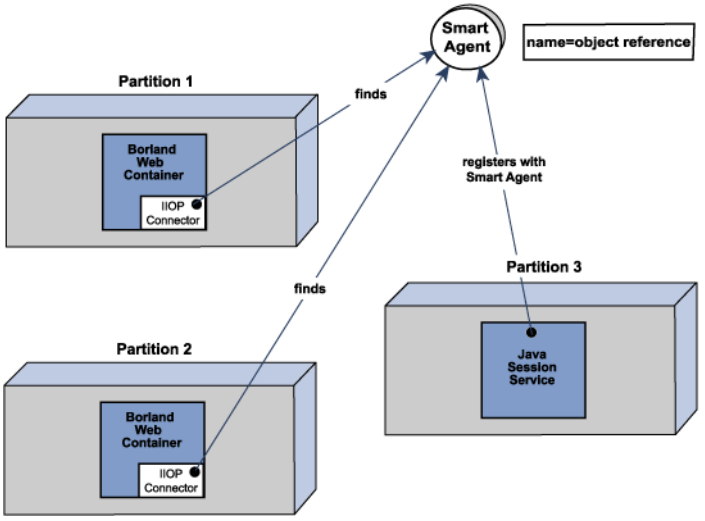
Figure 4.1 Client program binding to an object reference



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), go to 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 BES products. For information about Apache to CORBA connectivity, go to Chapter 8, “Apache web server to CORBA server connectivity”.

Important For documentation updates, go to www.borland.com/techpubs/bes.

Apache to Borland web container connectivity

All BES product offerings include an implementation of the open-sourced Apache web server version 2.0 as well as the Tomcat-based Borland web container (with the exception of the *VisiBroker Standalone* installation option). Also included is the IIOP connector, which provides connectivity from the Apache web server to the Tomcat-based Borland web container.

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, go to Chapter 4, “Web components”.

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" />
```

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_inst1	The name by which this connector can be reached by Apache and IIS 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, go to "Downloading large data" on page 50 .
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 charactistics. 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, go to "Downloading large data" on page 50.

Table 5.1 IIOP connector attributes

Attribute	Default	Description
port	0 (zero)	The IIOP connector port. If set to 0 (zero) - the default, a random port gets picked. 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).
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 attribue to false.For more information, go to "Browsers supporting only the HTTP 1.0 protocol" on page 51.

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/apache2/mod_iiop2.dll
  IIopLogFile <install_dir>/var/domains/<domain_name>/
configurations/<configuration_name>/mos/apache2/logs/mod_iiop.log
  IIopLogLevel error
  IIopClusterConfig <install_dir>/var/domains/<domain_name>/
configurations/<configuration_name>/mos/apache2/conf/WebClusters.properties
  IIopMapFile <install_dir>/var/domains/<domain_name>/
configurations/<configuration_name>/mos/apache2/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/ apache2/mod_iiop2.dll	Enables Apache 2.0 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/ apache2/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/ apache2/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/ apache2/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 2.0:

Windows example

```
LoadModule iiop2_module C:/BES/lib/apache2/mod_iiop2.dll
IIopLogFile C:/BES/var/domains/base/configurations/j2ee/mos/
```

```
apache2/logs/mod_iiop.log
  IIopLogLevel error
  IIopClusterConfig C:/BES/var/domains/base/configurations/j2ee/
mos/apache2/conf/WebClusters.properties
  IIopMapFile C:/BES/var/domains/base/configurations/j2ee/mos/
apache2/conf/UriMapFile.properties
```

Solaris example

```
  LoadModule iiop2_module /opt/BES/lib/apache2/mod_iiop2.so
  IIopLogFile /opt/BES/var/domains/base/configurations/j2ee/mos/
apache2/logs/mod_iiop.log
  IIopLogLevel error
  IIopClusterConfig /opt/BES/var/domains/base/configurations/j2ee/
mos/apache2/conf/WebClusters.properties
  IIopMapFile /opt/BES/var/domains/base/configurations/j2ee/mos/
apache2/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) `false`	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 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, go to "Implementing chunked download" 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 charactistics. 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, go to "Implementing chunked upload" on page 52.
`IIopSessionAffinity`	`true`	Controls whether Apache employs "session affinity" in it's request handling. When uncommented and set to `true`, Apache ensures that all requests associated with a particular session id are routed to the Borland web container from which the request originated. To disable this mechanism and have all requests be subject to the round-robin model configured for the particular cluster, set to `false`. **Note:** If session affinity is disabled (`=false`), it is crucial to ensure that the Borland web container's shared session store is correctly configured; otherwise the session data's integrity will be compromised. For more information, go to the Clustering of multiple Web components, Smart session handling section.

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/apache2/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 table, “Cluster definition attributes” on page 60). For example:

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

Note Failover and smart session are always enabled, for more information go to Chapter 7, “Clustering web components”.

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,

go to the *Management Console User's Guide*, Using the Deployment Descriptor Editor, Web Deploy Paths section.

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 47).

- 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 BES options available to you for handling large data transfers between a client and the Borland web container with Apache 2.0 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, go to Chapter 4, “Web components”.

To enable the chunked download:

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. By default, the download buffer size is set to `4096`. To change it, 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, go to the Chapter 4, “Web components”.

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

  ```
  <install_dir>/var/domains/<domain_name>/configurations/
  <configuration_name>/mos/apache2/conf
  ```

  For more information, go to Chapter 4, “Web components”.

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 to Borland web container connectivity

All BES product offerings (with the exception of the *VisiBroker Standalone* installation option) include the Tomcat-based Borland web container and its IIOP connector. Also included is the IIS/IIOP redirector which provides connectivity from the Microsoft Internet Information Services (IIS) web server (not included with BES products) to the Borland web container.

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, go to "Modifying the Borland web container IIOP configuration" on page 41 under the *Apache to Borland web container connectivity* section.

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 IIS/IIOP redirector configuration by implementing the following steps. For information on IIOP configuration for Windows 2003/IIS version 6.0, go to: `www.borland.com/devsupport/bes/faq`.

Windows 2000/IIS version 5.0

1 Configure the System PATH variable to include `<install_dir>\bin\` folder.

  As IIS runs as a system process, in order for the IIS/IIOP redirector to load successfully, the Visibroker ORB dlls need to be in the system path. Ensure that the `\<install_dir>\bin\` is included in the Windows 2000 system path.

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:\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 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 dropdown 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.

5 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.

6 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
```

Windows XP/IIS version 5.1

1 Configure the System PATH variable to include `<install_dir>\bin\` folder.

As IIS runs as a system process, in order for the IIS/IIOP redirector to load successfully, the Visibroker ORB dlls need to be in the system path. Ensure that the `<\install_dir>\bin\` is included in the Windows 2000 system path.

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 and the Web Sites node.

   d Under the Web Sites node, 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:\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 Restart IIS by stopping **then** starting the IIS Service:

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

b In the Stop/Start/Reboot dialog, from the dropdown 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 All Tasks | 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.

5 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.

6 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 table, "Cluster definition attributes" on page 60). For example:

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

Note Failover and smart session are always enabled, for more information go to Chapter 7, "Clustering web components".

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, go to the *Management Console User's Guide*, Using the Deployment Descriptor Editor, Web Deploy Paths section.

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 47).

- 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, go Chapter 4, “Web components”.

Session management with JSS

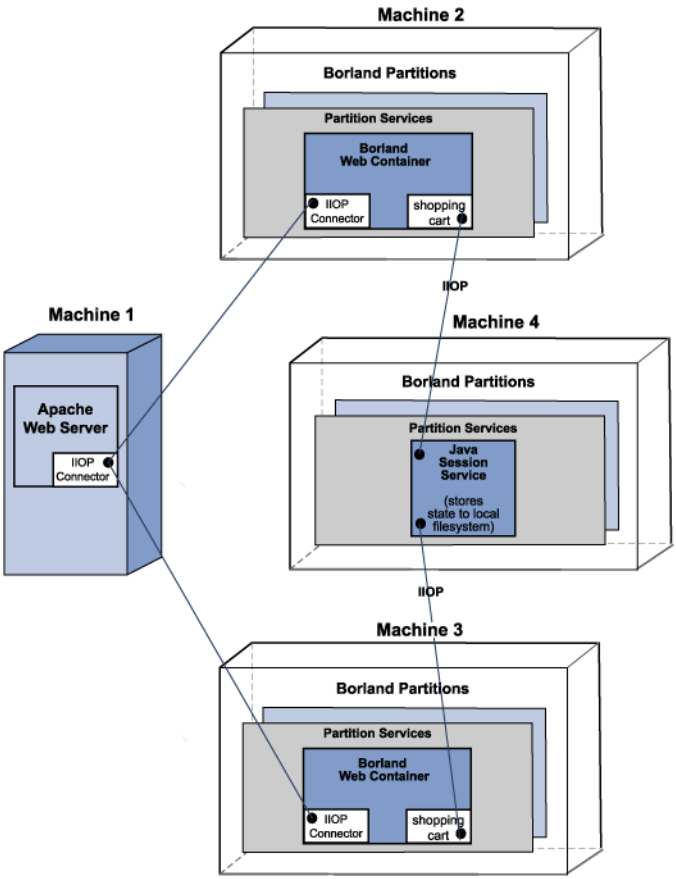
The following diagrams show typical landscapes of web components and how session information is managed by the JSS. The JSS session management is completely transparent to the client.

In the diagram, “JSS Management with a Centralized JSS and Two Web Containers” on page 64, 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

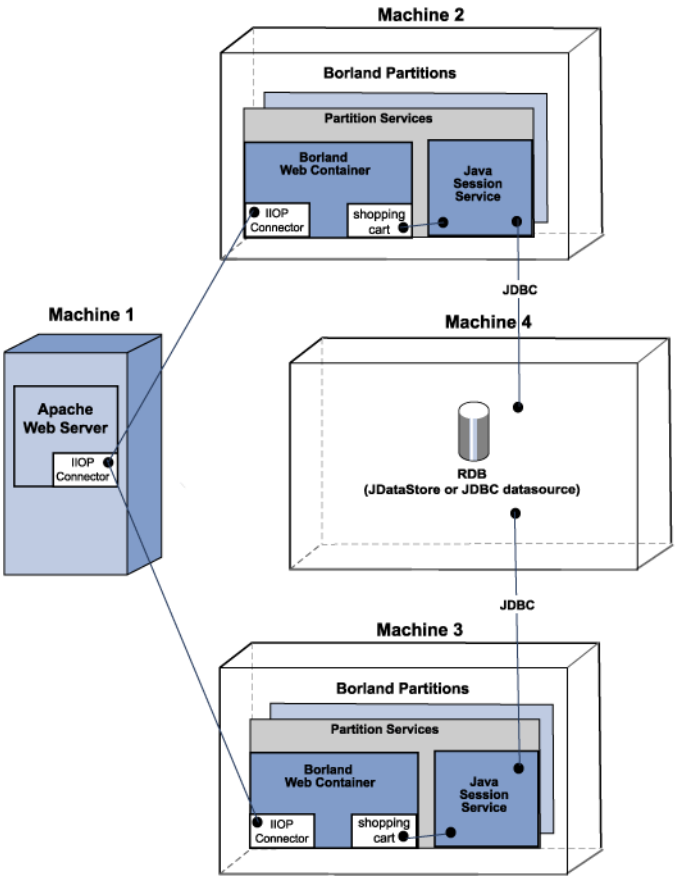


In the diagram, “JSS Management with Two Web Containers and a Centralized Backend Datastore” on page 66, are the following four virtual machines:

- 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. BES supports two types of configurations; the default is to use a JDatastore, but BES supports any JDBC datasource.

- 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" STRING PRIMARY KEY, "KEY_BASE"
BIGINT);
CREATE TABLE "JSS_WEB" ("KEY" STRING PRIMARY KEY, "VALUE" BINARY,
"EXPIRATION" BIGINT);
```

```
CREATE TABLE "JSS_EJB" ("KEY" STRING PRIMARY KEY, "VALUE" BINARY,
"EXPIRATION" BIGINT);
```

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, go to the Chapter 30, “Partition XML reference”, <service> element section.

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 (mo) directory attribute:

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

For more information about the `configuration.xml`, go to the *BDOC Reference*, Configurations and the configuration.xml file section.

For a listing and description of the session service (JSS) level properties, go to Chapter 31, “EJB, JSS, and JTS Properties”.

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, go to Chapter 4, “Web components”.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

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 Enterprise Server includes the IIOP connector for the Apache 2.0 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.

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

- "Load balancing support" on page 70
- "Fault tolerance (failover)" on page 71
- "Smart session handling" on page 72

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" on page 70
- "Corbaloc based load balancing" on page 70

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, go to Chapter 5, "Web server to web container connectivity".

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, go to Chapter 5, "Web server to web container connectivity".

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, go to Chapter 5, "Web server to web container connectivity".

```
<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`, go to Chapter 6, "Java Session Service (JSS) configuration".

Session storage implementation

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

- "Programmatic implementation" on page 73
- "Automatic implementation" on page 73

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, go to Chapter 5, “Web server to web container connectivity”.

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.

BES 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.

Note In order to implement the `ReqProcessor` IDL, you need to be running one of the following BES product offerings:

- Team Edition
- VisiBroker Edition
- AppServer Edition

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

Web-enabling your CORBA server

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

- “Determining the urls for your CORBA methods” on page 78.
- “Implementing the ReqProcessor IDL in your CORBA server” on page 78.

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, go to the *Management Console User's Guide*, Using the Deployment Descriptor Editor, Web Deploy Paths.

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

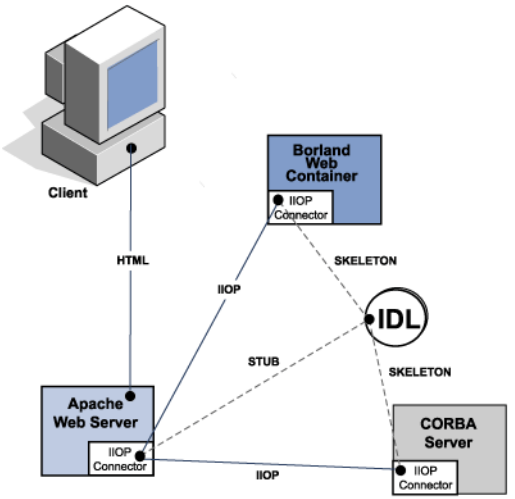
The `ReqProcessor` IDL includes the `process()` method which your Apache web server calls for internet requests. The web server passes the user's request as an argument to the `process()` method. Basically, the input for the `process()`

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, go to Chapter 5, “Web server to web container connectivity”.

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/apache2/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 table, "Cluster definition attributes" on page 82). For example:

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

Note Failover and smart session are always enabled, for more information go to Chapter 7, “Clustering web components”..

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 Enterprise Server Web Services

The Borland Enterprise Server provides an out-of-the-box web services capability in all Borland Partitions.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

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:

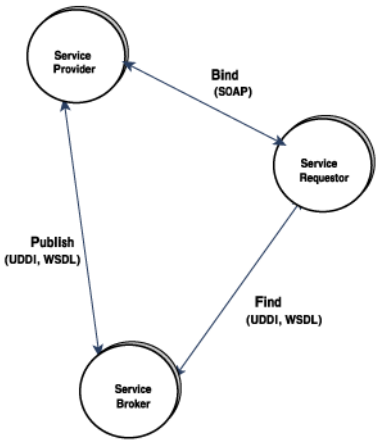
1 The *Service Provider* registers all available web services with the Service Broker.

2 The *Service Broker* publishes the web services for the Service Requestor to access. The information published describes the web service and its location.

3 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

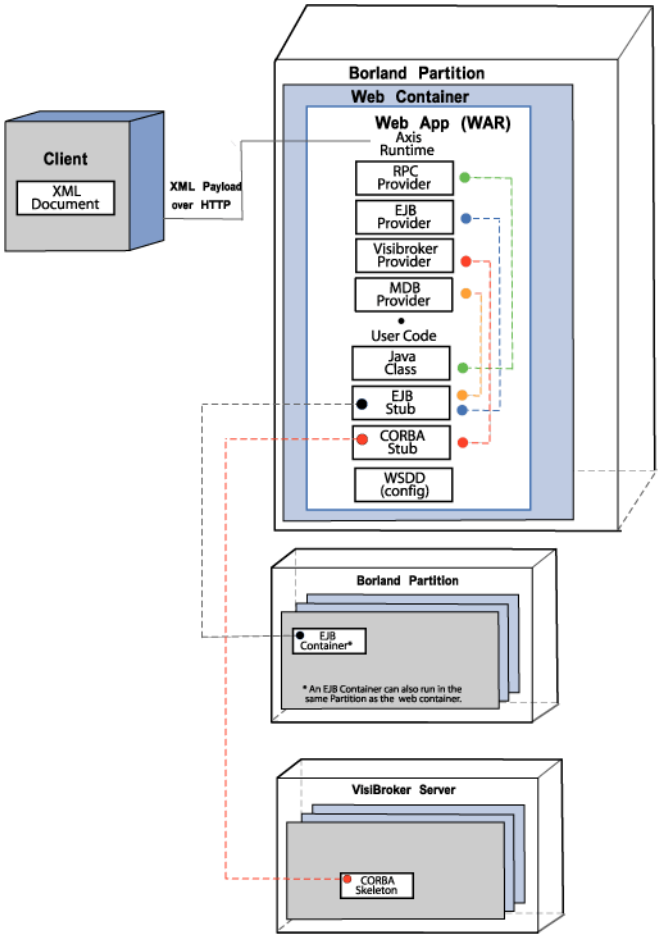
All BES Partitions are configured to support web services. You simply need to start a Partition and deploy WARs (or EARs containing WARs) containing web services.

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

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
- VisiBroker providers (Java and/or C++)
- 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 the *Management Console User's Guide,*Export EJB as a Web Service Wizard.

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>
```

Java:VISIBROKER provider

This provider assumes that the class serving the web service is a Visibroker server. When a web service request arrives, this provider:

1. Initializes the ORB.
2. Locates the CORBA object using "objectName" and "locateUsing" properties.
3. Invokes the specified method on the CORBA stub.

The parameters are:

- `objectName`: The name of the object. This is a mandatory parameter.
- `locateUsing`: This parameter specifies which mechanism the provider uses to locate the object.

The 3 possible values are:

`osagent` The object is assumed to be in osagent. The `bind()` method on helper is used to locate the object. The Server object must be registered with the same osagent that the Partition is using. If `poaName` is also specified, `objectName` is located under that POA. This is the default value of the parameter.

`nameservice` The object is assumed to be in Naming Service. The `resolve()` method on the root context is used to locate the object. The objectName must be the full name of the object starting from root context. For example:

```
USA/California/SanMateo/BigAnimal
```

The VisiBroker server object must be registered with the same name service that the Partition is using. Typically, a Partition starts a name service with the name "namingService". For a server to use the same name service, it must be started as follows:

```
vbj -DSVCnameroot=namingservice Server
```

`ior` The `objectName` provided is assumed to be an IOR. The `string_to_object()` method on the ORB is used to obtain the object. The IOR can be in standard form. For example:

```
corbaname::172.20.20.28:9999#USA/California/
SanMateo/BigAnimal
IOR:xxx
corbaloc::172.20.20.28:30303,:172.20.20.29:30304/
MyObject
```

- `className`: The name of the class that is loaded when a request arrives on this service. Typically, this is the name of the CORBA class generated and based on the IDL. The VisiBroker client stubs, including this class, must be present in the WAR.
- `allowedMethods`: The methods that are allowed to be invoked on this class. The CORBA object can have more methods than listed here; the methods listed here are available for remote invocation.
- `poaName`: The name of the POA that hosts the CORBA object. This parameter is relevant only when the `locateUsing` parameter value is `osagent`.

Example:

```
  <service name="Animal" provider="java:VISIBROKER">
    <parameter name="className" value="com.borland.examples.web
services.visibroker.AnimalModule.Animal"/>
    <parameter name="allowedMethods" value="talk sleep"/>
    <parameter name="objectName" value="BigAnimal"/>
    <parameter name="cacheObject" value="false"/>
```

```
  </service>
   <service name="Animal2" provider="java:VISIBROKER">
     <parameter name="className" value="com.borland.examples.web
services.visibroker.AnimalModule.Animal"/>
     <parameter name="allowedMethods" value="talk sleep"/>
     <parameter name="objectName" value="corbaname::172.20.20.28:9999#USA/
California/SanMateo/BigAnimal"/>
     <parameter name="locateUsing" value="ior"/>
  </service>
```

Java:MDB provider

This provider assumes that the incoming message is meant for a message queue or a topic. When a web service request arrives, this provider:

1. Looks up the connection factory.
2. Creates a Queue or a Topic connection.
3. Sends the message to the queue or topic.

