Chordiant

Chordiant 5

FOUNDATION SERVER

DEVELOPER'S GUIDE

Important Notice

Trademark Acknowledgments

Publishing Information

Chordiant 5 Foundation Server Developer's Guide
Release 5.7. Document version 1.0 February 2005

Please email suggestions and corrections to tech_com_ww@chordiant.com.

World Headquarters

Chordiant Software, Inc.
20400 Stevens Creek Blvd.
Cupertino, CA 95014
1-408-517-6100
Fax: 1-408-517-0270
www.chordiant.com

European Headquarters

Chordiant Software Int'l, Inc.
2 Goat Wharf, High Street
Brentford, Middlesex
TW8 0BA, UK
+44(0) 20 8380 0600
Fax: +44(0) 20 8380 0606

Customer Support

Americas:
us.support@chordiant.com
1-408-517-6200
Toll-free: 1-877-866-7243
EMEA:
+44 (0) 20 8580 0400
uk.support@chordiant.com

Contents

Preface

This manual describes the architecture and design disciplines of the Chordiant 5 Foundation Server. It also describes how to develop applications using Chordiant 5 Foundation Server, and how to administer security for applications running in this environment.

Who Should Use this Manual

This manual is intended for Chordiant Application Developers and System Integrators who need to define and develop applications using the Chordiant 5 Foundation Server.

Manual Organization

This manual contains the following chapters:

Chapter 1	Provides an introduction to the Chordiant 5 Foundation Server, and describes the important features and concepts of the JX Architecture.
Chapter 2	Describes the architecture of the Chordiant 5 Foundation Server, and provides a detailed explanation of the major components of the system. This chapter includes a description of the single-bean architecture and transactions within the JX EJB.
Chapter 3	Describes the life cycle of a Foundation Server application.
Chapter 4	Describes managing state in JX services, including stateless and stateful models.
Chapter 5	Describes Chordiant 5 Foundation Server helpers, including LogHelper, ConfigurationHelper, GatewayHelper, StaticHelper, and ClientAgentHelper.
Chapter 7	Describes configuration files and configuring Chordiant 5 Foundation Server.
Chapter 6	Describes monitoring the system through the Administrative Console, as well as exceptions and error handling.
Chapter 8	Describes how to build a distributed application using Chordiant 5 Foundation Server, including creating services and client agents. Also discusses web services, transactions, smartstubs, and the Resource Manager.
Chapter 9	Describes Chordiant Persistence Server and how to add persistence features to your business services.
Chapter 10	Describes how to use the Chordiant Event Server for asynchronous messaging in Foundation Server applications.
Chapter 11	Describes the security architecture of Chordiant 5 Foundation Server.

Chapter 12	Describes the Request Server used for web applications, including how to build web applications.
Chapter 13	Describes Network Presence and its role in working with thin clients.

Additional Documentation

For more information on Chordiant 5 Foundation Server, refer to the following documents:

- *Chordiant 5 Foundation Server Customization Guide* — Contains information on customizing Chordiant 5 Foundation Server.
- *Chordiant 5 Foundation Server Application Components Developer's Guide* — Contains information on working with application components, including service framework components, web services components, and persistence components.
- Chordiant 5 Performance Guide — Contains tuning and performance information.

For definitions of Chordiant terms, refer to the *Chordiant 5 Terminology Guide*.

Typographical Conventions

This section explains how to interpret the font changes and notes that you see in this manual.

CONVENTION	EXAMPLE
System filenames and pathnames	`Readme.txt` is a text file that is stored on the application server in the `/etc` (for UNIX) or `C:\` (for Windows NT) directory.
Document names and module names	See the "Security" section within the *Ongoing Tasks* document, or the online help from within the *Security* module.
Names of code elements and small pieces of code - or - Onscreen text and text typed on the keyboard	Use the getInfo method Type the password cmyk.
Screen element labels, including buttons and menus - or - Keys that you press on the keyboard	Click **OK**. Then from the **File** menu, select **Save**. To save the information on the page, press **CTRL + SHIFT + s**.
Variables that you must define based on your own settings	*{JAVA_HOME}*`/com/chordiant/jxw`

```
Gray boxes show code to be entered or viewed.
```

Note: A note shows important information that you should be sure to read. Many notes refer to other sections for more information.

Tip: A tip gives suggestions on how you can use the application faster or more efficiently.

Caution: A caution statement warns of steps you should take, or avoid, so you do not damage your equipment, data, or system reliability.

Chapter 1

Introduction

Chordiant 5 Foundation Server is a set of distributed components that work together to provide an execution environment for eBusiness applications. Using Chordiant 5 Foundation Server, you can create distributed multi-channel, multi-datastore, and data-driven eBusiness applications based on open standards that can be adapted for future technologies.

Chordiant 5 Foundation Server enables you to build services that access back-end data stores, and to present those services to client applications, including those based on both thick and thin client models.

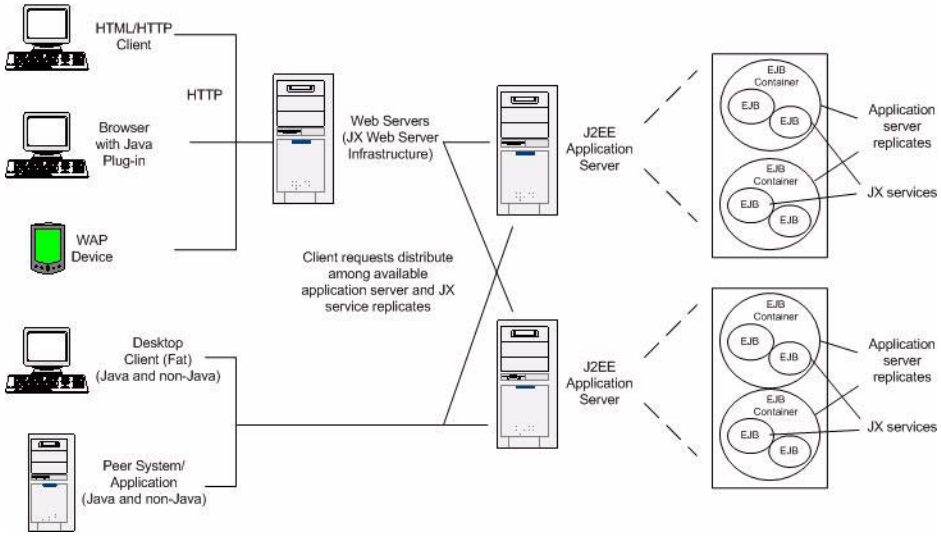


Figure 1-1: Chordiant 5 Foundation Server Overview

While Chordiant 5 Foundation Server offers a powerful and scalable application development environment, it is important to note that Chordiant 5 Foundation Server does not include any surrounding eBusiness applications.

IMPORTANT CHORDIANT 5 FOUNDATION SERVER CONCEPTS

There are several concepts that form the basis of the architecture used within the Chordiant 5 Foundation Server. The JX Architecture is so named because it is based on J2EE and XML specifications.

Table 1-1 describes these concepts in the context of the Chordiant Software system and the Chordiant 5 Foundation Server.

CONCEPT	DESCRIPTION
eBusiness applications	A set of software programs running on one or more computers, accessed by multiple users through a range of touch-points (channels).
Multi-channel	The capability of enabling access to eBusiness applications using a range of client entities including web browsers, wireless devices, desktop applications, and more. Multi-channel also includes access through peer system entities such as IVR/VRU systems, fax/email systems, as well as other external systems.
Multi-datastore	The ability for an eBusiness application to have consistent access to business data residing in the following locations: relational data stores, legacy application data stores, document management data stores, file system data stores, and more.
Data-driven	The ability to modify the behavior of an eBusiness application by manipulating data and meta-information instead of source code.
Distributed	An execution environment that enables eBusiness applications to run on multiple computers thereby offering scalable performance.
Vertical scaling	The ability to add service replicates on a given computer to achieve higher request throughput.
Horizontal scaling	The ability to add computers, and thereby add service replicates, to achieve higher request throughput.

Table 1-1: Important Chordiant 5 Foundation Server Concepts

CHORDIANT 5 FOUNDATION SERVER FEATURES AND ADVANTAGES

Chordiant 5 Foundation Server offers a framework built on the J2EE application model. Chordiant 5 Foundation Server enables you to rapidly create distributed eBusiness applications by providing a level of abstraction on top of the J2EE architecture, making it easier to build services and expose them to clients.