The parameters are:

- `connectionType`: The type of connection. Can take value QUEUE or TOPIC.
- `connectionFactory`:; The name of the connection factory. This is the serial name and not the JNDI name of the object. For example: `serial://jms/qcf`.
- `destination`: The name of the queue or topic to which the message is sent. This is the serial name and not the JNDI name of the object. For example: `serial://jms/q`

Examples:

```
<service name="MDBQService" provider="java:MDB">
   <parameter name="ConnectionType" value="QUEUE"/>
   <parameter name="ConnectionFactory" value="serial://jms/qcf"/>
   <parameter name="Destination" value="serial://jms/"/>
</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, go to:

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.

NOTE For more information on the web services deployment descriptor (WSDD), refer to the Axis User Guide located in `<install_dir>/doc/axis/user-guide.html`, or go to the 3rd Party Documentation, Axis Documentation section.

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" on page 96 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, go to the *Management Console User's Guide*, Viewing J2EE component configurations, Viewing Web Services WSDD properties section, or the Using the Deployment Descriptor Editor, Web Services section.

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 "Tools Overview" on page 96 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 Borland Enterprise Server installation in:

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

The examples that illustrate the different "Web Service providers" on page 88, are located in the web services examples directory in the `Java`, `EJB`, `MDB` or `VisiBroker` folder.

Your Borland Enterprise Server installation also includes several "Apache Axis Web Service samples" on page 96 in:

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

Using the Web Service provider examples

BES 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 "Tools Overview" on page 96, while deployment is performed using Tools Overview with BES.

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 BES. 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 go to Chapter 29, "iastool command-line utility".

- Deployment Wizard, for more information go to the *Management Console User's Guide*, Deployment Wizard section.

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 BES installation and is located in:

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

or go to the *Third Party Documentation*, Axis Documentation section.

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 Borland Enterprise Server 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

Web applications bundled with BES

This section describes the Apache Cocoon and Apache Struts for which BES includes support. For more information about Apache, go the *3rd Party Documentation*, Apache2 section.

The Cocoon servlet is pre-installed in the Tomcat web container upon which the Borland web container is based. The Struts framework is also pre-installed in the Borland web container.

For more information about the Borland web container, go to Chapter 4, “Web components”.

About Cocoon

Cocoon is a web publishing framework that renders XML data into a number of formats including HTML, WML, PDF, and the like. These formats are based on a set of properties provided by an XSL stylesheet.

All BES product offerings include the Cocoon servlet pre-installed in the Borland web container. The `web.xml` file contains specifications for the Cocoon servlet, which is pre-configured to point to the cocoon properties file in the `WEB-INF` directory of the context under which Cocoon is invoked. However, you can change this specification to point to another location. Additionally, the `web.xml` file includes a mapping for "*.xml" directing any such file to be processed by Cocoon. This mapping can be modified or deleted.

An unmodified copy of the properties file is included as part of the Cocoon webapp (`cocoon.war`) distributed with BES. You can customize this file to create your own `cocoon.properties` file. This unmodified `cocoon.properties` file is compiled in with the Borland web container. As the default properties file, it is used whenever the file specified in the Cocoon servlet definition cannot be found.

Cocoon is an open-source product distributed by the Apache Software Foundation. For additional information about Cocoon, see the web site:

```
http://xml.apache.org/cocoon
```

Chapter

11

Writing enterprise bean clients

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

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. Go to "Initializing the client" on page 102 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 Exception. When 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. Go to "Using a bean's handle" on page 106 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 manages 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.

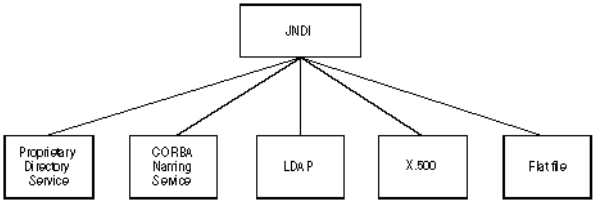
The `getMetaData()` method is most often used by development environments and tool builders that need to discover information about an enterprise bean, such as for linking together beans that have already been installed. Scripting clients might also want to obtain metadata on the bean.

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_refernces("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

12

The VisiClient Container

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

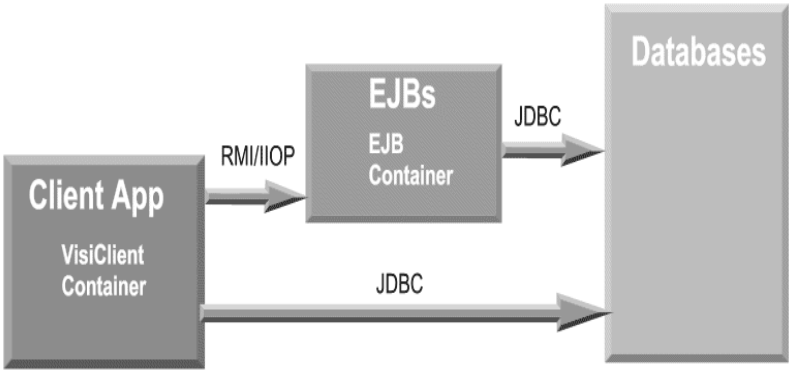
Containers are an integral part of J2EE applications. Most applications provide containers for each application type. Application clients depend on their containers to supply system services to all J2EE components.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

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 12.1 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 Enterprise Server Deployment Descriptor Editor for packaging and editing application client components. For more information, go to the *Management Console User's Guide,* Using the Deployment Descriptor Editor section..

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 Enterprise Server 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 Enterprise Server *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>file:///net/machine/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 Microsystem's 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 Microsystem's EJB 2.0 and J2EE 1.3 specifications.

The Borland Enterprise Server 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 12.1 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 BES

To run a J2EE application client on a client machine that does not have Borland Enterprise Server 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 Enterprise Server *Developer's Guide* for details about configuration and deployment. Client applications in Borland Enterprise Server 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

Borland Enterprise Server 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 Chapter 29, “iastool command-line utility”, `iastool verify` section on the available options.

Chapter

13

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 (including the setting of properties) 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.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

Passivating Session Beans

At deployment time, the deployer uses the Borland Enterprise Server's 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 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*, elb-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

14

Entity Beans and CMP 1.1 in Borland Enterprise Server

Here we'll examine how entity beans are deployed in the Borland Enterprise Server 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.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

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 Borland Enterprise Server 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 Entrollment 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 Borland Enterprise Server

The Borland Enterprise Server'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 16, “Using BES Properties for CMP 2.x”.

BES 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 the *Management Console User's Guide,* Using the Deployment Descriptor Editor section 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 Enterprise Server *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 the EJB Designer Tab in 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 Enterprise Server AppServer Edition, to set up all of the container-managed persistence information. You should refer to the *Borland Enterprise Server*

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
Container-Managed Fields Description	Bean:CMP1.1	CMP 1.1
Finders	Bean:CMP1.1	Finders
Finder Method	Bean:CMP1.1	Finders

Descriptor Element	Navigation Tree Node/ Panel Name	DDEditor Tab
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 table shows the basic and complex types supported by the Borland EJB Container:

Basic types:	`boolean Boolean`	`byte Byte`	`char Character`
	`double Double`	`float Float`	`int Integer`
	`long Long`	`short Short`	`String` `java.sql.Date`
	`BigDecimal` `java.util.Date`	`byte[]`	`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, 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 type	JDBC SQL type
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

Java types	Oracle	Sybase/MSSQL	DB2
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

15

Entity Beans and Table Mapping for CMP 2.0

Here we'll examine how entity beans are deployed in the Borland Enterprise Server 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.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

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 Borland Enterprise Server 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 Borland Enterprise Server

The Borland Enterprise Server'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, the relevant parts of which are the *Enterprise JavaBeans Developer's Guide* 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 2.0 implementation

In CMP 2.0, 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 2.0 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 Chapter 32, “ejb-borland.xml” 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. BES 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, found in the Table Properties section of Chapter 16, "Using BES Properties for CMP 2.x".

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 SQL generated from EJB-QL by the CMP engine. For more information, go to Chapter 17, "EJB-QL and Data Access Support". 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 possibilty that data integrity could be compromised.

The Borland Enterprise Server 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.

SelectForUpdateNoWAIT

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 2.0 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, you should refer to “Basic Mapping of CMP fields to columns” on page 158, below, 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 a class used to obtain connections to a database, JMS implementation, or backing store of some other type. Information on datasource configuration is in Chapter 21, “Connecting to Resources with BES: 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 QUANITY 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 `QUANITY.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

Borland Enterprise Server 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

16

Using BES 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 Enterprise Server *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 the EJB Designer Tab in the Deployment Descriptor Editor or in the XML directly.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

Using the Deployment Descriptor Editor

You can use the Deployment Descriptor Editor, which is part of the Borland Enterprise Server, to set up all of the container-managed persistence information. You should refer to the *Borland Enterprise Server Management Console User's Guide* for complete information on the use of the Deployment Descriptor Editor and other related tools. Most operations for CMP 2.x are

performed via the EJB Designer, a component of the Deployment Descriptor Editor. The following table shows descriptor information and where in the Deployment Descriptor Editor that information can be entered.

The EJB Designer

CMP 2.x properties are set using the EJB Designer. The EJB Designer appears as a Tab in the Deployment Descriptor Editor. See the *Management Console User's Guide,* Using the Deployment Descriptor Editor section for details on this feature.

J2EE 1.3 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

Descriptor Element	Navigation Tree Node/ Panel Name	DDEditor Tab
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

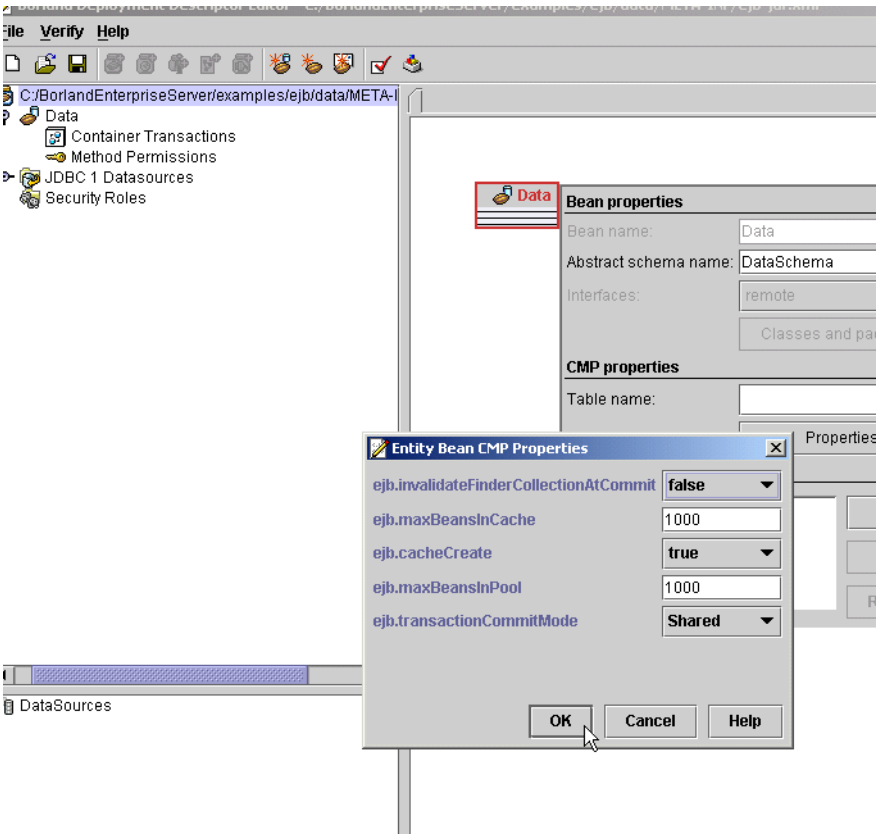
Borland Enterprise Server uses its EJB Designer, a component of the Deployment Descriptor Editor, to set CMP 2.x properties. The EJB Designer is fully-documented in the *User's Guide*.

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, shown in Chapter 32, “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.maxBeans InCache	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.maxBeans InPool	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().

Property	Type	Default	Description
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.

Property	Type	Default	Description
ejb.datasource	java.lang.String	N/A	Default JDBC datasource to use in case no table-properties have been set. CMP 2.x only.
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, BES 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`.
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.

Property	Type	Default	Description
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.
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.Bytejava.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

17

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 Enterprise Server provides some extensions to the EJB-QL functionality described in the Sun Microsystems EJB 2.x Specification.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

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 correspoding 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
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

Aggregate Function	Argument data type	Expected return type
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 Borland Enterprise Server 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
```

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 selectDynamicRemote(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)
      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.

Note There should not be any trace of the four 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 table shows the basic and complex types supported by the Borland EJB Container:

Basic types:	boolean Boolean	byte Byte	char Character
	double Double	float Float	int Integer
	long Long	short Short	String java.sql.Date
	BigDecimal java.util.Date	byte[]	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 explicitely 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>
```

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

18

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 in "Generating primary keys from a custom class" on page 192. To use the database tools, you can refer to "Implementing primary key generation by the CMP engine" on page 192 for information on the CMP engine generating primary keys depending upon the database vendor.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

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 Enterprise Server, 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. Setting table and column properties is discussed here.

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

19

Transaction management

This chapter describes how to handle transactions.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

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

BES 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 BES:

- Transaction Manager (formerly known as Partition Transaction Service)
- OTS (formerly known as 2PC Transaction Service)

A Transaction Manager exists in each BES 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 BES 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 BES for two-phase commit transactions, they are inherently expensive due to number of remote procedure calls (RPCs) and should be used only when needed. Go to, "When to use two-phase commit transactions" on page 198.

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 BES 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 BES, 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 a BES 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>serial://QueueConnectionFactory</jndi-name>
            </resource-ref>
            <resource-env-ref>
                <resource-env-ref-name>jms/OrderQueue</resource-env-ref-
name>
                <jndi-name>serial://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 a BES 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.

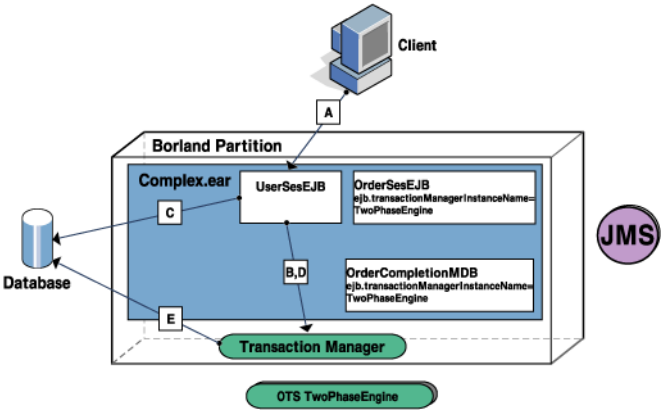
To configure this example deployment scenario:

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 BES Partition, and how the BES Partition locates the right transaction service based on the above configuration. All of the beans are assumed to have container-managed transactions.

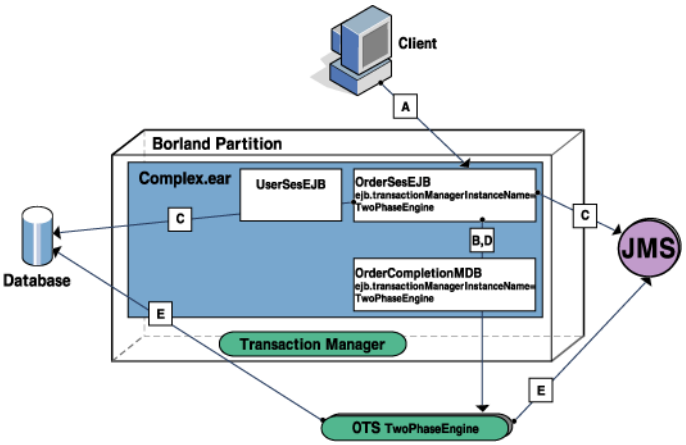
Example 1PC usage

1. (A) The client calls a method of **`UserSesEJB`**. This is an implementation of the method that creates users in the database.
2. (B) Before the call is actually invoked, as shown below, the Partition uses its in-process Transaction Manager to begin the transaction.
3. (C) The session bean does some database work.
4. (D) When the call is over, the Partition issues commit.
5. (E) The Transaction Manager calls `commit_one_phase()` on the database resource.



Example 2PC usage

1. (A) The client calls `OrderSesEJB.create()` method to create a new order.
2. (B) 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. (C) The session bean does some database work, and sends a message to a JMS queue.
4. (D) When the call is over, the Partition issues commit.
5. (E) 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 Chapter 20, “Message-Driven Beans and JMS” 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 local transaction is a transaction that is managed by the resource manager such as that associated with a database. A global transaction, on the other hand, is a transaction managed by the Transaction Manager or the OTS engine which are global transaction managers.

An EJB that uses bean-managed transaction demarcation uses the `javax.transaction.UserTransaction` interface to explicitly demarcate global transaction boundaries. When transaction demarcation is container managed, the container interposes each client call to control the transaction demarcation. It then controls this demarcation declaratively, according to the transaction attribute set by the Application Assembler in the deployment descriptor. The transaction attribute also determines whether the transaction is local or global.

The EJB container follows certain rules to determine when it is to do a local versus a global transaction for container-managed transactions. In general, a container calls the method within a local transaction after verifying that no global transaction already exists. It also verifies that it is not expected to start a new global transaction and that the transaction attributes are set for container-managed transactions. The container automatically wraps a method invocation within a local transaction if one of the following is true:

- If the transaction attribute is set to NotSupported and the container detects that database resources were accessed.
- If the transaction attribute is set to Supports and the container detects that the method was not invoked from within a global transaction.
- If the transaction attribute is set to Never and the container detects that database resources were accessed.

The EJB container supports the following characteristics for local transactions:

- 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.

The EJB container supports the following characteristics for global transactions:

- Local transactions support time-outs for database connections and transactions.
- Local transactions are lightweight from a performance standpoint.

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.

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 BES to access databases that support JDBC through vendor provided drivers. Requests for access to a database is centralized through the BES JDBC Connection Pool. This section describes modifications the BES 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:

`www.javasoft.com/products`

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

20

Message-Driven Beans and JMS

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

JMS and EJB

According to the EJB 2.0 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 wanting the sending/receiving of the message to share the transaction context of some other piece of work. We already know how to solve this problem using JMS and JTA in conjunction. 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 using JMS APIs 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.

To the client, 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.

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 JMS message to the queue/topic to which the MDB is configured to listen. Typically clients also use the `<resource-ref>` and `<resource-env-ref>` elements of their deployment descriptor and then point to the same JNDI names as the MDB's `<connection-factory-name>` and `<message-driven-destination-name>` descriptor elements. See Chapter 23, “Using JMS” for information on how to configure your deployment descriptors.

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.

Naming Support and Configuration

Since MDBs have no interfaces, MDBs do not have JNDI names in the normal sense like `EJBHome` objects do. Instead, they are associated with two objects that must preexist in JNDI before the MDB is deployed. These are:

- a connection factory to use for connecting to the JMS services provider and
- a 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.

Connecting to JMS Connection Factories from MDBs

MDBs provide a special case for connecting to JMS connection factories. In `ejb-jar.xml`, attaching a JMS resource to an MDB requires the `<message-driven-destination>` entry 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 uses of this element. Again, the Borland-specific XML file binds the logical name with the JNDI name. It uses two elements to accomplish this: `<connection-factory-name>` and `<message-driven-destination>`.

First, let's examine the `<connection-factory-name>` element from the ejb-borland.xml DTD:

```
<!ELEMENT connection-factory-name (#PCDATA)>
```

It's hardly a complicated element. Its value is the JNDI name of the JMS topic or queue connection factory. That is, it is identical to the `<jndi-name>` element found within the `<resource-ref>` declaration we just discussed. So, the `ejb-borland.xml` declaration for an MDB looks like this:

```
<message-driven>
 <ejb-name>MyMDBTopic</ejb-name>
 ...

 <connection-factory-name>serial://resources/tcf</connection-factory-name>
```

Now, we need to specify the JNDI name of the individual queue or topic. This is done with the `<message-driven-destination>` element. Let's have a look at its DTD entry.

```
<!ELEMENT message-driven-destination-name (#PCDATA)>
```

This is yet another simple element. It's value is essentially identical to the value of the `<jndi-name>` element found within the `<resource-env-ref>` declaration discussed above. Now our XML looks like the following:

```
<message-driven>

 <ejb-name>MyMDBTopic</ejb-name>

 ...

 <message-driven-destination>serial://resources/tcf</message-driven-
destination>

 <connection-factory-name>serial://resources/tcf</connection-factory-name>

 ...