Figure 1-2 illustrates the standard J2EE application model. Within this application model, you create JX services that run as Enterprise Java Beans, which reside in the application server (an Enterprise Java Bean Container).

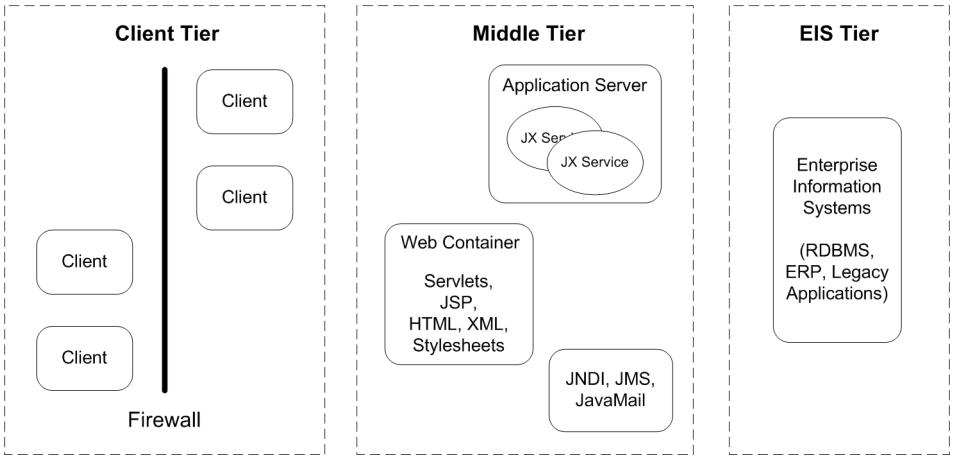


Figure 1-2: The J2EE Application Model

Within the Chordiant 5 Foundation Server, services implement a well-defined interface that embodies enterprise business logic. Enterprise business logic consists of transactions, data access, interactions with peer services, business logic, and rules.

Building a service using Chordiant 5 Foundation Server involves subclassing a JX base class, and writing the custom Java code. Unlike programming to the J2EE application model, however, you do not need to employ an Enterprise Java Beans (EJB) compiler, nor consider the specific application server to which you plan to deploy.

Business services perform data access using Chordiant Persistence Server (Persistence). Chordiant Persistence Server saves you from having to interact with a specific database interface directly. It handles the interaction with data and legacy systems for you. In contrast, when programming using the J2EE application model, you are responsible for directly programming to the database interface, such as JDBC.

Likewise, Chordiant 5 Foundation Server does not require you to expose your service interface directly to clients. Instead, Chordiant 5 Foundation Server uses a client agent to which you expose your interfaces. This means that you do not need to perform client binding or deployment for the EJB.

Chordiant 5 Foundation Server also offers a single distributed interface coupled with a transform technology that enables components within the distributed application, including clients and servers, to exchange information in a seamless fashion using an extensible, XML-based payload. Because the payload is XML-based, you can extend its data structure without having to rebuild the distributed interfaces.

FOUNDATION SERVER COMPONENT INTERACTIONS

Chordiant 5 Foundation Server employs a layered software architecture that includes well-defined interactions between components such as the client application, the client agent, the service, the data access component (Chordiant Persistence Server), and finally, the underlying data store.

Figure 1-3 shows the Chordiant 5 Foundation Server application architecture, illustrating the multi-layered structure of the software system.

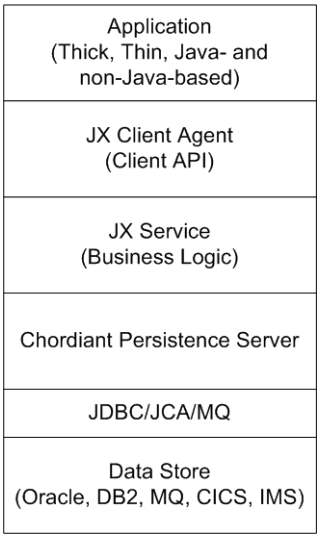


Figure 1-3: Chordiant 5 Foundation Server Component Interactions

Table 1-2 describes the major components of Chordiant 5 Foundation Server.

COMPONENT	DESCRIPTION
Client Application	The client application that, in many cases, is responsible for displaying information and accepting user input.
JX Client Agent	A proxy to the services on the server, enabling the client application to access the service's functionality.

Table 1-2: Chordiant 5 Foundation Server Components

COMPONENT	DESCRIPTION
JX Service	Embodies the business logic and data access requirements of the application. Services run on the application server, typically reside in the middle tier, and can be stateful or stateless. You implement services using Java by subclassing a known class and implementing a known interface. Services can be single- or multi-threaded.
Chordiant Persistence Server (data access component)	An abstraction of the data manipulation features available to the service. The Chordiant Persistence Server component implements the standard CRUD (Create, Retrieve, Update, Delete) operations, and uses an underlying data interface, such as JDBC, Java Connect Architecture (JCA) or MQ, to interface with the specific data store.
Data Store	The underlying data storage system. Examples of data stores include RDBMS databases, legacy applications and flat files.

Table 1-2: Chordiant 5 Foundation Server Components (Continued)

Additional Components and Concepts

Chordiant 5 Foundation Server includes several additional components and concepts that ease the development of distributed applications. Table 1-3 describes these components and concepts.

ITEM	DESCRIPTION
Callback	A feature that enables the server to call the client application. Typical uses include enabling a service, such as workflow, chat, or email to notify the client about some event. The callback feature is implemented as part of the GatewayHelper's capabilities
GatewayHelper	Provides a network addressable mechanism for software components to call the client application. The GatewayHelper therefore enables services to perform a callback to client applications. Note that all interactions in a typical J2EE application are initiated from the client to the server. Using the Chordiant 5 Foundation Server, however, you can have the server spontaneously initiate a callback to the client application. This enables the server to offer notifications to the client without requiring the client to poll the server.

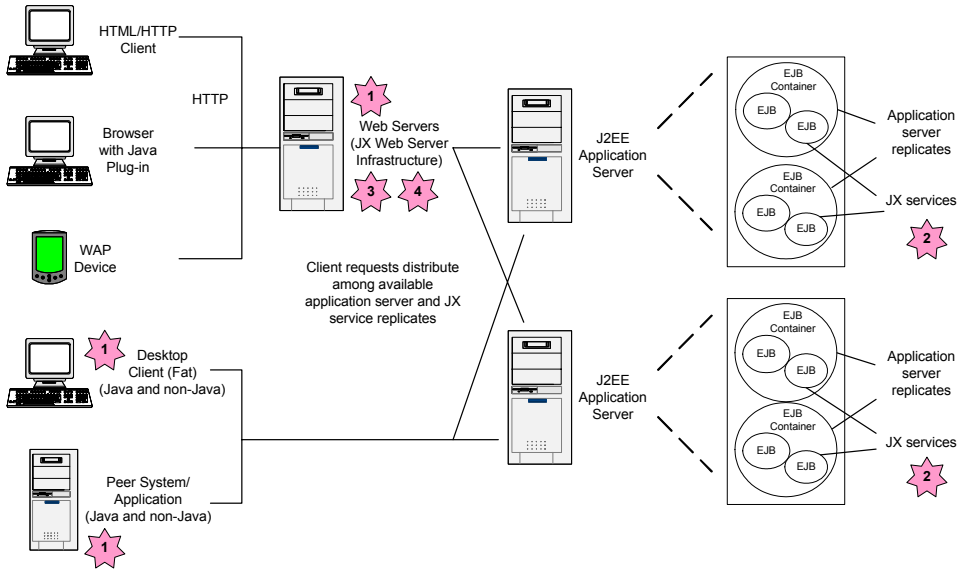
Table 1-3: Additional Components and Concepts

ITEM	DESCRIPTION
Servlets and JSPs	In the Chordiant 5 Foundation Server, you can use servlets to extend a server's functionality. Servlets are written in Java, and run on the web server. Unlike services, servlets represent the application and not the business logic. Typically, servlets and JSPs output HTML to HTTP clients, such as web browsers or wireless phones. Chordiant 5 Foundation Server also offers the infrastructure enabling servlets or JSPs to output XML, which is transformed using XSL stylesheets to either HTML or other specific renderings, such as WML. The advantage of this approach is that it enables you to reuse application logic over multiple channels, such as web browsers and wireless phones. Servlets and JSPs interact with client agents to access business logic services.
Stylesheets	XSL-based information that provides the styling for XML output, interpreted using a transform engine. Stylesheets enable channel- and device- (context) specific rendering of generated information.