</message-driven>
```

Once deployed, you can access datasources from the MDB `MyMDBTopic`.

The connection factory objects are much like JDBC datasources and the names of topics and queues are similar to specifying a database table name. The connection factory objects themselves are typed, for example `TopicConnectionFactory` and `QueueConnectionFactory`, depending upon what you are listening for. Furthermore, each of subtype of XA flavor is are needed when global transactions are in effect. In the typical development environment, there is only one JMS service provider, so only a pair of connection factories are required: one for topics and one for queues. Go to Chapter 23, “Using JMS”, Configuring JMS Connection Factories section for more information on configuring these.

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.

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. If you have the Borland Deployment Operations System (BDOC) installed and running, it will automatically maintain the desired state of its clustered services under active management.

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 fafaultolerance features, refer to Chapter 24, “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 to the JMS service when an MDB fails to consume a message. The section on redelivery covers setting the redeliver attempt property as well as two properties for delivering messages to a dead queue.

Rebinding

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

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>serial://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

See Chapter 23, “Using JMS” for information on using JMS within transactions. This section deals exclusively with using MDBs in transactions.

The most common scenario of using MDBs in transactions is the desire to have two-phase commit (2PC) completion of transactions started for the `onMessage()` method. Such an MDB will have the `REQUIRED` transaction attribute. However, if you have a Partition's 2PC transaction service running, then all modules deployed to that Partition can discover it and attempt to use it. The 2PC engine will perform one-phase commit when only one resource is registered, but it will suffer the RMI overhead of the 2PC engine when the Partition's standard transaction manager would perform far better. To avoid this possible waste of resources, you can add the following bean-level property to the MDBs that require 2PC. Use the Console to navigate to the module and Right-click it. Select DDEditor and select the required bean from the Navigation Pane once the DDEditor appears. Click the Properties tab and add this property. Define this property as a string:

```
ejb.transactionManagerInstanceName
```

For more information on 2PC and the transaction manager, go to Chapter 19, “Transaction management”.

Note `ejb.transactionManagerInstance` is also supported for Entity and Session beans.

Chapter

21

Connecting to Resources with BES: using the Definitions Archive (DAR)

J2EE specifies a uniform mechanism to establish connections to relational databases using JDBC datasources, message brokers using JMS resource objects, and general Enterprise Information Systems (EIS) using Connectors resource adapters. JNDI bound objects called *resource connection factories* are used to implement this mechanism. For example, a JDBC datasource is an object that your applications use to establish connections to a database. In the Borland Enterprise Server, creating, editing, and deploying all types of resource entries is done using the Management Console and DDEditor. You capture the properties that define the resource objects in an XML descriptor and then use normal deployment procedures to bind the objects into JNDI. The Borland Partition's Naming Service takes care of creating the Java objects representing the data that was provided in the descriptor and binding it to the proper JNDI name during deployment. Once bound, you may reference the datasource from your enterprise beans and servlets using the resource-reference elements in their deployment descriptors. This is a portable way to specify a particular database instance, message broker instance, or legacy instance.

Borland provides mechanisms for deploying both. Go toChapter 22, “Using JDBC” and Chapter 23, “Using JMS” for more specific information. Setting up and deploying these requires the following steps:

1. If any external drivers are needed, which is usually the case, they must be deployed to the target Partition as a library archive. See *User's Guide,* Using Partitions chapter, for information on deploying to Partitions. Note that this step is not necessary if you are using the native all-Java database, JDataStore as your backend, or the JMS services broker.
2. Using the Console choose the predeployed JNDI Definitions Module in the Deployed Modules folder named `default-resources.dar` and right-click to launch the DDEditor. The predeployed module already has a datasource defined for JDataStore and sample JMS connection factories and destinations for the JMS services broker.
3. Add a new definition or edit an existing one in the module to suit your environment, then save the module. The changes will take effect on the server. Information on defining JDBC datasources and JMS connection factories is provided below.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

JNDI Definitions Module

J2EE resource connection factory objects are bound to JNDI when they are deployed as a part of a JNDI Definitions Module. This module is similar to other J2EE standard Java archive types, and ends in the extension `.dar`. This module is also referred to as a DAR, therefore. This module adds to the standard J2EE module types like JAR, WAR, and RAR. It can be packaged as a part of an EAR, or deployed stand-alone. Since datasources are contained in their own module(s), they appear in the Server tree of the Borland Enterprise Server's console, meaning you can easily enable and disable them by right-clicking their representations in the tree.

Note A DAR is *not* a part of the J2EE specification. It is a Borland-specific implementation designed to simplify the deployment and management of connection factories. You do *not* package connection factory classes in this archive type. Those classes must be deployed as a library to individual Partitions.

Since the Partition itself constructs the connection factory class, the only contents of the DAR that you must provide is an XML descriptor file called `jndi-definitions.xml`. Like other descriptors, this is placed within the `META-INF` directory of the DAR. The contents of the DAR hence look like the following:

```
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. Note that DAR deployment semantics are exactly like other J2EE modules. You can deploy any number of distinctly named DARs in the same Partition or to a cluster. Should two deployed DARs have objects with identical JNDI names, the last deployed module overwrites its object on the same node.

You can create a new DAR by simply providing a descriptor file in the directory structure above, JARing it into a DAR, and deploying it to a Partition using the server's tools. Or, you can run the DDEditor and select `File|New|JNDI Definitions Module` and allow the DDEditor to walk you through the creation procedure from scratch. Or, of course, you can edit the existing one.

Migrating to DARs from previous versions of Borland Enterprise Server

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 Borland Enterprise Server, 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:

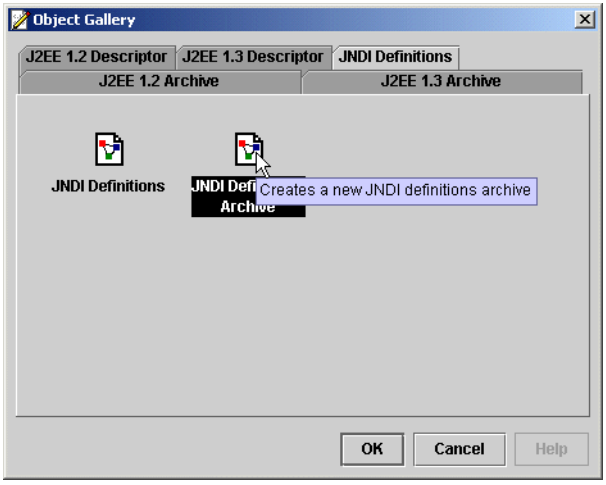
   ```
   prompt>jar uvMf default-resources.dar META-INF/jndi-definitions.xml
   ```

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 new JNDI Definitions Module

The DDEditor walks you through creating a new JNDI Definitions Module from scratch. Open the DDEditor and select "File|New..." The Object Gallery appears.



Select the JNDI Definitions tab and select JNDI Definitions Archive to create the new DAR. Click OK. You may now add JDBC datasources or add JMS resources, or you can 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 the datasource properties from the DAR, instantiates the datasource objects, and binds all of them into the JNDI service. To do this, open the Console and select "Wizards|Deployment Wizard." Follow the on-screen instructions. Detailed information on using this Wizard is found in the *User's Guide*.

Disabling and Enabling a JNDI Definitions Module

If you need to undeploy a set of datasources but still retain their definitions for future redeployment, right-click the module in the Management Console and choose "Disable." In addition, if you want to delete a set of deployed datasources permanently from your application, choose "Delete."

Packaging JNDI Definitions 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 defined in a JNDI Definitions Archive. Using the Assembly Tools in the Console it is easy to package the whole set of individual archives into a single EAR Module.

For example, if you have a set of EJBs that need access to a set of datasources then first create an EAR containing the EJB Jar archive and "Add -> JndiDefinitions to Application". Typically you would need to open up the EJB Jar in a DDEditor session to retarget the Resource References to point to the appropriate datasources defined in the JNDI Definitions.

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.

JNDI service provider for hosting resource factories

Plain Java objects like JDBC datasources and other J2EE resource connection factory objects need to be looked up from JNDI. Any JNDI provider that can store Java serializable and referenceable objects is supported, such as an LDAP directory external to the enterprise server. The Borland Enterprise Server's Naming Service provides a means to store the datasource objects into a Serial Context instead of the default CosNaming namespace supporting only CORBA references. The Serial Context URL begins with `serial://` and is

the default naming space for datasource objects. However, any plain Java object that implements both `java.io.Serializable` and `javax.naming.Referenceable` can be stored under this special JNDI URL context.

Configuring persistent storage locations for Serial Context

By default the Partition's Naming Service persistently stores JNDI objects in a particular default location on disk (`<install_dir>/var/domains/base/configurations/<configuration_name>/mos/<Partition_name>/adm/vbroker.properties/ns_serial_pstore`). Typically you do not need to alter any properties pertaining to the JNDI Serial Context back-end. You can however relocate the storage to any disk area you like by customizing this property:

```
vbroker.naming.serial.delegateURL
```

This is done by editing the Partition's `vbroker.properties` file, located in `<install_dir>/var/domains/base/configurations/<configuration_name>/mos/<Partition_name>/adm/vbroker.properties`.

Chapter

22

Using JDBC

Datasources are bound into a JNDI service after the JNDI Definitions Module is deployed. Datasource objects are stored into the Serial Context provider in the naming service. The portable J2EE mandated way to lookup a datasource is using a Resource Reference in the individual component's `ejb-jar.xml` descriptor file. Refer to the J2EE 1.3 Specification for information on how to use this element. This Resource Reference is itself a binding to the actual JNDI name in the Naming Service that is performed as a deployment time editing step.

In order to fully link the application component to its environment, however, the logical name for the resource must be resolved to a JNDI name. The Borland proprietary deployment descriptor, `ejb-borland.xml`, complements `ejb-jar.xml` and provides this mechanism in its own Resource Reference elements. The contents of this element varies depending upon the application component type and the resource to which it's trying to connect.

J2EE applications must lookup deployed datasources from the JNDI environment naming context, that is `java:comp/env/...` In order to access a resource from the JNDI environment naming context, you must first deploy a J2EE application component to an appropriate container with a declared reference to the required datasource. The procedure for doing so for JDBC datasources is described here. First, however, we'll explore how to configure the JDBC datasources to which your application components will connect.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

Configuring JDBC Datasources

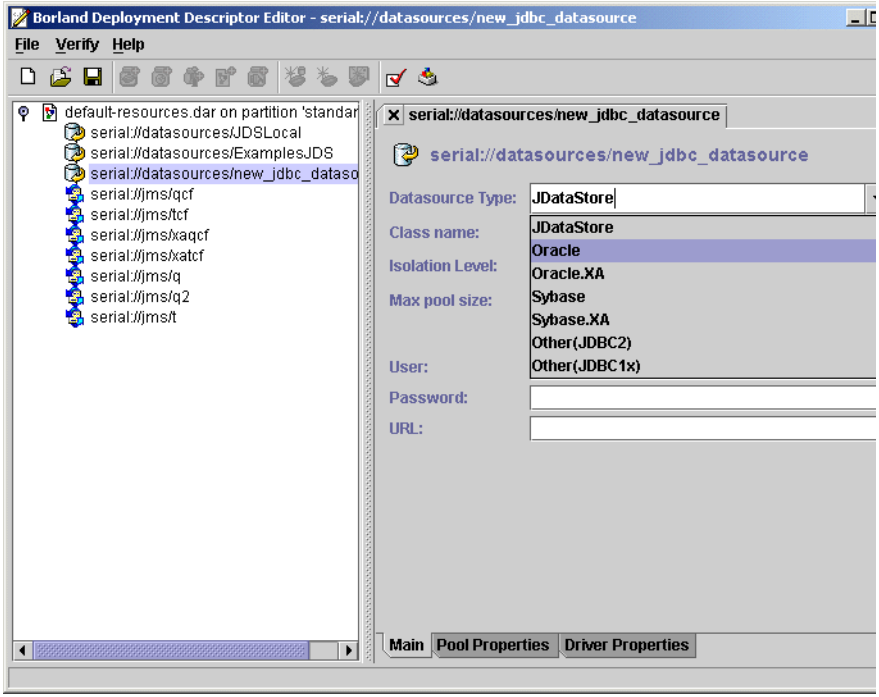
Using the Console, navigate to the "Deployed Modules" list in the Partition whose datasources you need to configure. By default, every Partition has a predeployed JNDI Definitions Module (DAR) called `default-resources.dar`. Right-click on the module and choose "Deployment Descriptor Editor" from the 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.

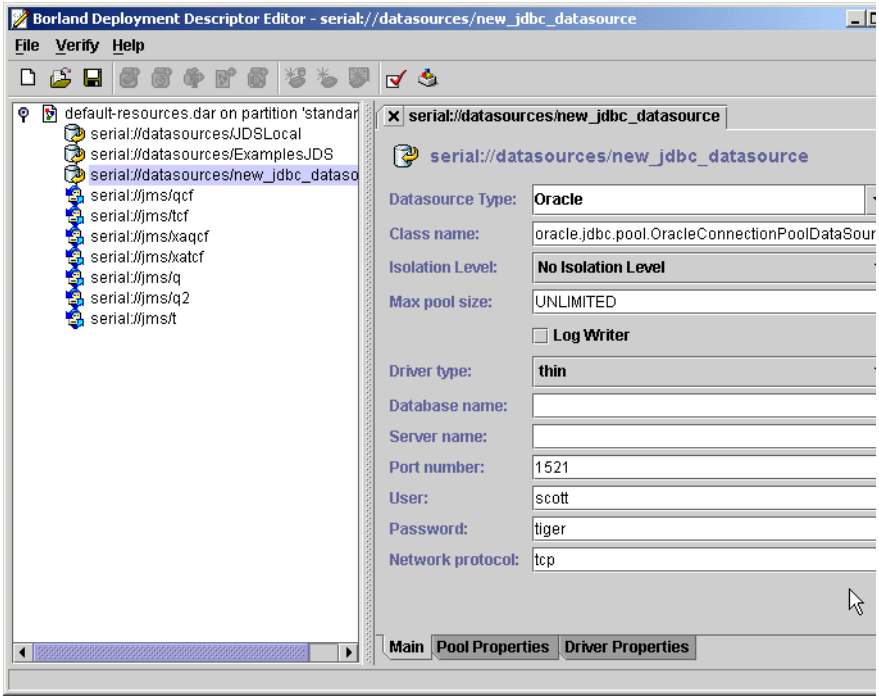
To create a new JDBC datasource, Right-click the `jndi-definitions.xml` node at the top of the tree and select "New JDBC Datasource..." from the Context Menu.

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.

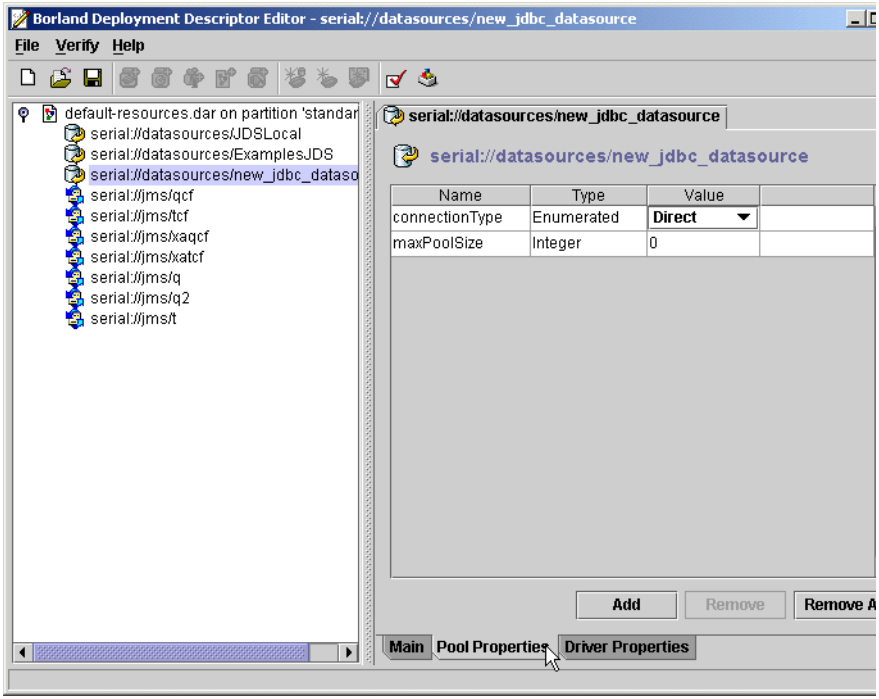
The DDEditor has knowledge of some common JDBC drivers and can autofill the class names and essential properties for the appropriate JDBC datasource. If the JDBC datasource type you want appears in the Datasource Type list then choose it, otherwise select "Other(JDBC2)".



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.

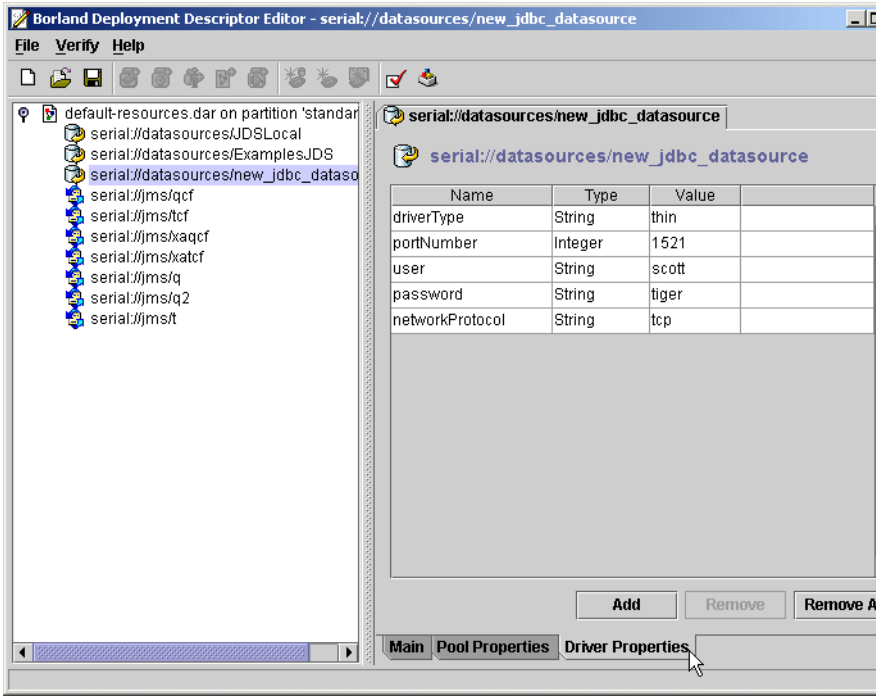


The Driver Properties and Pool Properties tabs capture some of the information from the Main tab, but also allow you to set any less common properties that do not appear in the Main tab.



Too 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

below in “Defining the Connection Pool Properties for a JDBC Datasource” on page 232. 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 any external drivers are needed, which is usually the case, they must be deployed to the target Partition as a library archive. See the User's Guide for information on deploying libraries to Partitions. Note that this step is 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. The steps are:

1. From the Console, select Tasks->Deployment->Deploy J2EE modules to a Partition.
2. "Add" the library file and check the option to restart the Partition.
3. Proceed to deploy in the chosen Partition(s.)
4. Close the wizard; the driver is listed under the "Deployed Modules" folder of the Partition once it restarts.

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 shown above. The complete list of properties, allowed values, defaults, and descriptions appear in the following table:

Name	Allowed Values	Description	Default Value	connectionType	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 calls are limited to optimize JDBC 2 connection pool performance. Setting optimizeXA to false disables this optimization. Under certain conditions the optimization must be disabled. For instance when conflicts arise with resource manager optimizations during two-phase commit protocol.	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)

Name	Allowed Values	Description	Default Value	connectionType	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
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	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.	isolationLevel	Enumerated: ■ TRANSACTION_NONE ■ TRANSACTION_READ_COMMITTED ■ TRANSACTION_READ_UNCOMMITTED ■ TRANSACTION_REPREATABLE_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.

Name	Allowed Values	Description	Default Value	connectionType	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
reuseStatements	Boolean	Optimization directive requesting prepared SQL statements to be cached for reuse. Applies to all connections obtained from the connection pool.	True	initSQL	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.
refreshFrequency	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)	dbPingSQL	String	Specifies a 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.

Name	Allowed Values	Description	Default Value	connectionType	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
resSharingScope	Enumerated: ■ Shareable ■ Unshareable	Indicates whether connection statements and result sets are cached for reuse -- Shareable -- thereby optimizing throughput of connections. If set to Unshareable, connections are closed once the application closes the connection.	Shareable	maxPreparedStatementCacheSize	Integer	Each connection within a BES 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	40

Name	Allowed Values	Description	Default Value	connectionType	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
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. Note that 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.

System Property Name	Type	Description	Default
`DataSourceDebug`	`Boolean`	Report activity at datasource level for all datasources	`False`
`ConnectionPoolDebug`	`Boolean`	Report activity at connection pool level for all datasources	`False`
`ConnetionPoolStateDebug`	`Boolean`	Reports transitions of connections in connection pool	`False`
`JDBCProxyDebug`	`Boolean`	Report activity at connection level for all connections	`False`
`PreparedStatementCacheDebug`	`Boolean`	Report activity at prepared statement level for all statements	`False`

Descriptions of Borland Enterprise Server'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. Here follows 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 the 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, Borland Enterprise Server 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” on page 232.

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>serial://datasources/Oracle</jndi-name>

 <driver-datasource-jndiname>serial://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>serial://datasources/Oracle</jndi-name>
 <driver-datasource-jndiname>serial://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>
 </visitranact-datasource>
```

Note the driver datasource JNDI name in bold. Now we'll add the following:

```
 <driver-datasource>
 <jndi-name>serial://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 Application Components

In `ejb-borland.xml`, the Resource Reference element is used to resolve logical names to JNDI names for JDBC datasources. You use the element within your individual component definitions. For example, a Resource Reference for an entity bean must be found within the `<entity>` tags. Let's look at the ejb-borland.xml representation of the Resource Reference element:

```
<!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 Reference 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 germane 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>serial://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 JNDI Serial Context service 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

23

Using JMS

JMS connection factories are bound into a JNDI service after the JNDI Definitions Module is deployed. Connection factory objects are stored into the Serial Context provider in the naming service. The portable J2EE mandated way to look up a datasource is using a Resource Reference in the individual component's `ejb-jar.xml` descriptor file. Refer to the J2EE 1.3 Specification for information on how to use this element. This Resource Reference is itself a binding to the actual JNDI name in the Naming Service that is performed as a deployment time editing step.

In order to fully link the application component to its environment, however, the logical name for the resource must be resolved to a JNDI name. The Borland proprietary deployment descriptor, `ejb-borland.xml`, complements `ejb-jar.xml` and provides this mechanism in its own Resource Reference elements. The contents of this element varies depending upon the application component type and the resource to which it's trying to connect.

J2EE applications must look up deployed connection factories from the JNDI environment naming context, that is `java:comp/env/...` To access a resource from the JNDI environment naming context, you must first deploy a J2EE application component to an appropriate container with a declared reference to the required datasource. The procedure for JMS resources is described here.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

Configuring JMS Connection Factories and Destinations

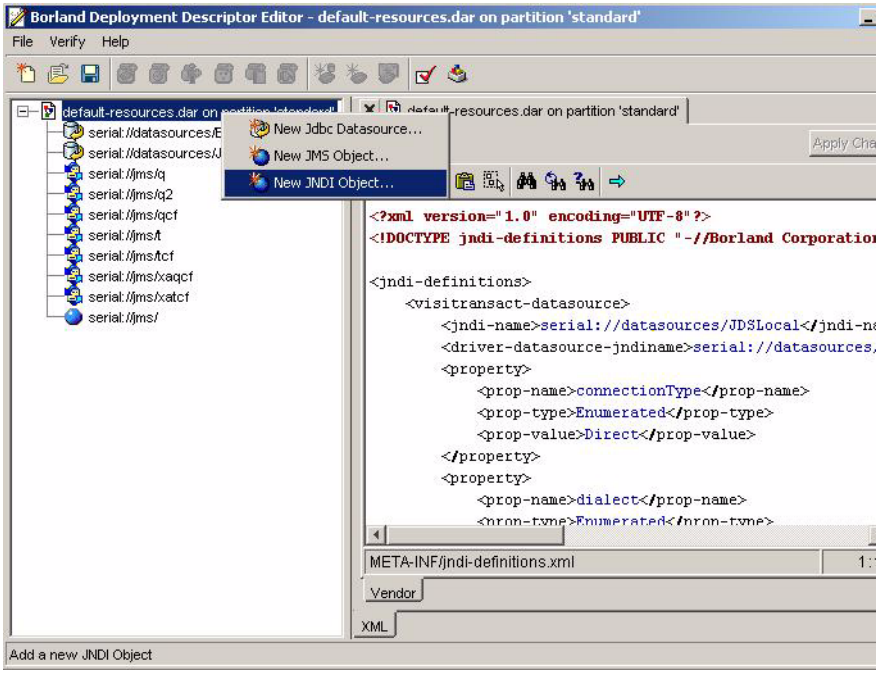
Using the Management Console, navigate to the "Deployed Modules" list in the Partition whose JMS connection factories you need to configure. By default, every Partition has a predeployed JNDI Definitions Module (DAR) called `default-resources.dar`. Right-click on the module and choose "Edit deployment 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. For each connection factory, you can choose Tibco, Sonic or another ("Other") JMS provider. The DDEditor has knowledge of Tibco and Sonic and auto fills the class names for each. You can also choose the object resource type from the JMS Object type drop-down list.

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.

Use the Properties tab to add any additional properties not provided on the Main tab.

To create a new JMS object, right-click either the `jndi-definitions.xml` or the `default-resources.dar` nodes and select "New JMS Object" from the Context Menu.



A dialog box prompts for a JNDI name for JMS object that is to be created. Once given, a representation of this JMS object appears in the tree in the Navigation Pane. Click its representation to open its configuration panel.

The DDEditor has knowledge of Tibco and Sonic, and can auto fill the class names for the appropriate JMS object.

Note The Main panel will not suggest any JMS objects other than Tibco or Sonic. You need to use the Properties tab to set any appropriate properties.

Use the Properties tab to set any needed properties.

When you're finished, choose "File|Save..." and the module will be saved back to the Partition and redeployed.

Queue creation

For specific information on queue creation, refer to Chapter 24, “JMS provider pluggability”.

Enabling Sonic

To enable Sonic:

1. Create a Sonic Managed Object. For information on creating Managed Objects, refer to the *BDOC Developer’s Guide,* Managed Objects section.
2. Modify `sonic.config` in `<install_dir>\bin\sonic.config` to include the Sonic home directory.

   The JMS home is the Sonic home directory, the root directory of the Sonic install, `JMS.home = <sonic home>`. The default is blank.

JMS and Transactions

The way to use JMS APIs in a transaction is detailed in the EJB 2.0 specification section 17.3.5.

```
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
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.
```

The user's code should never use any XA APIs in JMS. The program should look exactly as if the code is written like a normal JMS program and it is the Container's responsibility to handle any XA handshakes. The only thing you need to ensure is that you configured the <resource-ref> that points to the JMS Connection factory JNDI object to use the XA variant. If it is non-XA, the program still runs, but there are not any atomicity guarantees, in other words, it is a local transaction.

Also note that for the Container to automatically handle the transaction handshakes it is necessary to have the code run in a "container", either appclient, EJB or web. A simple java client won't show the correct characteristics, one has to write it as a J2EE application client instead. Also make sure that all connection factories are looked up via a <resource-ref>. This allows the Container to trap the JMS API calls and insert appropriate hooks.

Note We handle the following case:

```
// transaction context exists
conn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
...
sender.send( tm )
```

to mean the same as if the transaction context weren't there. We consider this to mean "give me what I asked for since the first parameter is explicitly false". This is not explicitly mentioned anywhere in the spec but this is in line with test cases in the J2EE Compatibility Test Suite (CTS 1.3.1). It seems useful to have this capability in a transaction you may want to send a log message irrespective of the enclosing transaction's commit/rollback.

Multi resource access is also supported. Here we look at providing the capability to do a unit of work which is composed of sending/receiving JMS messages along with some other resource access. That is, it is desirable to be able to write some code (an EJB for example) which does some work against a resource (Oracle, SAP, whatever) and also puts a message into a queue and the server provides transactional guarantees for that combination of work. That is, either the work against Oracle is done AND the message is queued, or something fails, and the work done against Oracle is rolled back AND the message is not queued.

In code, we support having a method like this:

```
// business method in a session bean, Container marks the transaction
     void doSomeWork()
     {
```

```
// execute JDBC work
   java.sql.Connection dbConn = datasource.getConnection();
     // execute SQL

// call some other EJBs in the same transaction
     // grab a reference to an EJB Remote interface
   ejbRemote.doWork();

// send a JMS message to a queue
   jmsSender.send(msg);

}
```

All the operations execute as a single unit of recoverable work (transaction) and leverages resource connection pooling (JDBC pool, JMS pool, and so forth).

Enabling the JMS services security

Refer to Chapter 24, “JMS provider pluggability” for vendor-specific information on JMS services such as security.

Advanced Concepts for Defining JMS Connection Factories

The predeployed module `default-resources.dar` contains some default Connection Factories, Topic and Queue, defined for the bundled JMS service. You can edit these existing definitions to suit your environment or create a new one. Again, like JDBC datasources, JMS connection factories 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 BES, you need to deploy that vendor's connection factory classes to your Partitions before you can use them.

Like JDBC Datasources, JMS Connection Factories are defined in the `jndi-definitions.xml` descriptor. The following DTD example shows the relevant element, in bold:

```
<!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)>
```

You only need to define the `<jndi-object>` element to register the JMS connection factory with JNDI. In this element, you specify the following:

- **jndi-name:** this is the name that will be looked up from JNDI to establish connections with the messaging service. Like JDBC datasources, JMS resources can and should use the `serial://` namespace whenever possible.
- **class-name:** this is the name of the connection factory class supplied by the JMS service provider and deployed as libraries in your Partition.
- **properties:** these are properties specific to the JMS provider that need to be passed to it.

At deployment time, the server creates the physical JMS destinations specified in the `jndi-definitions` list.

Refer to Chapter 24, “JMS provider pluggability” for vendor-specific information on JMS queues.

Connecting to JMS Connection Factories from Application Components

Connections to JMS Connection Factories are achieved in much the same way as those for JDBC. You declare your object references in the Resource Reference elements of both the standard and Borland-specific deployment descriptors. However, there are a few differences. First, you also must specify a Resource Environment Reference that points to a specific queue or topic contained within the Resource Reference datasource. Again, logical names are bound to actual JNDI names using the Borland-specific descriptor at deployment time.

Connecting to JMS Connection Factories from components other than MDBs

Use of Resource References and Resource Environment References in the `ejb-jar.xml` descriptor is described in the J2EE 1.3 Specification. In simple terms, the Resource Reference (`<resource-ref>` element) refers to a connection factory for the component to use to connect to the JMS service provider. The Resource Environment Reference (`<resource-env-ref>` element) points to the specific queue or topic to talk to. Let's look at an XML example for a session bean from `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>
```

Although this portable descriptor provides some logical names for the resources, they still need to bound to the actual JNDI name of the deployed datasource object. This is accomplished with the `ejb-borland.xml` descriptor, which binds the logical names with their JNDI locations. Like we did with JDBC datasources, we will use the `<resource-ref>` element of the `ejb-borland.xml` descriptor to do this:

```
<session>
 <ejb-name>session_bean</ejb-name>
 ...

 <resource-ref>
 <res-ref-name>jms/MyJMSQueueConnectionFactory</res-ref-name>
  <jndi-name>serial://resources/qfc</jndi-name>
 </resource-ref>
```

Note that the `<res-ref-name>` elements from both descriptor files are identical. Now, we need to bind the logical name for the queue (or topic, whatever the case may be) to its JNDI name. We accomplish this with the `<resource-env-ref>` element of the Borland-specific descriptor. It's DTD element looks like the following:

```
<!ELEMENT resource-env-ref (resource-env-ref-name, jndi-name)>
```

Like the Resource Reference element, we specify two things:

- **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>` in `ejb-jar.xml`.
- **jndi-name:** this is the JNDI name of the topic or queue that resolves the logical name.

So, in order to complete the connection entry, we add the Resource Environment Reference to the descriptor, yielding:

```
<session>
 ...
```

```
 <resource-ref>
  <res-ref-name>jms/MyJMSQueueConnectionFactory</res-ref-name>
  <jndi-name>serial://resources/qfc</jndi-name>
 </resource-ref>
 <resource-env-ref>
 <resource-env-ref-name>jms/MyJMSQueue</resource-env-ref-name>
  <jndi-name>serial://resources/q</jndi-name>
 </resource-env-ref>

 ...

</session>
```

Keep in mind that Resource References and Resource Environment References can be used for all types of objects that require connections to resources. Application clients can also use these references. You also need to provide a Borland-specific XML file, `application-client-borland.xml`, however. For example:

```
<application-client>

 ...

 <resource-ref>
  <res-ref-name>jms/MyJMSQueueConnectionFactory</res-ref-name>
  <jndi-name>serial://resources/qfc</jndi-name>
 </resource-ref>
 <resource-env-ref>
  <resource-env-ref-name>jms/MyJMSQueue</resource-env-ref-name>
  <jndi-name>serial://resources/q</jndi-name>
 </resource-env-ref>
 ...

</application-client>
```

Now, code from within the application-client archive can access these datasources as well. Bear in mind, however, that only EJB containers can host the DAR file itself. the Borland Client Container does not have this functionality.

Chapterb

24

JMS provider pluggability

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

BES is designed to support any arbitrary JMS provider as long as some requirements are met. There are three levels 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.

JMS provider clustering and security are also discussed in this section.

Note Tibco JMS is bundled with BES 6. Sonic 5 is certified, but not bundled with BES. Sonic must be acquired and installed separately, and the `sonic.config` file must be updated to let BES know where Sonic is installed. The location of this file is `<install_dir>\bin\sonic.config`. The JMS home is the Sonic home directory, the root directory of the Sonic install, for example, `jms.home = <sonic home>`.

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 BES 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. BES 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 admin 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 BES DDEditor tool can define, edit and deploy these objects into the BES JNDI tree without needing a JMS product-specific mechanism.

For specific information on using other JMS service providers with BES and requirements for admin objects (queues, topics, and connection factories), go to “Configuring admin objects for other JMS providers” on page 256.

Service management

The BES 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 Sonic) providers out of the box. Extending this to any JMS product is documented in “Service management for supported and other JMS providers” on page 258.

The BES Management Console allows for template-based additions of any number of JMS servers, of any vendor type, to the same BES configuration in a management domain.

The above list implicitly defines *levels* of pluggability. You will achieve the best scenario if all three are met, but just the runtime pluggability as well as vendor-specific ways to achieve the other levels may be sufficient in many situations.

Runtime pluggability

To achieve the level of "runtime pluggability", you need to comply with the J2EE specifications. A CTS compliant JMS product that additionally implements the JMS specification optional APIs can easily plug into the BES runtime. To support transactional messaging for MDBs and J2EE container intercepted messaging (connection and session pooling), meaning the queue or topic must be a transactional resource, BES requires that the JMS XA APIs support JTA integration. In addition, the JMS product should support the

`ConnectionConsumer` interface since a central idea of MDBs is the concurrent consumption of messages. The ConnectionConsumer achieves this, and works in conjunction with some optimal methods of the `javax.jms.Session` objects.

Tibco and Sonic

Tibco and Sonic have already achieved the runtime level of pluggability. Additionally, BES 6.0 passed CTS 1.3.1 on both JMS providers.

Other JMS providers

For other JMS providers to achieve a runtime level of pluggability, compliance with the J2EE specifications is necessary. BES requires that the JMS XA APIs support JTA integration. In addition, the JMS product should support the `ConnectionConsumer` interface. Refer to Sun's chapter on the JMS API for specific information on complying with the specification.

Configuring admin objects

The next level of pluggability, involves configuring admin ojects, like queues, topics and connection factories. The BES DDEditor tool can define, edit and deploy these objects into the BES JNDI tree without needing a JMS product-specific mechanism.

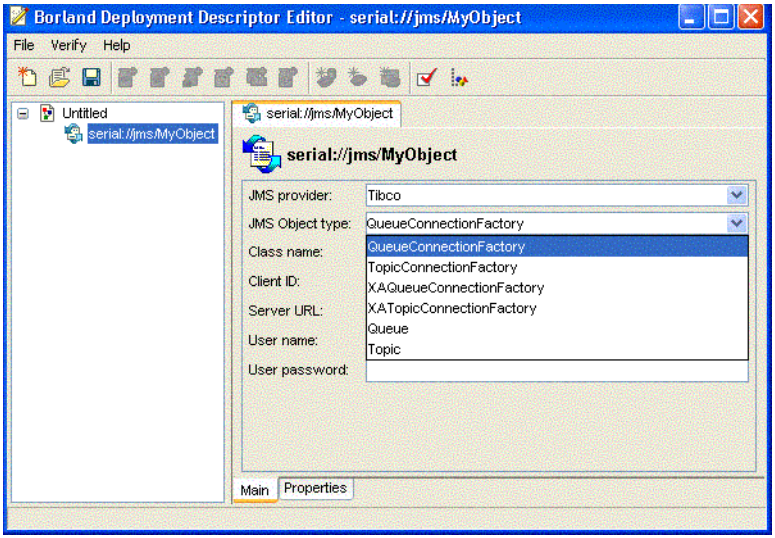
Tibco and Sonic

Both Tibco and Sonic achieve this level of pluggability. Their admin object properties are defined in BES and can be configured graphically, using the DDEditor.

To set the following admin object properties using the DDEditor:

1. Launch the DDEditor from within the Management Console or standalone from the Start menu.
2. Select New and JNDI Definitions Archive to create a new JMS object.
3. Right-click and select New JMS Object.
4. Give the JMS object a name in the New JMS Object window and click OK. Your JMS object will appear under your archive.
5. Click on your JMS object, and select the Main tab.
6. Configure your object by selecting various fields from the drop-down menus and entering information in the properties' fields.

7 To add additional properties for the JMS object, select the Properties tab and click "Add" to add properties (name, type, value).



Tibco Admin Console

BES includes the Tibco Admin Console for additional configuration. To launch the Tibco Admin Console, select it from the Tools menu in the BES Management Console.

Configuring admin objects for other JMS providers

BES provides configurations in the DDEditor for Tibco, Sonic and other JMS providers. However, there are required properties and classes for using other JMS providers with BES. These must be configured in the `jms.properties` file located in `C:\Documents and Settings\username\.bmc60`.

The following are the required properties that must be configured in the `jms.properties` file:

```
Tibco.QueueConnectionFactory.props=serverUrl,clientID,userName,userPassword
Tibco.XAQueueConnectionFactory.props=serverUrl,clientID,userName,userPasswo
rd
Tibco.TopicConnectionFactory.props=serverUrl,clientID,userName,userPassword
Tibco.XATopicConnectionFactory.props=serverUrl,clientID,userName,userPasswo
rd
Tibco.Queue.props=address
Tibco.Topic.props=address
Sonic.QueueConnectionFactory.props=connectionURLs,defaultUser,defaultPasswo
rd,connectID,clientID,sequential,loadBalancing
Sonic.TopicConnectionFactory.props=connectionURLs,defaultUser,defaultPasswo
rd,connectID,clientID,sequential,loadBalancing
Sonic.XAQueueConnectionFactory.props=connectionURLs,defaultUser,defaultPass
```

```
word,connectID,clientID,sequential,loadBalancing
Sonic.XATopicConnectionFactory.props=connectionURLs,defaultUser,defaultPass
word,connectID,clientID,sequential,loadBalancing
Sonic.Queue.props=queueName
Sonic.Topic.props=topicName
Other.QueueConnectionFactory.props=
Other.TopicConnectionFactory.props=
Other.XAQueueConnectionFactory.props=
Other.XATopicConnectionFactory.props=
Other.Queue.props=
Other.Topic.props=
Tibco.QueueConnectionFactory.class=com.tibco.tibjms.TibjmsQueueConnectionFa
ctory
Tibco.TopicConnectionFactory.class=com.tibco.tibjms.TibjmsTopicConnectionFa
ctory
Tibco.XAQueueConnectionFactory.class=com.tibco.tibjms.TibjmsXAQueueConnecti
onFactory
Tibco.XATopicConnectionFactory.class=com.tibco.tibjms.TibjmsXATopicConnecti
onFactory
Tibco.Queue.class=com.tibco.tibjms.TibjmsQueue
Tibco.Topic.class=com.tibco.tibjms.TibjmsTopic
Sonic.QueueConnectionFactory.class=progress.message.jclient.QueueConnection
Factory
Sonic.TopicConnectionFactory.class=progress.message.jclient.TopicConnection
Factory
Sonic.XAQueueConnectionFactory.class=progress.message.jclient.xa.XAQueueCon
nectionFactory
Sonic.XATopicConnectionFactory.class=progress.message.jclient.xa.XATopicCon
nectionFactory
Sonic.Queue.class=progress.message.jclient.Queue
Sonic.Topic.class=progress.message.jclient.Topic
Other.QueueConnectionFactory.class=
Other.TopicConnectionFactory.class=
Other.XAQueueConnectionFactory.class=
Other.XATopicConnectionFactory.class=
Other.Queue.class=
Other.Topic.class=
PropertyName=type, default
type=QueueConnectionFactory|TopicConnectionFactory|
XAQueueConnectionFactory|XATopicConnectionFactory|Queue|Topic
connectionURLs=String,localhost:2506
defaultUser=String
defaultPassword=String
connectID=String
clientID=String
sequential=Boolean,false
loadBalancing=Boolean,false
queueName=String
topicName=String
serverUrl=String,localhost:7222
userName=String
```

```
userPassword=String
address=String
```

Service management for supported and other JMS providers

Service management is handled by the BES service control infrastructure. JMS service processes like starting, stopping and server configuration are provided in the managed object configuration for supported (Tibco and Sonic) JMS providers out of the box. This level of pluggability for other JMS providers can be achieved with some configuration.

Other JMS providers

The following steps are required for introducing a JMS server other than Tibco or Sonic into BES:

1. If the main entry into the JMS server is through a Java class, and you are required to shut down the server using a management API, proceed to step 2 (to create a runner). Otherwise, go to step 3 (to create a launcher).
2. A runner works in conjunction with BES, a launcher to provide a graceful shutdown. Refer to the Sonic5Runner class as an example implementation of a runner. A runner is required to implement the two methods below:
   - `public static main(String[] argv)`: starting the server
   - `public static cleanup(boolean b)`: shutting down the server
3. A launcher is consisted of an executable and a config file. The executable files are all identical. In Windows, you can make a copy of `partition.exe` and give it a new name.
4. Make a config file with the same name as the executable created in step 3.
5. Populate the config file with the required runtime properties for the JMS server. The runtime properties are usually in a .bat file. Refer to `launcher.html` in `\JSoft\IAS\src\share\native` for more details. You can also refer to `sonicmq.config` (in <install_dir>\bin) for an example of runtime properties.
6. Create a config file similar to `tibco.config` or `sonic.config`, which are located in `<install_dir>\bin\`. Include all the client and admin JARs for the JMS server.
7. Modify the `jms.config` file, located in `<install_dir>\bin`, setting the JMS provider as the new one being added.
8. Create a Managed Object and include it in the `configuration.xml` file. If the new JMS server is a native server, use Tibco MO as an example. The Tibco MO is located in `<install_dir>\var\templates\managed_objects\jms`. If the new JMS server is an all Java server, use the Sonic MO as an example, which is located in the same file as Tibco MO.

9 If the JMS server is an all Java server, repeat steps 3 through 5 to create a launcher for the Admin Console.

Required libraries for other JMS providers

When trying to connect to another JMS provider the required libraries must first be deployed to the target Partition. The steps are:

1 Start your BES server and Management Console.

2 From the Console, open the Wizards menu and select the Deployment Wizard. The wizard allows you to deploy modules to a Partition.

3 "Add" the library file and check the option to restart the Partition on deploy.

4 Proceed to deploy to the chosen Partition(s). Close the wizard.

You should see your driver listed under the "Deployed Modules" folder of the Partition(s). The DDEditor panels do not provide as much custom help for other JMS vendor products as it does for JMS vendors certified to work with BES. Refer to the other JMS vendor's documentation to ascertain the correct name of the ConnectionFactory, Topic or Queue implementation class in order to create JNDI objects.

Note All required libraries are already configured so nothing extra is needed if you are going to use JMS services bundled with BES.

Added value for Tibco

Tibco provides this added value:

- Transparent installation
- No post installation configuration
- Clustering of Tibco servers can be done visually using the BES Management Console
- Tibco Admin Console is available from BES Management Console *Tools* menu.

Enabling Sonic

To enable Sonic:

1 Create a Sonic Managed Object. For information on creating Managed Objects, refer to the *BDOC Developer's Guide,* Managed Object section.

2 Modify `sonic.config` in `<install_dir>\bin\sonic.config` to include the Sonic home directory.

The JMS home is the Sonic home directory, the root directory of the Sonic install, `JMS.home = <sonic home>`. The default is blank.

Sonic

Using the DDEditor, you can configure a name (JMS service URL, sequential, loadBalancig) a type, and a value in the XML <jndi-object>, as shown in the example below:

```
<jndi-object>
        <jndi-name>serial://jms/message</jndi-name>
        <class-name>progress.message.jclient.QueueConnectionFactory</class-
name>
        <property>
            <prop-name>connectionURLS</prop-name>
            <prop-type>String</prop-type>
            <prop-value>localhost:2506</prop-value>
        </property>
        <property>
            <prop-name>sequential</prop-name>
            <prop-type>Boolean</prop-type>
            <prop-value>false</prop-value>
        </property>
        <property>
            <prop-name>loadBalancing</prop-name>
            <prop-type>Boolean</prop-type>
            <prop-value>true</prop-value>
        </property>
    </jndi-object>
```

Creating a clustered JMS service

Tibco

The Tibco JMS server works in pairs to provide fault tolerance. When working in pairs, one acts as the primary server and the other acts as a secondary server, with only the primary server accepting client connections and handling JMS messages. For the secondary server to take over active connections and properly handle persistent messages, both the servers must have access to a shared state. The primary and backup servers must both have access to a shared state. This shared state can be shared storage devices or other mechanisms such as replication.

The shared state must be accessible to both of the servers (primary and secondary). Some common ways to implement shared storage are Network Attached Storage (NAS) or Storage Area Network (SAN) devices. Replication schemes can also be used for replicating the shared state.

For more information on Tibco shared state and storage, refer to Tibco's documentation, located in `<install_dir>/jms/tibco/doc/html`.

The clustering of Tibco servers can be done visually using the BES Management Console. To cluster Tibco servers:

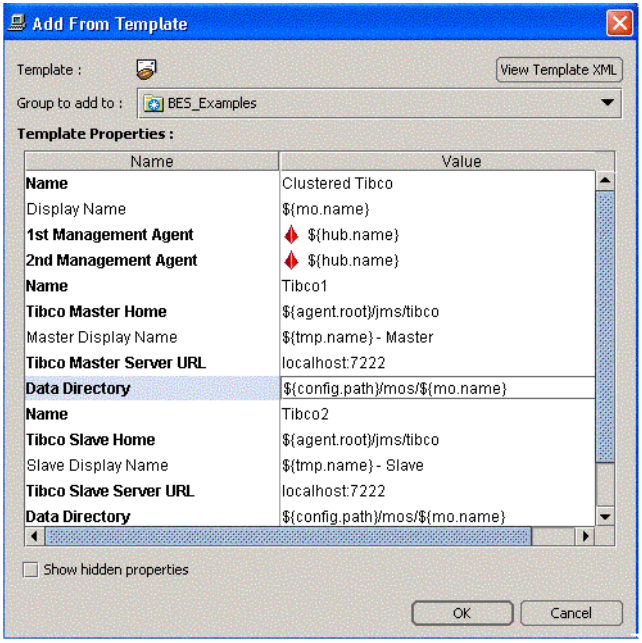
1 Create a shared directory that both of the Tibco servers have full access to.

2 Right-click the your Configuration and select Add Managed Object from Template/jms/Clustered Tibco.

3 Select a group to add clustered Tibco to from the drop-down menu.

4 Make the following changes to the template properties for the second Tibco server:

- `2nd Management Agent=hub.name` where `hub.name` is the name of your Management Hub.
- Name - this is the name of the second Tibco server.
- Tibco Slave URL - this is the port for the second Tibco server. The default is 7222. If the primary and secondary servers are on different machines the port numbers need to be different, they are the same if on the same machine.
- Data Directory - This is the path to the shared datastorage for both primary and secondary servers.



Integrating clustered Tibco servers into BES

The fault tolerant Tibco JMS server pair is represented in BES as a redundancy group Managed Object. Although there are two Tibco servers running, BES shows the pair of servers as a single Managed Object. The definition for the Redundancy Group Managed Object is shown below:

```
<managed-objects>
 <managed-object name="${tmp.name}"      agent="${hub.name}"display-
name="${tmp.display}" initial-manage="${tmp.managed}"
```

```
initial-monitor="true" type="redundancy-group">

 <redundancy-group member="2" desired-range-max="2">
   <member mo-ref="${tmp.agent1}/${tmp.master.name}"/>
   <member mo-ref="${tmp.agent2}/${tmp.slave.name}"/>
</redundancy-group>
</managed-object>
</managed-objects>
</configuration>
```

Note The Tibco servers in the example above are running on different hosts.

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>serial://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,hnguyen:7222</prop-value>
 </property>
</jndi-object>
<jndi-object>
 <jndi-name>serial://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,hnguyen:7222</prop-value>
 </property>
</jndi-object>
<jndi-object>
 <jndi-name>serial://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,hnguyen:7222</prop-value>
 </property>
</jndi-object>
 <jndi-name>serial://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,hnguyen:7222</prop-value>
 </property>
</jndi-object>
```

Sonic

Use the Sonic Wizard to create a Managed Sonic Messaging Service. For details on the specifics of your Sonic Messaging Service, refer to the Sonic documentation.

For clustering Sonic, you have two choices:

- If you have already used the Sonic Wizard to create a managed Sonic service, check the box and choose the appropriate instance from the drop down list.
- You can create a new managed Sonic service by clicking the Launch Sonic Wizard button. When you've finished, click Refresh and select your new Sonic service from the drop-down list.

Enabling security for JMS

Note For information on SSL, please refer to your JMS service provider's documentation. Tibco documentation is located in `<install_dir>\jms\tibco\doc\html`.

Tibco

To enable security for Tibco, you can either modify the `tibjmsd.conf` file located in: `/<install_dir>/var/domains/base/configurations/<configuration_name>/mos/tibco/tibjmsd.conf`, or you can set it using the Tibco Admin tool.

Enabling security for Tibco:

1. Open the Tibco Admin tool.
2. Type `connect`.
3. Enter Login name and Password.
4. Type `set server authorization=enabled`. You can also do this by modifying `tibjmsd.conf`, `authorization=enabled`.
5. Create a user, type `create user <name> [<description>] [password=<password>]`.
6. Add a member, type `add member <group-name> <user-name> [,<user-name2>,...]`.

Disabling security for Tibco:

1. Set the server authorization to disabled: `set server authorization=disabled`.

Sonic

To enable and disable security for Sonic, you must ensure `db.ini` file is updated before initialization of the database. There are two levels at which security must be applied, the backend database and the Sonic Broker. Use the Management Console to configure Sonic Broker and you can use a dbtool command to take care of the database. Most importantly, the database and Sonic broker must be in sync.

Enabling security for Sonic:

1. At the Sonic installation location, edit the `db.ini` file at the root of the installation and set the `ENABLE_SECURITY` property to `true` and save the modified file.
2. From the Management Console, connect to Sonic's domain.
3. On the Configure tab, right-click on the Sonic server that is being security enabled and choose Properties.
4. Select Enable Security.
5. Select an Authorization Domain.
6. Select an Authorization Policy.
7. Click OK to accept the changes.
8. On the Manage tab, navigate to the Sonic server in the container where it is deployed.
9. Right-click the container that hosts the Sonic server that is being security enabled and choose Operations > Shutdown.
10. From a command-line, issue `dbtool/ r b` from the Sonic `<install_dir>/bin`.
11. From the Sonic Explorer console Configure tab, expand Security then expand Default Authentication and select Users.
12. Right-click Users and add a new user.
13. On the Group Memberships tab, add the new user to all groups.

    To restart the container, stop and start Sonic from the BES Management Console.

Disabling security for Sonic:

1. Reconnect in the Sonic Explorer console. Use `username` "Administrator" and `password` "Administrator".
2. On the Configure tab, right-click the Sonic server that is being security enabled and select Properties.
3. Uncheck Security and choose OK to accept the changes.
4. At the Sonic installation location, edit the `db.ini` file at the root of the installation and set the parameter `ENABLE_SECURITY` to `false` and save the modified file.

5. On the Manage tab of the Sonic Explorer console, navigate to the Sonic server in the container where it is deployed.
6. Right-click on the container that hosts the Sonic server that is being security enabled and choose Operations > Shutdown.
7. From the command line issue `dbtool/ r b` from `<Sonic_install>/bin` directory.

Chapter

25

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.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

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

26

VisiConnect overview

Important For documentation updates, go to www.borland.com/techpubs/bes.

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) 1.3 Platform and the J2EE Connector Architecture (Connectors) 1.0 standards, this task has been greatly simplified.

VisiConnect, the Borland implementation of the Connectors 1.0 standard, provides a simplified environment for integrating various EISs with the Borland Enterprise Server. 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 Borland Enterprise Server 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 Borland Enterprise Server, each providing the integration implementation between the EIS and

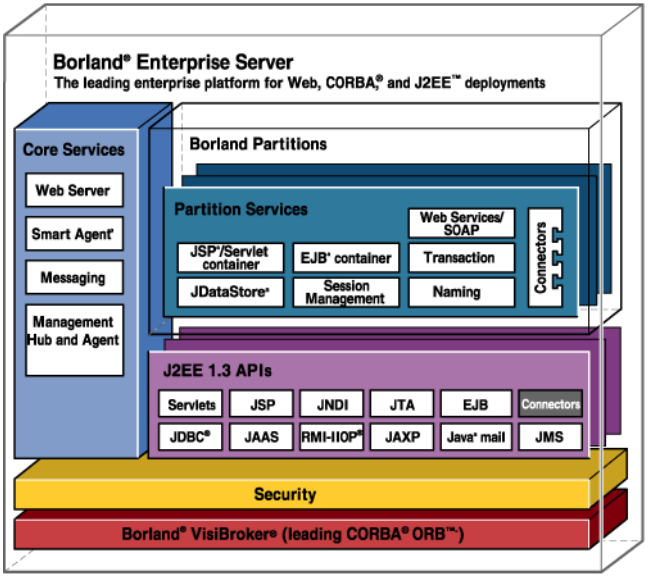
the Borland Enterprise Server. With VisiConnect, the Borland Enterprise Server ensures access to heterogeneous EISs. In turn, the EIS vendors need provide only one standard Connectors-compliant resource adapter. By default, this resource adapter has the capability to deploy to the Borland Enterprise Server.

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 1.3 Architecture, the Connectors is an extension of the J2EE Container, otherwise known as the application server. In compliance with the J2EE 1.3 Platform and Connectors 1.0 specifications, VisiConnect is an extension of the Borland Enterprise Server, and not a service in and of itself. The following diagram illustrates VisiConnect within the Borland Enterprise Server Architecture:

Figure 26.1 VisiConnect within the Borland Enterprise Server



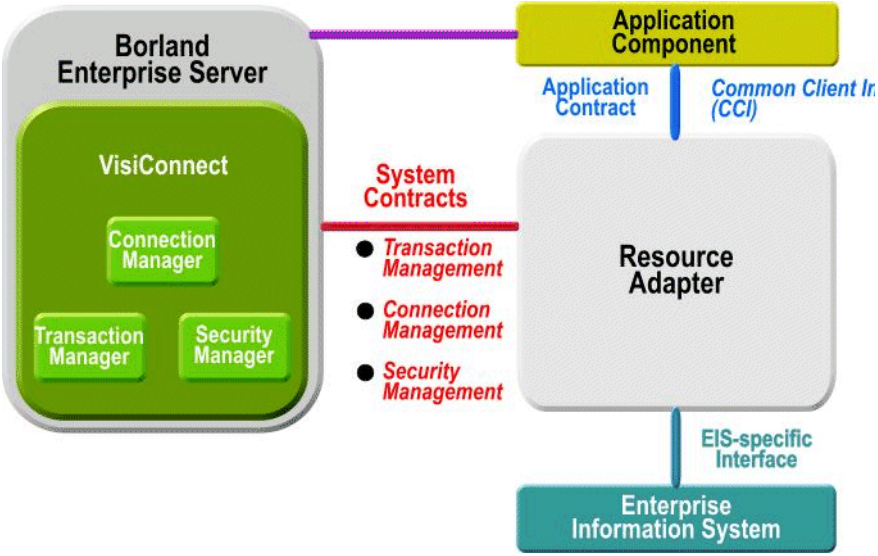
(VisiConnect is represented above by the module titled "**Connectors**.")

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 (Borland Enterprise Server).
- **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 1.0 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.

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 Borland Enterprise Server's transaction and security management.

The Borland Enterprise Server 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 Borland Enterprise Server'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, Borland Enterprise Server starts a local transaction based on the current transaction context. When the application closes the connection, Borland Enterprise Server 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 Borland Enterprise Server 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 1.0 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 1.0 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

Borland Enterprise Server user must be authenticated whenever they request access to a protected Borland Enterprise Server resource. For this reason, each user is required to provide a credential (a username/password pair or a digital certificate) to Borland Enterprise Server. The following types of authentication mechanisms are supported by Borland Enterprise Server:

- Password authentication a user ID and password are requested from the user and sent to Borland Enterprise Server 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 Borland Enterprise Server, the user ID and password of the user do not flow in the clear. Therefore, Borland Enterprise Server 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, Borland Enterprise Server 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 Borland Enterprise Server User owns the certificate and then retrieves the authenticated User from the Borland Enterprise Server 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 7.5 of the Connectors 1.0 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 Borland Enterprise Server 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 1.0 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, Borland Enterprise Server dynamically modifies the runtime environment for Resource Adapters. If the Resource Adapter has not defined specific security policies, Borland Enterprise Server overrides the runtime environment for the Resource Adapter with the default security policies specified in the Connectors 1.0 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 11.2, "Security Permissions", in the Connectors 1.0 specification (http://java.sun/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 1.0 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 1.3 Platform. Through the CCI, legacy EIS clients can integrate

directly with the Borland Enterprise Server; this provides for a smoother, less costly migration path to J2EE 1.3.

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 Borland Enterprise Server 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 1.0 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.

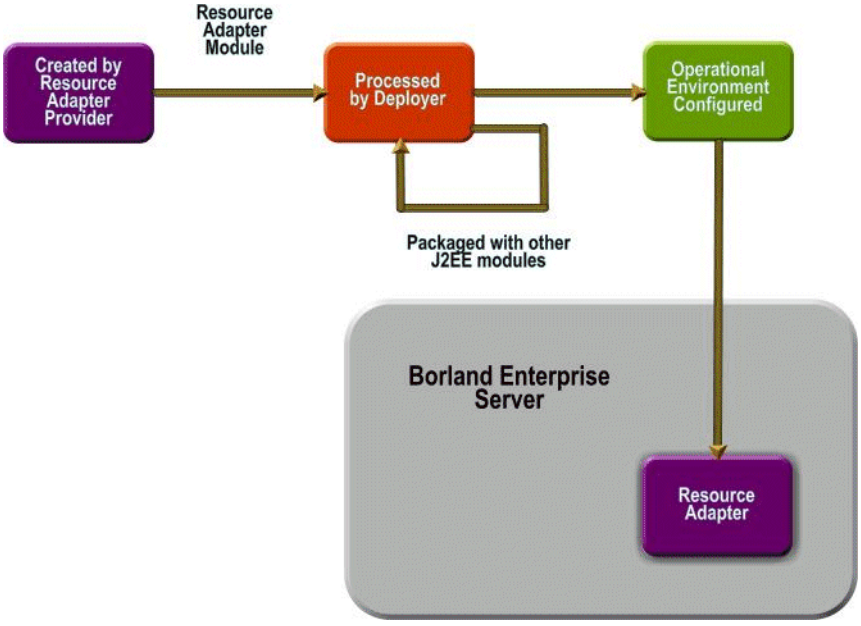
You must create records for all of the data structures that are externalized by the EIS functions you want to reuse. You then use the records as input and output objects that pass data via a Resource Adapter to and from an EIS. You will want to consider the following options when creating a record:

- **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 1.3 Platform compliant application servers, such as the Borland Enterprise Server.

Figure 26.2 Packaging and Deployment in the Borland Enterprise Server 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 Borland Enterprise Server 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 Container
- Local and Remote Connectors Support
- Additional Classloading Support
- Secure Password Credential Storage
- Connection Leak Detection
- Security Policy Processing of ra.xml Specifications

VisiConnect Container

The VisiConnect Container is designed to support development and deployment of J2EE applications which bundle Resource Adapters, or standalone Resource Adapter components. The Borland Enterprise Server provides integrated VisiConnect Container services. These services enable the creation and management of integrated VisiConnect Containers or VisiConnect Containers across multiple partitions. The Container is used to deploy, run, manage and monitor Resource Adapters. 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 Borland Enterprise Server provides a default VisiConnect Container for deployment. Additional Containers can be created as needed.. The VisiConnect Container can be run as a standalone process. For details, please refer to "Using VisiConnect".

Local and Remote Connectors Support

VisiConnect supports both the Local and Remote types of Connectors.

Local Connectors implement any interface, except java.rmi.Remote, in their connection factory. These are deployed to the serial jndi context handler of Borland Enterprise Server and are accessible from VisiConnect running within a Borland Enterprise Server or standalone, via VisiNaming, Borland's modular naming service. The deployment of Local Connectors is optimized for access by components running locally to VisiConnect.

Remote Connectors implement java.rmi.Remote in their connection factory, along with any other required interfaces. They are deployed to the CosNaming jndi context handler of Borland Enterprise Server, and are accessible from VisiConnect running within a Borland Enterprise Server or standalone, via VisiNaming. The deployment of Remote Connectors is optimized for access by component running remotely to VisiConnect.

Local Connectors are used with most J2EE Applications, while Remote Connectors are mainly used in the following cases:

- Migrating a non-J2EE application to J2EE, where it is required to interface the legacy application to Connectors as a migration step
- Running a non-J2EE application outside the aegis of an Application Server, where it is required to interface the application with Connectors
- Running CORBA clients/servers which are required to interface with Connectors
- Partitioning an application environment by host, where, for example, it is required to run the Connectors layer on a host remote to the Application layer.
- Other scenarios where it is required to access Connectors outside of J2EE, or remotely.

Note This list is by no means conclusive.

Most Resource Adapters available on the market are written as Local Connectors. However, the requirements for converting a Local Connector to a Remote Connector are minimal

- Packaging two classes in the Resource Adapter Archive (.rar) for the Local Connector, one which extends the Resource Adapter's connection factory interface to extend java.rmi.Remote, and one which extends the Resource Adapter's connection factory implementation to implement java.rmi.Remote.
- Modifying the Resource Adapter's deployment descriptors to reflect the new connection factory interface and connection factory implementation class.

VisiConnect bundles pre-built Remote extensions to javax.resource.cci.ConnectionFactory and javax.sql.DataSource for your convenience. To use these for converting a candidate Local Connector to a Remote Connector, use com.borland.enterprise.visiconnect.cci.ConnectionFactory or com.borland.enterprise.visiconnect.DataSource as your connection factory interface, extend the connection factory implementation to implement these interfaces, and use either of these implementations as your connection factory implementation class.

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 (Borland Enterprise Server roles, which may be associated with Borland Enterprise Server 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. Borland Enterprise Server also grants a Resource Adapter explicit permissions to access system resources.

Resource Adapters

Source code for six Resource Adapters are provided with VisiConnect as examples. These Resource Adapters are wrappers for JDBC 2.0 calls. The Local, Remote and Secured JDBC Connectors expose these calls directly via the JDBC 2.0 API. The Local, Remote and Secured CCI Connectors, exposes these calls indirectly via CCI. Deployment descriptors supporting the three transaction levels are provided for each Resource Adapter.

Simplified application examples for the three 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 Borland Enterprise Server.

Chapter

27

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 1.3 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 Enterprise Server, 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.

Important Before using VisiConnect, Borland recommends that you read the Connectors 1.0 specification.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

VisiConnect Container

The VisiConnect Container hosts Resource Adapters. Multiple Resource Adapters can be deployed in the same container. The container 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.

VisiConnect can have multiple partitions. Partitions are defined and configured as needed to deploy objects on the network. Custom partitions can be added for VisiConnect Containers for various data sources (such as IBM CICS) and support them separately from Resource Adapter development.

Container Overview

The VisiConnect Container is a complete implementation of the Connectors 1.0 specification, including all optional functionality.

Every Resource Adapter object in the deployed Connector is simultaneously both a Resource Adapter object and a CORBA object.

The VisiConnect Container can be deployed as a standalone, 100% pure Java service or as a fully-distributed deployment. This flexibility enables adjustment of the application's scalability and availability based on user requirements.

Unlike other Connectors implementations, the VisiConnect Container server has no restrictions on partitioning. Any number of Resource Adapters can go into any number of containers 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.

Container built on top of VisiBroker and RMI-IIOP

The VisiConnect Container is built on top of Borland VisiBroker. Communication between clients and Resource Adapters, among Resource Adapters, and between Resource Adapters and other CORBA-based applications is all done using IIOP by way of VisiBroker. VisiBroker is fully compliant with the CORBA 2.3 specification which specifies that RMI-over-IIOP must be implemented in terms of objects-by-value. RMI-over-IIOP must be implemented in terms of objects-by-value for true interoperability. As such, VisiConnect is interoperable with any other server supporting RMI-over-IIOP.

Security credentials are propagated by VisiBroker. This ensures that a client's credentials are propagated from the client to the server.

Transactional context is propagated by VisiBroker. This ensures that when a CORBA client begins a transaction and then accesses the VisiConnect Container's server, transactional context is propagated in the call to the server and the server uses this transaction context when making calls to various resources in its environment.

Two-phased commit transactions are managed by Borland VisiTransact Transaction Manager. Two-phase commit is supported if the Resource Adapter supports it. If the Resource Adapter in use does not support it, then two-phase commit cannot be done.

Container is a CORBA Server

Borland's java2iiop compiler, as well as the VisiConnect Container runtime, is CORBA compliant. The VisiConnect Container understands RMI method calls used for Resource Adapters but it uses IDL definitions internally to store the interface definitions. Although the java2iiop compiler takes the Java interfaces and generates stubs and skeletons from them, you can also generate IDL

from your Java interfaces for use in other languages. To a CORBA client, the VisiConnect Container is a CORBA server. The VisiConnect Container tools are CORBA tools that are equally capable of handling Resource Adapters.

The Borland VisiConnect Container is based on JNDI over CosNaming and JTS/OTS. Together, these provide complete support for CORBA.

Container as a partition service and standalone process

The VisiConnect container can run as a Partition service or as a standalone process.

For information on the VisiConnect container as a Partition service go to Chapter 3, “Using Partition services” in the "Management Console User's Guide".

Running a Standalone Container

If desired, additional versions of the VisiConnect container can be created and executed as a standalone process.

To run a standalone VisiConnect Container, execute the following command from a UNIX or Windows command line environment:

```
vbj com.borland.enterprise.visiconnect.ConnectorService visiconnect <RAR0
RAR1 RAR2 ...> -jns -jts
```

This will start up the standalone Connector Container named "visiconnect", loading the specified RAR(s), and starting the VisiNaming (-jns) and VisiTransact (-jts) sub-services.

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.

Configuring Connection Properties

VisiConnect allows you to set configuration properties through the use of the property element in the ra-borland.xml deployment descriptor file. To configure a set of configuration properties for a resource adapter, you specify a property-name and property-value pair for each configuration property to declare.

You can also use the property element to override the values specified in the ra.xml deployment descriptor file. At start-up, VisiConnect compares the values of property in ra-borland.xml against the values of config-property in

the ra.xml file. If the configuration property names match, VisiConnect uses the property-value for the corresponding configuration property name.

Minimizing the Runtime Performance Cost Associated with Creating Managed Connections

Creating Managed Connections can be expensive depending on the complexity of the Enterprise Information System (EIS) that the Managed Connection is representing. As a result, you may decide to populate the connection pool with an initial number of Managed Connections upon start-up of Borland Enterprise Server and therefore avoid creating them at run time. You can configure this setting using the initial-capacity element located in the ra-borland.xml descriptor file. The default value for this element is 1 Managed Connection.

As stated in the Connectors 1.0 specification, when an application component requests a connection to an EIS through the Resource Adapter, VisiConnect first tries to match the type of connection being requested with any existing and available Managed Connection in the connection pool. However, if a match is not found, a new Managed Connection may be created to satisfy the connection request.

VisiConnect provides a setting to allow a number of additional Managed Connections to be created automatically when a match is not found. This feature provides you with the flexibility to control connection pool growth over time and the performance hit on the server each time this growth occurs. You can configure this setting using the capacity delta element in the ra-borland.xml descriptor file. The default value is 1 Managed Connection.

Since no initiating security principal or request context information is known at VisiConnect start-up, the initial Managed Connections, configured with initial-capacity, are created with a default security context containing a default subject and a client request information of null. When additional Managed Connections configured with capacity-increment are created, the first Managed Connection is created with the known initiating principal and client request information of the connection request. The remaining Managed Connections up to the capacity-delta limit are created using the same default security context used when creating the initial Managed Connections.

Controlling Connection Pool Growth

As more Managed Connections are created over time, the amount of system resources such as memory and disk space that each Managed Connection consumes increases. Depending on the Enterprise Information System (EIS), this amount may affect the performance of the overall system. To control the effects of Managed Connections on system resources, Borland Enterprise Server allows you to configure a setting for the allowed maximum number of allocated Managed Connections.

You configure this setting using the maximum-capacity element in the ra-borland.xml descriptor file. If a new Managed Connection (or more than one Managed Connection in the case of capacity-delta being greater than one) needs to be created during a connection request, Borland Enterprise Server ensures that no more than the maximum number of allowed Managed Connections are created. If the maximum number is reached, Borland Enterprise Server attempts to recycle a Managed Connection from the connection pool. However, if there are no connections to recycle, a warning is logged indicating that the attempt to recycle failed and that the connection request can only be granted for the amount of connections up to the allowed maximum amount. The default value for maximum-capacity is 10 Managed Connections.

Controlling System Resource Usage

Although setting the maximum number of Managed Connections prevents the server from becoming overloaded by more allocated Managed Connections than it can handle, it does not control the efficient amount of system resources needed at any given time. Borland Enterprise Server provides a service that monitors the activity of Managed Connections in the connection pool during the deployment of a resource adapter. If the usage decreases and remains at this level over a period of time, the size of the connection pool is reduced to an efficient amount necessary to adequately satisfy ongoing connection requests.

This system resource usage service is turned on by default. However, to turn off this service, you can set the cleanup-enabled element in the ra-borland.xml descriptor file to false. Use the cleanup-delta element in the ra-borland.xml descriptor file to set the frequency with which Borland Enterprise Server calculates the need for connection pool size reduction, and if reduction is needed, selectively removes unused Managed Connections from the pool. The default value of this element is 15 minutes.

Detecting Connection Leaks

Connection leaks result from faulty application components, such as an Enterprise JavaBean (EJB), not doing their job to close a connection after using them. As stated in the Connectors 1.0 specification, once the application component has completed its use of the EIS connection, it sends a close connection request. At this point, VisiConnect is responsible for any necessary cleanup and making the connection available for a future connection request. However, if the application component fails to close the connection, the connection pool can be exhausted of its available connections, and future connection requests can therefore fail.

VisiConnect provides two mechanisms for preventing this scenario:

- Leveraging a garbage collector
- Providing an idle timer for tracking the usage of connection objects

Garbage Collection

VisiConnect automatically detects connection leaks by leveraging its Java Virtual Machine (JVM) garbage collector mechanism. When an application component terminates and the connections it uses become de-referenced, the garbage collector calls the connection object's finalize() method.

When the garbage collector calls the finalize() method, if VisiConnect determines the application component has not closed the connection, the server automatically closes the connection by calling the resource adapter's ManagedConnection.cleanup() method; VisiConnect behaves as it would had it received a CONNECTION_CLOSED event upon proper closure of the application component connection.

Idle Timer

Because the garbage collector does not behave in a predictable manner and may in fact never be called, VisiConnect provides a second connection leak detection method, the idle timer. The idle timer allows VisiConnect to track the last time each connection was used. You can configure the idle timer for each connection to an EIS using the Borland Enterprise Server Deployment Descriptor Editor. For more information, refer to "Using the Deployment Descriptor Editor" in the User's Guide.

When an application component obtains a connection for usage but is not actively using it, the idle timer starts ticking. As a precaution against closing a connection that is actually active, when a connection has reached its configured maximum limit, VisiConnect does not automatically close the connection. Instead, VisiConnect waits to close the connection that has exceeded its idle time until it is absolutely necessary to do so.

If the connection pool for a resource adapter has exceeded its maximum number of allocated connections and there are no allocated connections in the free pool, a connection request fails. At times, connections exist that have been leaked and not been put back on the free pool, even though they are inactive. In this scenario, VisiConnect closes connections that have exceeded their maximum idle time at the time of a connection request so that the request succeeds.

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 Borland Enterprise Server, 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 Borland Enterprise Server, 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>
```

and 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 Developer's 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 an <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 a Borland Enterprise Server 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 Borland Enterprise Server 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 Borland Enterprise Server. 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.
-usermame	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 1.0 specification, you must be able to deploy a Resource Archive (RAR) as part of an Enterprise Archive (EAR). With BES AppServer Edition and VisiConnect you can also deploy a standalone RAR. Once the RAR is deployed, you must:

- write code to obtain a connection
- create an Interaction object
- create an Interaction Spec
- create record and/or result set instances
- and 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 1.3 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 Borland Enterprise Server, 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 303, 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 Borland Enterprise Server. 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 1.0 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 Borland Enterprise Server-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 Borland Enterprise Server, 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 Borland Enterprise Server, 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 Borland Enterprise Server *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 Borland Enterprise Server.

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 Borland Enterprise Server 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 Borland Enterprise Server.

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 Borland Enterprise Server. 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 Borland Enterprise Server.

Until Borland-specific deployment properties are provided in the ra.borland.xml file, the RAR cannot be deployed to the server. The following attributes must be specified first in ra-borland.xml:

- Name of the connection factory
- Description of the connection factory
- JNDI name bound to the connection factory
- Reference to a separately deployed connection factory that contains Resource Adapter components which can then be shared with the current Resource Adapter.
- Directory where all shared libraries should be copied
- Connection pool parameters that set the following behaviors:
  - the initial number of managed connections the server attempts to allocate at deployment time
  - the maximum number of managed connections the server allows to be allocated at any one time
  - the number of managed connections the server attempts to allocate when fulfilling a request for a new connections
  - whether the server attempts to reclaim unused managed connections to save system resources
  - the time the server waits between attempts to reclaim unused managed connections
  - the frequency of time to detect and reclaim connections that have exceeded their usage time
  - the amount of usage time allowed for a connection
- Values for configuration properties defined in a <config-entry> element of the Sun standard Resource Adapter deployment descriptor
- 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.
- A flag to indicate whether logging is required for the `ManagedConnectionFactory` or `ManagedConnection` classes.
- The file to store logging information for the two aforementioned classes

Anatomy of ra-borland.xml

The Borland-specific deployment descriptor carries information for defining a deployable Resource Adapter Connection Factory. It provides for complete specification of all configurable connection factory parameters including connection pool parameters, security parameters, and the ability to define

values for configuration parameters which exist in the ra.xml deployment descriptor. A minimum set of this information is required for deployment. The following is an example of ra-borland.xml:

```
<connector>
    <connection-factory>
        <factory-name>shmeAdapterConnectionFactory</factory-name>
        <factory-description>SHME Resource Adapter Connection Factory</
factory-description>
      <jndi-name>serial://shme/shmeAdapterConnectionFactory</jndi-name>
        <ra-libraries>/usr/local/shme/Adapter/lib/shmeEisSdkUnix.so</ra-
libraries>
        <pool-parameters>
            <initial-capacity>0</initial-capacity>
            <maximum-capacity>10</maximum-capacity>
            <capacity-delta>1</capacity-delta>
            <cleanup-enabled>true</cleanup-enabled>
            <cleanup-interval>60</cleanup-interval>
        </pool-parameters>
        <security-map>
            <description>Map of billing staff users to the EIS operator
identity</description>
            <user-role>billing</user-role>
            <run-as>
            <description>SHME EIS application operator</description>
                <role-name>SHME_OPR</role-name>
            <run-as>
        </security-map>
    </connection-factory>
</connector>
```

Configuring the <ra-link-ref> element

The optional <ra-link-ref> element allows you to associate multiple deployed Resource Adapters with a single deployed Resource Adapter. This provides for linking and reusing resources already configured in a base Resource Adapter to another Resource Adapter, modifying only a subset of attributes. Use of the <ra-link-ref> element avoids duplication of resources (for example, classes, JARs, images, etc.) where possible. Any values defined in the base Resource Adapter deployment are inherited by the linked Resource Adapter unless otherwise specified.

When using <ra-link-ref>, perform one of the following:

- Assign the <maximum-capacity> element to zero (0) using the DDEditor. This allows the linked Resource Adapter to inherit its <pool-parameters> element values from the base Resource Adapter.
- Assign the <maximum-capacity> element to any value other than zero (0). The linked Resource Adapter will inherit no values from the base Resource Adapter. With this option, all of the <pool-parameters> element values for the linked Resource Adapter must be specified.

Configuring the Security Map

To use container-managed sign-on, the Borland Enterprise Server must identify a resource role and then request the connection to the EIS on behalf of the resource role. To make this identification, the server looks for a security map specified with the `<security-map>` element in the `ra-borland.xml` descriptor file. This security map builds the required associations between the server's user roles (Borland Enterprise Server users with identities defined in a security realm) and resource roles (users known to the Resource Adapter and/or EIS). See "Security Map" above for details on using the Security Map.

Developing the Resource Adapter

This section describes how to develop a Connectors 1.0-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 1.0 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 1.0 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 1.0 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 1.0 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 Borland Enterprise Server Console or the `iastool` utilities, or as a part of an EAR. See the Borland Enterprise Server *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. Borland Enterprise Server 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 Borland Enterprise Server until the module is undeployed.

The ra-borland.xml deployment descriptor DTD

This section provides a complete reference to the Borland Enterprise Server-specific XML deployment properties used in the VisiConnect Resource Adapter Archive and an explanation of how to edit the XML deployment descriptors.

Editing Descriptors

You may either edit the descriptor with the Borland DDEditor or manually using an ASCII text editor. Information on using the DDEditor to edit descriptors is provided in Chapter 5, "Using the Deployment Descriptor Editor," in the Borland Enterprise Server *User's Guide*.

If you choose to edit the descriptor manually, use an ASCII text editor which does not reformat the XML or insert addition characters that could invalidate the file. (Remember, this file must conform to the DTD--being well-formed is not sufficient.) You must be careful to use the correct case for file and directory names, *even if the host operating system ignores case*. You may set default values for an element either by omitting it entirely or leaving the value blank.

DOCTYPE Header Information

You must supply the correct `DOCTYPE` header for each descriptor you create. Using incorrect elements within the `DOCTYPE` header can result in parsing errors which may be difficult to diagnose.

The header refers to the location and version of the Document Type Definition (DTD) file of the deployment descriptor. Although this header references an external URL at java.sun.com, the Borland Enterprise Server contains its own local copy of the DTD file, so that the host server need not require Internet access. However, the `<!DOCTYPE ... >` element must still be included in the ra.xml file, and it must reference the external URL, as the version of the DTD defined in this element is used to identify the version of the deployment descriptor.

The following table shows the DOCTYPE headers for both ra.xml and ra-borland.xml:

Table 27.1 DOCTYPE headers

Descriptor File	DOCTYPE Header
ra.xml	<!DOCTYPE connector PUBLIC -//Sun Microsystems, Inc.//DTD Connector 1.0//EN 'http://java.sun.com/dtd/ connector_1_0.dtd>
ra-borland.xml	<!DOCTYPE ejb-jar PUBLIC"-//Borland Software Corporation//DTD Connector 1.0// EN" "http://www.borland.com/devsupport/ appserver/dtds/connector-_1_0-borland.dtd">

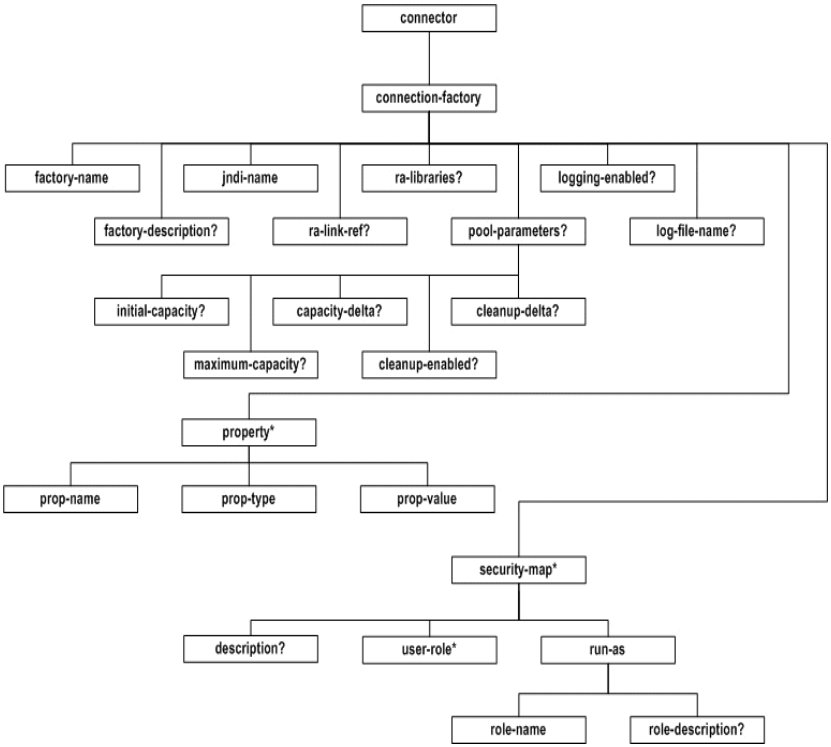
Deployment descriptors with incorrect header information may yield error messages similar to the following when used with a utility that parses the XML:

```
SAXException: This document may not have the identifier identifier-name
```

where *identifier-name* generally includes the text from the PUBLIC element.

Element Hierarchy

The element hierarchy of ra-borland.xml is as follows:



Code sample ra-borland.xml element hierarchy

```
<connector>
    <connection-factory>
        <factory-name></factory-name>
        <factory-description></factory-description>
        <jndi-name></jndi-name>
        <ra-link-ref></ra-link-ref>
        <ra-libraries></ra-libraries>
        <pool-parameters>
            <initial-capacity></initial-capacity>
            <maximum-capacity></maximum-capacity>
            <capacity-delta></capacity-delta>
            <cleanup-enabled></cleanup-enabled>
            <cleanup-interval></cleanup-interval>
        </pool-parameters>
        <logging-enabled></logging-enabled>
        <log-file-name></log-file-name>
        <property>
            <prop-name></prop-name>
```

```
            <prop-type></prop-type>
            <prop-value></prop-value>
        </property>
        <security-map>
            <description/>
            <user-role></user-role>
             and either
            <use-caller-identity/>
            or
            <run-as>
                <role-name></role-name>
                <role-description/>
            </run-as>
        </security-map>
    </connection-factory>
</connector>
```

The DTD

```
<!--
This DTD defines the Borland specific deployment information for defining
a deployable Resource Adapter Connection Factory. It provides for complete
specification of all configurable Connection Factory parameters including
Connection Pool parameters, Security parameters for Resource Role Mapping
and the ability to define values for configuration parameters which exist
in
the ra.xml deployment descriptor.
-->

<!ELEMENT connector (connection-factory)>

<!--
The connection-factory element is the root element of the
Borland specific deployment descriptor for the deployed
resource adapter.
-->

<!ELEMENT connection-factory (factory-name, factory-description?, jndi-
name, ra-link-ref?, ra-libraries?, pool-parameters?, (logging-enabled, log-
file-name)?, property*, security-map*, authorization-domain?)>

<!--
The factory-name element defines that logical name that
will be associated with this specific deployment of the
Resource Adapter and its corresponding Connection Factory.

The value of factory-name can be used in other deployed
Resource Adapters via the ra-link-ref element. This will
allow multiple deployed Connection Factories to utilize a
common deployed Resource Adapter, as well as share
configuration specifications.
```

```
This is a required element.
-->

<!ELEMENT factory-name (#PCDATA)>

<!--
The factory-description element is used to provide text
describing the parent element. The factory-description
element should include any information that the deployer
wants to describe about the deployed Connection Factory.

This is an optional element.
-->

<!ELEMENT factory-description (#PCDATA)>

<!--
The jndi-name element defines the name that will be used to bind the
Connection Factory Object into the JNDI Namespace. Client EJBs and
Servlets will use this same JNDI in their defined Reference Descriptor
elements of the Borland specific deployment descriptors.

This is a required element.
-->

<!ELEMENT jndi-name (#PCDATA)>

<!--
The ra-link-ref element allows for the logical association of
multiple deployed Connection Factories with a single deployed Resource
Adapter. The specification of the optional ra-link-ref element with
a value identifying a separately deployed Connection Factory will
result in this new deployed Connection Factory sharing the
Resource Adapter which had been deployed with the referenced
Connection Factory.

In addition, any values defined in the referred Connection Factories
deployment will be inherited by this newly deployed Connection Factory
unless specified.

This is an optional element.
-->

<!ELEMENT ra-link-ref (#PCDATA)>

<!--
The ra-libraries element identifies the directory location to be
used for all native libraries present in this resource adapter
deployment. As part of deployment processing, all encountered
native libraries will be copied to the location specified.

It is the responsibility of the Administrator to perform the necessary
platform actions such that these libraries will be found at runtime.
```

```
This is a required element IF native libraries are present.
-->

<!ELEMENT ra-libraries (#PCDATA)>

<!--
The pool-parameters element is the root element for providing Connection
Pool specific parameters for this Connection Factory.

VisiConnect will use these specifications in controlling the behavior
of the maintained pool of Managed Connections.

This is an optional element. Failure to specify this element or any
of its specific element items will result in default values being
assigned. Refer to the description of each individual element for
the designated default value.
-->

<!ELEMENT pool-parameters (initial-capacity?, maximum-capacity?, capacity-
delta?, cleanup-enabled?, cleanup-delta?)>

<!--
The initial-capacity element identifies the initial number of managed
connections which VisiConnect will attempt to obtain during deployment.

This is an optional element.

Failure to specify this value will result in VisiConnect using its
defined default value.

Default Value: 1
-->

<!ELEMENT initial-capacity (#PCDATA)>

<!--
The maximum-capacity element identifies the maximum number of
managed connections which VisiConnect will allow. Requests for newly
allocated managed connections beyond this limit will result in a
ResourceAllocationException being returned to the caller.

This is an optional element.

Failure to specify this value will result in VisiConnect using its
defined default value.

Default Value:  10
-->

<!ELEMENT maximum-capacity (#PCDATA)>

<!--
The capacity-delta element identifies the number of additional
managed connections which the VisiConnect will attempt to obtain
```

```
during resizing of the maintained connection pool.

This is an optional element.

Failure to specify this value will result in VisiConnect using its
defined default value.

Default Value: 1
-->

<!ELEMENT capacity-delta (#PCDATA)>

<!--
The cleanup-enabled element indicates whether or not the
Connection Pool should have unused Managed Connections reclaimed
as a means to control system resources.

This is an optional element.

Failure to specify this value will result in VisiConnect using its
defined default value.

Value Range: true|false

Default Value: true
-->

<!ELEMENT cleanup-enabled (#PCDATA)>

<!--
The cleanup-delta element identifies the amount of time the
Connection Pool Management will wait between attempts to reclaim
unused Managed Connections.

This is an optional element.

Failure to specify this value will result in VisiConnect using its
defined default value.

Default Value: 1
-->

<!ELEMENT cleanup-delta (#PCDATA)>

<!--
The logging-enabled element indicates whether or not the log writer
is set for either the ManagedConnectionFactory or ManagedConnection.
If this element is set to true, output generated from either the
ManagedConnectionFactory or ManagedConnection will be sent to the file
specified by the log-filename element.

This is an optional element.
```

```
Failure to specify this value will result in VisiConnect using its
defined default value.

Value Range: true|false

Default Value: false
-->

<!ELEMENT logging-enabled (#PCDATA)>

<!--
The log-file-name element specifies the name of the log file which output
generated from either the ManagedConnectionFactory or a ManagedConnection
are sent.

The full address of the file name is required.

This is an optional element.
-->

<!ELEMENT log-file-name (#PCDATA)>

<!--
Each property element identifies a configuration property
name, type and value that corresponds to an ra.xml entry
element with the corresponding property-name.

At deployment time, all values present in a property
specification will be set on the ManagedConnectionFactory.

Values specified via a property will supersede any default
value that may have been specified in the corresponding ra.xml
config-property element.

This is an optional element.
-->

<!ELEMENT property (prop-name, prop-type, prop-value)>
<!ELEMENT prop-name (#PCDATA)>
<!ELEMENT prop-type (#PCDATA)>
<!ELEMENT prop-value (#PCDATA)>

<!--
The security-map element specifies whwhether the caller's security identity
is to be used for the execution of the methods of the enterprise bean
or whether a specific run-as identity is to be used. It contains an
optional description and a specification of the security identity to
be used.

Each security-map element provides a mechanism to define appropriate
Resource Role values for Resource Adapter/EIS authorization processing,
through the use of the run-as element.
```

```
This element allows for the specification of a defined set of
user roles and the corresponding run-as roles (representing
EIS identities) that should be used when allocating
Managed Connections and Connection Handles.

A default Resource run-as role can be defined for the Connection
Factory via the map. By specifying a user-role value of '*'
and a corresponding run-as role, the defined run-as will be utilized
whenever the current role is NOT matched elsewhere in
the map.

This is an optional element, however, it must be specified in some
form if Container Managed Sign-on is supported by the Resource Adapter
and used by ANY client.

In addition, the deployment-time population of the Connection Pool
with Managed Connections will be attempted using the defined
'default' run-as if one is specified.
-->

<!ELEMENT security-map (description?, user-role+, (use-caller-identity|run-
as))>

<!--
The user-role element contains one or more role names, defined for
use as the security identity, or mapped to a appropriate Resource
Role run-as identity, for interactions with the resource.
-->
<!ELEMENT user-role (#PCDATA)>

<!--
The use-caller-identity element specifies that the caller's security
identity be used as the security identity for the execution of the
Resource Adapter's methods.

Used in: security-map
-->
<!ELEMENT use-caller-identity EMPTY>

<!--
The run-as element specifies the run-as identity to be used for the
execution of the enterprise bebean. Itontains an optional description, and
the name of a security role.

Used in: security-map
-->
<!ELEMENT run-as (description?, role-name)>

<!--
The role-name element contains the name of a security role.

The name must conform to the lexical rules for an NMTOKEN.

Used in: run-as
```

```
-->

<!ELEMENT role-name (#PCDATA)>

<!--
The authorization-domain element specifies the authorization domain to
be used for determining the definable set of valid user roles.
-->

<!ELEMENT authorization-domain (#PCDATA)>

<!ELEMENT description (#PCDATA)>
```

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 1.0 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 1.0 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 1.0 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. Borland Enterprise Server 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 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>serial://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>serial://eis/shmeAdapter</jndi-name>
                <cmp-resource>False</cmp-resource>
            </resource-ref>
        </session>
    </enterprise-beans>
</inprise-specific>
```

Other Considerations

Converting a Local Connector to a Remote Connector

VisiConnect is the sole Connector Container which supports RemoteConnectors. The issue of Remote Connectors is presently outside the Connectors specifications. As a result, most Resource Adapters available on the market are implemented as Local Connectors. Therefore, if you have a requirement to use a Remote Connector, such as:

- Migrating a non-J2EE application to J2EE, where it is required to interface the legacy application to Connectors as a migration step,
- Running a non-J2EE application outside the aegis of an Application Server, where it is required to interface the application with Connectors,
- Running CORBA clients/servers which are required to interface with Connectors,
- Partitioning an application environment by host, where, for example, it is required to run the Connectors layer on a host remote to the Application layer, or
- Working with some other requirements where it is required to access Connectors outside of J2EE, or remotely, and you do not have a Remote Connector for the desired EIS, you will need to convert the available Local Connector to a Remote Connector. Fortunately, this process is very straightforward and quick to accomplish.

Conversion

The main steps for converting a Local Connector into a Remote Connector are:

1. Extend the connection factory class of the Local Connector, and have the extension:
   - Implement `java.rmi.Remote`
   - Throw `java.rmi.RemoteException` from each method VisiConnect bundles ready extensions to `javax.sql.DataSource` and `javax.resource.cci.ConnectionFactory`: `com.borland.enterprise.visiconnect.DataSource` extends `javax.sql.DataSource` to `maremoteteable`
   - `com.borland.enterprise.visiconnect.cci.ConnectionFactory` extends `javax.resource.cci.ConnectionFactory`to make it remote.
2. The VisiConnect Remote Connector Example illustrates how to extend the connection factory in this fashion, for both CCI and non-CCI cases. This is available at `$installRoot/examples/j2ee/visiremote`. As such, if you wish to make a Resource Adapter with a connection factory implementing `javax.sql.DataSource` or `javax.resource.cci.ConnectionFactory` a Remote Connector, all you need to do is to use the

`com.borland.enterprise.visiconnect.DataSource` or the `com.borland.enterprise.visiconnect.cci.ConnectionFactory` interfaces.

3 Update the classes in the `ra.xml` standard deployment descriptor file. For example, before extending the interfaces, the `ra.xml` may look something like this:

```
   <managedconnectionfactory-
class>com.borland.enterprise.connector.serial.
LocalTxManagedConnectionFactory</managedconnectionfactory-class>
   <connectionfactory-interface>javax.sql.DataSource</connectionfactory-
interface>
   <connectionfactory-impl-
class>com.borland.enterprise.connector.serial.JdbcDataSource</
connectionfactory-impl-class>
   <connection-interface>java.sql.Connection</connection-interface>
   <connection-impl-
class>com.borland.enterprise.connector.serial.JdbcConnection</
connection-impl-class>
```

4 After extending the interfaces, the `ra.xml` may look something like this:

```
   <managedconnectionfactory-
class>com.borland.enterprise.connector.serial.LocalTxManagedConnectionFa
ctory</managedconnectionfactory-class>
   <connectionfactory-
interface>com.borland.enterprise.visiconnect.DataSource</
connectionfactory-interface>
   <connectionfactory-impl-
class>com.borland.enterprise.connector.serial.RemoteJdbcDataSource</
connectionfactory-impl-class>
   <connection-interface>java.sql.Connection</connection-interface>
   <connection-impl-
class>com.borland.enterprise.connector.serial.JdbcConnection</
connection-impl-class>
```

As this illustrates, this conversion impacts the connection factory only. No other Resource Adapter classes are affected by this conversion.

1 Compile the Java code for the extended interface and implementation into class files.

2 Package these into the Resource Adapter's Java Archive (.jar) file.

3 Update the Resource Adapter Archive (.rar) file with this extended .jar.

4 Deploy the Resource Adapter Archive, or include it in a 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 Borland Enterprise Server.

You now have a Remote Connector. The Resource Adapter's connection factory will be registered in JNDI via CosNaming. You can browse this using the JNDI Browser in the Console. Any CORBA client can now use this Connector. The client can be written in any language which has an IDL

mapping, which includes C, C++, Delphi (Object Pascal), Ada, COBOL, COBOL Scripting Language, Lisp, PL/1, Python, and Smalltalk.

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 1.3 and Connectors 1.0 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 1.0 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 Borland Enterprise Server 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:

1. Returns a valid, non-null reference of the `Referenceable` object, or
2. 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` implemenation 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 Local 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 objecfall back 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 Borland Enterprise Server.

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

28

Apache Ant and running BES examples

Many of the BES examples now employ the Ant build script system. In addition to Ant's core functionality, the BES version of Ant includes customized tasks for several of the BES command line tools, including commands of the following:

- `appclient`
- `iastool`
- `java2iiop`
- `idl2java`

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.

Syntax and general usage

The following table shows the currently defined tasks and their relationship to the equivalent commands.

Ant Task Name	Equivalent Command Line	Function	Fileset Attribute
appclient	appclient	Runs a client application.	
iascompilejsp	iastool -compilejsp	Precompiles JSP's.	
iasdeploy	iastool -deploy	Deploys a J2EE module.	jars
iasgendeployable	iastool -gendeployable	Generates a manually deployable module.	
iasgenclient	iastool -genclient	Generates a client library.	jars
iasgenstubs	iastool -genstubs	Generate a stub library.	
iasmerge	iastool -merge	Merges a set of JAR files into a single JAR file.	jars
iaskill	iastool -kill	Kills a Managed Object.	
iasrestart	iastool -restart	Restarts a Managed Object.	
iasstop	iastool -stop	Stops a Managed Object.	
iasundeploy	iastool -undeploy	Undeploys a managed object.	
java2iiop	java2iiop	Executes the java2iiop command.	
idl2java	idl2java	Converts IDL to Java classes.	

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 “Multiple File Arguments” on page 336.

Translating BES commands into Ant tasks

Basic Syntax

The options for a command-line tool translated into equivalent XML attributes. For example, the command line:

```
iastool -verify -src cart_beans_client.jar -role DEVELOPER -warn -nostrict
```

translates into the Ant task:

```
<iasverify src="cart_beans_client.jar" role="DEPLOYER" warn="true"
strict="false" />
```

For boolean-style attributes, use the base name of the attribute. 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" warn="false"
strict="false"
```

Note It is **not** valid to use "nowarn" as an attribute. For instance, the following line causes a syntax error:

```
****** INCORRECT SYNTAX!!! ******
<iasverify src="cart_beans_client.jar" role="DEPLOYER" nowarn="false"
strict="false" />
```

Omitting attributes

Omitting an attribute from the Ant task call has the same effect as omitting the option from the command line. Since some attributes are `true` by default, omitting an attribute does **not** necessarily set the attribute to `false`. For example, the syntax for the `iastool -verify` command is:

```
iastool -verify -src srcjar -role <DEVELOPER|ASSEMBLER|DEPLOYER> [-nowarn]
[-strict] [-classpath <classpath>]
```

As the `-nowarn` and `-strict` options are the default command options, the following commands are equivalent:

```
iastool -verify -src cart_beans_client.jar -role DEVELOPER
```

```
iastool -verify -src cart_beans_client.jar -role DEVELOPER -nowarn -strict
```

The equivalent Ant calls are:

```
<iasverify src="cart_beans_client.jar" role="DEPLOYER" />
```

```
<iasverify src="cart_beans_client.jar" role="DEPLOYER" warn="false"
strict="true" />
```

For more information on the default values of these options, go to Chapter 29, “iastool command-line utility”.

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:

```
<iasmerge 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:

```
<iasmerge target="build/client.jar" type="lib" >
<fileset dir="client/build" includes="local_client.jar" />
<fileset dir="build" includes="local_stubs.jar" />
</iasmerge>
```

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:

```
<iasmerge target="${build.dir}/client.jar" type="lib" >
<fileset dir="${build.dir}" includes="**/*.jar" />
</iasmerge>
```

Class path attributes can include multiple paths separated by semicolons:

```
<iasverify src="cart_beans_client.jar" role="DEPLOYER"
classpath="alib.jar;blib.jar" />
```

or use the `<classpath>` element:

```
<iasverfify src="cart_beans_client.jar" role="DEPLOYER" >
<classpath>
<pathelement location="alib.jar" />
<pathelement location="blib.jar" />
</classpath>
```

Building the example

Note Many of the BES examples have their own `readme.html` files located in:

```
<install_dir>/examples
```

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 partion 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 `<bes_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 `<bes_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 29

iastool command-line utility

This section describes the `iastool` command-line utility that you can use to manage your managed objects.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

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 29.1 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 341.
`-compress`	Compress a JAR file. For more information, see "compress" on page 342
`-deploy`	Deploy a J2EE module to the specified Partition. For more information, see "deploy" on page 343.
`-dumpstack`	Dump the stack trace of a Partition process to the partition event log file. For more information, see "dumpstack" on page 345.

Table 29.1 iastool command-line utilities

Use...	To...
`-genclient`	Generate a library containing client stubs, EJB interfaces, and dependent classes. For more information, see “genclient” on page 346.
`-gendeployable`	Generate a manually deployable module. For more information, see “gendeployable” on page 347.
`-genstubs`	Generate a library containing client or server stubs only. For more information, see “genstubs” on page 348.
`-info`	Display system configuration information. For more information, see “info” on page 349.
`-kill`	Kill a managed object. For more information, see “kill” on page 350.
`-listpartitions`	List the partitions on a hub. For more information, see “listpartitions” on page 352.
`-listhubs`	List the available hubs on a management port. For more information, see “listhubs” on page 353.
`-listservices`	List the services on a hub. For more information, see “listservices” on page 354.
`-merge`	Merge a set of JAR files into a single JAR file. For more information, see “merge” on page 355.
`-migrate`	Migrate a module from J2EE 1.2 to J2EE 1.3. For more information, see “migrate” on page 357.
`-patch`	Apply one or more patches to a JAR file. For more information, see “patch” on page 357.
`-ping`	Ping a managed object or hub for its current state. For more information, see “ping” on page 358.
`-pservice`	Enables, disables, or gets the state of a partition service. For more information, see “pservice” on page 360.
`-removestubs`	Remove all stub files from a JAR file. For more information, see “removestubs” on page 362.
`-restart`	Restart a hub or managed object. For more information, see “restart” on page 362.
`-setmain`	Set the main class of a standalone Client Jar or a Client Jar in an EAR. For more information, see “setmain” on page 364.

Table 29.1 iastool command-line utilities

Use...	To...
`-start`	Start a managed object. For more information, see "start" on page 365.
`-stop`	Stop a hub or managed object. For more information, see "stop" on page 366.
`-uncompress`	Uncompress a JAR file. For more information, see "uncompress" on page 368.
`-undeploy`	Remove a J2EE module from a Partition. For more information, see "undeploy" on page 369.
`-usage`	Display the usage of command-line options. For more information, see "usage" on page 370.
`-verify`	Verify a J2EE module. For more information, see "verify" on page 370.

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.

Syntax

```
-compilejsp -src <war_or_ear> -target <target_file> [-package
<package_root>]
[-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 file to be generated. The file name you specify cannot already exist. The full or relative path to the file must be specified. There is no default.

Option	Description
`-package <package_root>`	Specifies the base package name for the precompiled JSP servlet classes. The default is `com.bes.compiledjsp`.
`-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
```

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 the *VisiBroker for Java Developer's Guide* Programmer tools for Java section.
`-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 372 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 event log file. 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

Option	Description
`-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 372 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 the *VisiBroker for Java Developer's Guide,* Programmer tools for Java section.
`-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 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 the *VisiBroker For Java Developer's Guide,* Programmer tools for Java section.
`-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 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 the *VisiBroker for Java Developer's Guide,* Programmer tools for Java section.
`-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\BES
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\BES\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 372 for more information.

Examples

The following example kills the managed object `j2ee-server` using the default management port:

```
iastool -kill -hub BES1 -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 BES1 -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 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.

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 372 for more information.

Examples

The following example lists the partitions running on the hub BES1 using the default management port:

```
iastool -listpartitions -hub BES1
```

The following example lists the partitions running on the hub BES1 using the management port 24410:

```
iastool -listpartitions -hub BES1 -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.

Option	Description
-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 372 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 372 for more information.

Example

The following example lists all services running on the `salsa` hub:

```
iastool -listservices -hub salsa
```

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 J2EE version 1.2 to J2EE version 1.3.

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 -src <srcjar> -target <targetjar>
```

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
`-src <srcjar>`	Specifies the J2EE version 1.2 file to convert. The full or relative path to the JAR file must be specified. There is no default.
`-target <targetjar>`	Specifies the name of the J2EE version 1.3 file to be created. The full or relative path to the JAR 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_3.jar
```

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.

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 372 for more information.

Examples

The following example pings the hub BES1 in the default management port:

```
iastool -ping -hub BES1
```

The following example pings the partition naming service running on the hub BES1 in the management port 24410:

```
iastool -ping -hub BES1 -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 372 for more information.

Example

The following example shows how to enable the partition naming service on the standard partition.

```
iastool -pservice -hub BES1 -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.
`-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 372 for more information.

Examples

The following example restarts the hub `BES1` on the default management port:

```
iastool -restart -hub BES1
```

The following example restarts the partition naming service running on the hub `BES1` on the management port 24410:

```
iastool -restart -hub BES1 -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 372 for more information.

Example

The following example starts the partition naming service running on the hub `BES1` in the `j2ee` configuration on management port 24410:

```
iastool -start -hub BES1 -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.

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 372 for more information.

Example

The following example stops the partition naming service running on the hub `BES1` in the `j2ee` configuration on the management port 24410:

```
iastool -stop -hub BES1 -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.
`-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.

Option	Description
`-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 372 for more information.

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 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:

- "Piping a file to the iastool utility" on page 373
- "Passing a file to the iastool utility" on page 373

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:\BES\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:\BES\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

30

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.

Important For documentation updates, go to `www.borland.com/techpubs/bes`.

<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 Enterprise Server 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:

- <statistics.agent>
- <security>
- <container>
- <user.orb>
- <management.orb>
- <shutdown>
- <services>
- <archives>

<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.

Attribute	Description
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 seperated 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.

Attribute	Description
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 seperate 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 (;) seperated 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.

Attribute	Description
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.

Attribute	Description
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) .
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.

Attribute	Description
`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

31

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`

Property	Description	Default
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 strucutres 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 BES 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

Property	Description	Default
ejb.system_classpath_first=true|false	If set to true, the custom classloader will look at the system classpath first.	false
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

Property	Description	Default
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
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

Property	Description	Default
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.	no limit
ejb.mdb.threadMin=<num>	Minimum number of threads allowed in the MDB thread pool.	0

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 appservers 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

Property	Type	Description	Default
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
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 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 e.g. in entity beans. 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
ejb.checkExistenceBeforeCreate	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. Rarely useful.	false
ejb.mdb.local_transaction_optimization	Boolean	Not yet implemented	false
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 BES populates the pool with.	None
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

Property	Type	Description	Default
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

Property	Type	Description	Default
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 e.g. 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.	False

Session Service (JSS) Properties

The Session Service can run as part of standalone EJB container (-jss option) or as part of the 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>/<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>/standard/adm/properties/partition.xml
```

For more information, go to the *partition.xml reference*, "<service> element" on page 380.

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 (mo) directory attribute:

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

For more information about the `configuration.xml`, go to the *Deployment Operations Center (BDOC) Developer's Guide*, configuration.xml reference.

The JSS supports two kinds of backend storage: JDataStore or a JDBC datasource. For more information, go to Chapter 6, “Java Session Service (JSS) configuration”.

Property	Console Property Name	Description	Default
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>/<partition_name>. If not specified and the JSS runs in a standalone EJB container, the default value is EJB/JSS[<container_name>] .
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>/<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.

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.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.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)

Property	Console Property Name	Description	Default
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.debug=true|false	Debug	Print debug information. If set to true, the JSS prints out debug traces.	false

Old style EJB Container and JSS Properties

Some properties can also be specified as they were named in IAS 4.x. The table below defines the correspondence between new and old style properties. Note, that this works only with an EJB container started standalone.

Old Style (IAS 4.x>	New style (BES 5.x)
EJBCopyArgs	ejb.copy_arguments=true
EJBUseJavaSerialization	ejb.use_java_serialization=true
EJBNoSleep	ejb.no_sleep=true
EJBNoClassLoader	ejb.classload_policy=none
EJBOneClassLoader	ejb.classload_policy=container
EJBPassivationTimeout=<num>	ejb.sfsb.passivation_timeout=<num>
EJBDiagnosticPeriod=<num>	ejb.collect.stats_gather_frequency=<num>
EJBQuietTimers	ejb.collect.statistics=true
EJBTimers	ejb.collect.display_statistics=true
EJBDetailTimers	ejb.collect.display_detail_statistics=true
EJBDebug	ejb.trace_container=true
EJBDefaultStorageTimeout=<num>	jss.maxIdle=<num>

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.	`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 comparision 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 transaciton.	`False`

Property	Description	Default
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
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

JTS System Properties

JTS properties can also be specified as system properties as used in IAS 4.x and BAS 4.5. The table below defines the correspondence between new and old style properties. Note, that this works only with an EJB container started standalone.

Old Style (IAS 4.x>	New style (BES 5.x)
EJBAllowUnrecoverableCompletion	jts.allow_unrecoverable_completion=true
EJBNoGtids	jts.no_global_tids=true
EJBNoLtids	jts.no_local_tids=true
EJBTxTimeoutEnable	jts.timeout_enable=true
EJBTxTimeoutInterval=<num>	jts.timeout_interval=<num>
EJBTxDefaultTimeout=<num>	jts.default_timeout=<num>
EJBTxDefaultMaxTimeout=<num>	jts.default_max_timeout=<num>
EJBTxDebug	jts.trace=true
EJBTxTransactionDebugTimeout=<num>	jts.transaction_debug_timeout=<num>

Chapter

32

ejb-borland.xml

Important For documentation updates, go to www.borland.com/techpubs/bes.

DTD

```
<!ELEMENT ejb-jar (enterprise-beans, datasource-definitions?,
 table-properties*, relationships?, authorization-domain?,
 property*, assembly-descriptor?)>

<!ELEMENT enterprise-beans (session | entity | message-driven)+>

<!ELEMENT session (ejb-name, bean-home-name?, bean-local-home-name?,
 timeout?, ejb-ref*, ejb-local-ref*, resource-ref*,resource-env-ref*,
property*)>

<!ELEMENT entity (ejb-name, bean-home-name?, bean-local-home-name?,
 ejb-ref*, ejb-local-ref*, resource-ref*, resource-env-ref*,
 (cmp-info | cmp2-info)?, property*)>

<!ELEMENT message-driven (ejb-name, message-driven-destination-name,
 connection-factory-name, pool?, ejb-ref*, ejb-local-ref*,
 resource-ref*, resource-env-ref*, property*)>

 <!ELEMENT pool (max-size?, init-size?, wait-timeout?)>
 <!ELEMENT ejb-ref (ejb-ref-name, jndi-name?)>
 <!ELEMENT ejb-local-ref (ejb-ref-name, jndi-name?)>
 <!ELEMENT resource-ref (res-ref-name, jndi-name, cmp-resource?)
```

```
<!ELEMENT resource-env-ref (resource-env-ref-name, jndi-name)
<!ELEMENT datasource-definitions (datasource*)>
<!ELEMENT datasource (jndi-name, url, username?, password?,
 isolation-level?, driver-class-name?, jdbc-property*, property*)>
<!ELEMENT jdbc-property (prop-name, prop-value)>
<!ELEMENT property (prop-name, prop-type, prop-value)>
<!ELEMENT cmp-info (description?, database-map?, finder*)>
<!ELEMENT database-map (table?, column-map*)>
<!ELEMENT finder (method-signature, where-clause, load-state?)>
<!ELEMENT column-map (field-name, column-name?, column-type?, ejb-ref-
name?)>
<!ELEMENT connection-factory-name (#PCDATA)>
<!ELEMENT message-driven-destination-name (#PCDATA)>
<!ELEMENT max-size (#PCDATA)>
<!ELEMENT init-size (#PCDATA)>
<!ELEMENT wait-timeout (#PCDATA)>
<!ELEMENT cmp-resource (#PCDATA)>
<!ELEMENT method-signature (#PCDATA)>
<!ELEMENT where-clause (#PCDATA)>
<!ELEMENT load-state (#PCDATA)>
<!ELEMENT prop-name (#PCDATA)>
<!ELEMENT prop-type (#PCDATA)>
<!ELEMENT prop-value (#PCDATA)>
<!ELEMENT field-name (#PCDATA)>
<!ELEMENT column-name (#PCDATA)>
<!ELEMENT column-type (#PCDATA)>
<!ELEMENT table (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT ejb-name (#PCDATA)>
<!ELEMENT bean-home-name (#PCDATA)>
<!ELEMENT bean-local-home-name (#PCDATA)>
<!ELEMENT timeout (#PCDATA)>
<!ELEMENT ejb-ref-name (#PCDATA)>
<!ELEMENT jndi-name (#PCDATA)>
<!ELEMENT res-ref-name (#PCDATA)>
<!ELEMENT resource-env-ref-name (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT username (#PCDATA)>
<!ELEMENT password (#PCDATA)>
<!ELEMENT isolation-level (#PCDATA)>
<!ELEMENT driver-class-name (#PCDATA)>
<!ELEMENT authorization-domain (#PCDATA)>
<!ELEMENT table-properties (table-name, column-properties*, property*)>
<!ELEMENT column-properties (column-name, property*)>
<!ELEMENT table-name (#PCDATA)>
<!ELEMENT cmp2-info (cmp-field*, table-name, table-ref*)>
<!ELEMENT relationships (ejb-relation+)>
<!ELEMENT ejb-relation (ejb-relationship-role, ejb-relationship-role)>
<!ELEMENT ejb-relationship-role (relationship-role-source, cmr-field?)>
<!ELEMENT relationship-role-source (ejb-name)>
<!ELEMENT cmr-field (cmr-field-name, table-ref, property*)>
<!ELEMENT cmr-field-name (#PCDATA)>
```

```
<!ELEMENT table-ref (left-table, cross-table*, right-table)>
<!ELEMENT left-table (table-name, column-list)>
<!ELEMENT right-table (table-name, column-list)>
<!ELEMENT cross-table (table-name, column-list, column-list)>
<!ELEMENT column-list (column-name+)>
<!ELEMENT cmp-field (field-name, (cmp-field-map* | column-
name),property*)>
<!ELEMENT cmp-field-map (field-name, column-name)>
<!ELEMENT assembly-descriptor (security-role*)>
<!ELEMENT security-role (role-name, deployment-role?)>
<!ELEMENT role-name (#PCDATA)>
<!ELEMENT deployment-role (#PCDATA)>
```

Chapter

33

application-client-borland.xml

Important For documentation updates, go to www.borland.com/techpubs/bes.

DTD

```
<!ELEMENT application-client (ejb-ref*, resource-ref*,
           resource-env-ref*, property*)>
<!ELEMENT ejb-ref (ejb-ref-name, jndi-name?)>
<!ELEMENT resource-ref (res-ref-name, jndi-name)>
<!ELEMENT resource-env-ref (resource-env-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)>
<!ELEMENT resource-env-ref-name (#PCDATA)>
```

Chapter

34

ra-borland.xml

Important For documentation updates, go to www.borland.com/techpubs/bes.

DTD

```
<!--
This DTD defines the Borland specific deployment information for defining
a deployable Resource Adapter Connection Factory. It provides for complete
specification of all configurable Connection Factory parameters including
Connection Pool parameters, Security parameters for Resource Role Mapping
and the ability to define values for configuration parameters which exist
in
the ra.xml deployment descriptor.
-->

<!ELEMENT connector (connection-factory)>

<!--
The connection-factory element is the root element of the
Borland specific deployment descriptor for the deployed
resource adapter.
-->

<!ELEMENT connection-factory (factory-name, factory-description?, jndi-
name,
ra-link-ref?, ra-libraries?, pool-parameters?), (logging-enabled, log-file-
name?),
property*, security-map*, authorization-domain?)>
```

```
<!--
The factory-name element defines that logical name that
will be associated with this specific deployment of the
Resource Adapter and its corresponding Connection Factory.

The value of factory-name can be used in other deployed
Resource Adapters via the ra-link-ref element. This will
allow multiple deployed Connection Factories to utilize a
common deployed Resource Adapter, as well as share
configuration specifications.

This is a required element.
-->

<!ELEMENT factory-name (#PCDATA)>

<!--
The factory-description element is used to provide text
describing the parent element. The factory-description
element should include any information that the deployer
wants to describe about the deployed Connection Factory.

This is an optional element.
-->

<!ELEMENT factory-description (#PCDATA)>

<!--
The jndi-name element defines the name that will be used to bind the
Connection Factory Object into the JNDI Namespace. Client EJBs and
Servlets will use this same JNDI in their defined Reference Desciptor
elements of the Borland specific deployment descriptors.

This is a required element.
-->

<!ELEMENT jndi-name (#PCDATA)>

<!--
The ra-link-ref element allows for the logical association of
multiple deployed Connection Factories with a single deployed Resource
Adapter. The specification of the optional ra-link-ref element with
a value identifying a separately deployed Connection Factory will
result in this newly deployed Connection Factory sharing the
Resource Adapter which had been deployed with the referenced
Connection Factory.

In addition, any values defined in the referred Connection Factories
deployment will be inherited by this newly deployed Connection Factory
unless specified.

This is an optional element.
-->
```

```
<!ELEMENT ra-link-ref (#PCDATA)>

<!--
The ra-libraries element identifies the directory location to be
used for all native libraries present in this resource adapter
deployment. As part of deployment processing, all encountered
native libraries will be copied to the location specified.

It is the responsibility of the Administrator to perform the necessary
platform actions such that these libraries will be found at runtime.

This is a required element IF native libraries are present.
-->

<!ELEMENT ra-libraries (#PCDATA)>

<!--
The pool-parameters element is the root element for providing Connection
Pool specific parameters for this Connection Factory.

VisiConnect will use these specifications in controlling the behavior
of the maintained pool of Managed Connections.

This is an optional element.Failure to specify this element or any
of its specific element items will result in default values being
assigned. Refer to the description of each individual element for
the designated default value.
-->

<!ELEMENT pool-parameters (initial-capacity?, maximum-capacity?,
capacity-delta?, cleanup-enabled?, cleanup-delta?)>

<!--
The initial-capacity element identifies the initial number of managed
connections which VisiConnect will attempt to obtain during deployment.

This is an optional element.

Failure to specify this value will result in VisiConnect using its
defined default value.

Default Value: 1
-->

<!ELEMENT initial-capacity (#PCDATA)>

<!--
The maximum-capacity element identifies the maximum number of
managed connections which VisiConnect will allow.Requests for newly
allocated managed connections beyond this limit will result in a
ResourceAllocationException being returned to the caller.

This is an optional element.
```

```
Failure to specify this value will result in VisiConnect using its
defined default value.

Default Value:  10
-->

<!ELEMENT maximum-capacity (#PCDATA)>

<!--
The capacity-delta element identifies the number of additional
managed connections which the VisiConnect will attempt to obtain
during resizing of the maintained connection pool.

This is an optional element.

Failure to specify this value will result in VisiConnect using its
defined default value.

Default Value: 1
-->

<!ELEMENT capacity-delta (#PCDATA)>

<!--
The cleanup-enabled element indicates whether or not the
Connection Pool should have unused Managed Connections reclaimed
as a means to control system resources.

This is an optional element.

Failure to specify this value will result in VisiConnect using its
defined default value.

Value Range: true|false

Default Value: true
-->

<!ELEMENT cleanup-enabled (#PCDATA)>

<!--
The cleanup-delta element identifies the amount of time the
Connection Pool Management will wait between attempts to reclaim
unused Managed Connections.

This is an optional element.

Failure to specify this value will result in VisiConnect using its
defined default value.

Default Value: 1
-->
```

```
<!ELEMENT cleanup-delta (#PCDATA)>

<!--
The logging-enabled element indicates whether or not the log writer
is set for either the ManagedConnectionFactory or ManagedConnection.
If this element is set to true, output generated from either the
ManagedConnectionFactory or ManagedConnection will be sent to the file
specified by the log-filename element.

This is an optional element.

Failure to specify this value will result in VisiConnect using its
defined default value.

Value Range: true|false

Default Value: false
-->

<!ELEMENT logging-enabled (#PCDATA)>

<!--
The log-file-name element specifies the name of the log file which output
generated from either the ManagedConnectionFactory or a ManagedConnection
are sent.

The full address of the file name is required.

This is an optional element.
-->

<!ELEMENT log-file-name (#PCDATA)>

<!--
Each property element identifies a configuration property
name, type and value that corresponds to an ra.xml entry
element with the corresponding property-name.

At deployment time, all values present in a property
specification will be set on the ManagedConnectionFactory.

Values specified via a property will supersede any default
value that may have been specified in the corresponding ra.xml
config-property element.

This is an optional element.
-->

<!ELEMENT property (prop-name, prop-type, prop-value)>
<!ELEMENT prop-name (#PCDATA)>
<!ELEMENT prop-type (#PCDATA)>
<!ELEMENT prop-value (#PCDATA)>
```

```
<!--
The security-map element specifies whether the caller's security identity
is to be used for the execution of the methods of the enterprise bean
or whether a specific run-as identity is to be used. It contains an
optional description and a specification of the security identity to
be used.

Each security-map element provides a mechanism to define appropriate
Resource Role values for Resource Adapter/EIS authorization processing,
through the use of the run-as element.

This element allows for the specification of a defined set of
user roles and the corresponding run-as roles (representing
EIS identities) that should be used when allocating
Managed Connections and Connection Handles.

A default Resource run-as role can be defined for the Connection
Factory via the map. By specifying a user-role value of '*'
and a corresponding run-as role, the defined run-as will be utilized
whenever the current role is NOT matched elsewhere in
the map.

This is an optional element, however, it must be specified in some
form if Container Managed Sign-on is supported by the Resource Adapter
and used by ANY client.

In addition, the deployment-time population of the Connection Pool
with Managed Connections will be attempted using the defined
'default' run-as if one is specified.
-->

<!ELEMENT security-map (description?, user-role+, (use-caller-identity|run-
as))>

<!--
The user-role element contains one or more role names, defined for
use as the security identity, or mapped to a appropriate Resource
Role run-as identity, for interactions with the resource.
-->
<!ELEMENT user-role (#PCDATA)>

<!--
The use-caller-identity element specifies that the caller's security
identity be used as the security identity for the execution of the
Resource Adapter's methods.

Used in: security-map
-->
<!ELEMENT use-caller-identity EMPTY>

<!--
The run-as element specifies the run-as identity to be used for the
execution of the enterprise bean.It contains an optional description, and
```

```
the name of a security role.

Used in: security-map
-->
<!ELEMENT run-as (description?, role-name)>

<!--
The role-name element contains the name of a security role.

The name must conform to the lexical rules for an NMTOKEN.

Used in: run-as
-->

<!ELEMENT role-name (#PCDATA)>

<!--
The authorization-domain element specifies the authorization domain to
be used for determining the definable set of valid user roles.
-->

<!ELEMENT authorization-domain (#PCDATA)>

<!ELEMENT description (#PCDATA)>
```

Chapter

35

jndi-definitions.xml

Important For documentation updates, go to www.borland.com/techpubs/bes.

DTD

```
<!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)>
```

Chapter

36

web.xml

Important For documentation updates, go to www.borland.com/techpubs/bes.

DTD

```
<!ELEMENT web-app(context-root?, resource-env-ref*, resource-ref*,
        ejb-ref*, ejb-local-ref*, property*, web-deploy-path*,
        authorization-domain?, security-role*)>
<!ELEMENT ejb-ref (ejb-ref-name, jndi-name)>
<!ELEMENT ejb-local-ref (ejb-ref-name, jndi-name?)>
<!ELEMENT resource-ref (res-ref-name, jndi-name)>
<!ELEMENT resource-env-ref (resource-env-ref-name, jndi-name)>
<!ELEMENT web-deploy-path (service, engine, host)>
<!ELEMENT context-root (#PCDATA)>
<!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)>
<!ELEMENT resource-env-ref-name (#PCDATA)>
<!ELEMENT service (#PCDATA)>
<!ELEMENT engine (#PCDATA)>
<!ELEMENT host (#PCDATA)>
<!ELEMENT authorization-domain (#PCDATA)>
<!ELEMENT security-role (role-name, deployment-role?)>
```

```
<!ELEMENT role-name (#PCDATA)>
<!ELEMENT deployment-role (#PCDATA)>
```

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