Table 1-3: Additional Components and Concepts (Continued)

CHORDIANT 5 FOUNDATION SERVER DEVELOPMENT

Figure 1-4 shows the components that you can build with Chordiant 5 Foundation Server.



1. Client Agents
2. Services
3. Servlets
4. Stylesheets/XSL

Figure 1-4: Components Developed with Chordiant 5 Foundation Server

Chapter 2

Understanding JX Architecture

Before you begin working with Chordiant 5 Foundation Server, you should understand the JX architecture. This chapter introduces the JX architecture, its main components and technologies, and the interaction between clients and services running on application servers.

Enterprise Services Topology

Figure 2-1 illustrates the service topology supported by Chordiant 5 Foundation Server.

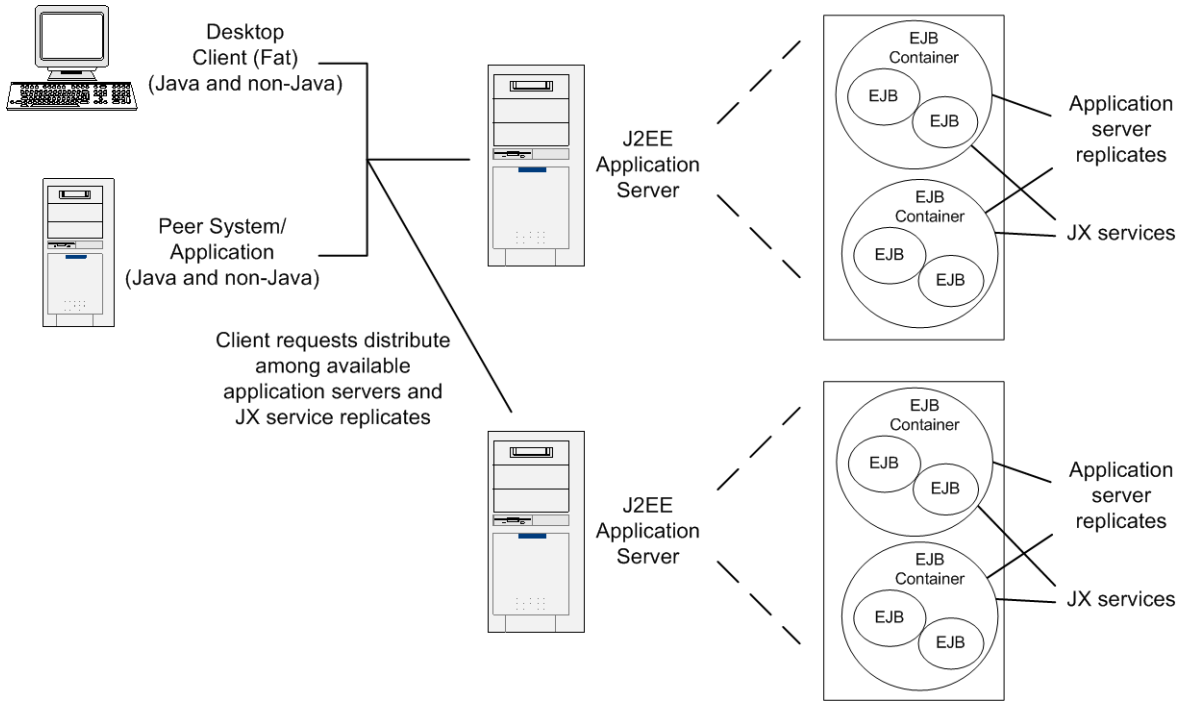


Figure 2-1: Service Topology

The service topology supported by the Chordiant 5 Foundation Server comprises:

- **One or more J2EE application servers**

  Each physical server can host multiple application server replicates, which are the containers for the JX services, running as Enterprise Java Beans.

- **Clients**

  These include thick desktop clients and peer systems running dedicated applications.

Using multiple servers, Chordiant 5 Foundation Server enables you to create a robust, fault-tolerant, and load-balanced execution environment for business applications. The system distributes client requests among application servers and JX application service replicates.

ENTERPRISE SERVICES DETAIL

Services within the Chordiant 5 Foundation Server run as Enterprise Java Beans within an EJB container (EJB Containers are application server replicates). Figure 2-2 illustrates the relationship between a JX service, the JX infrastructure, and the EJB container.

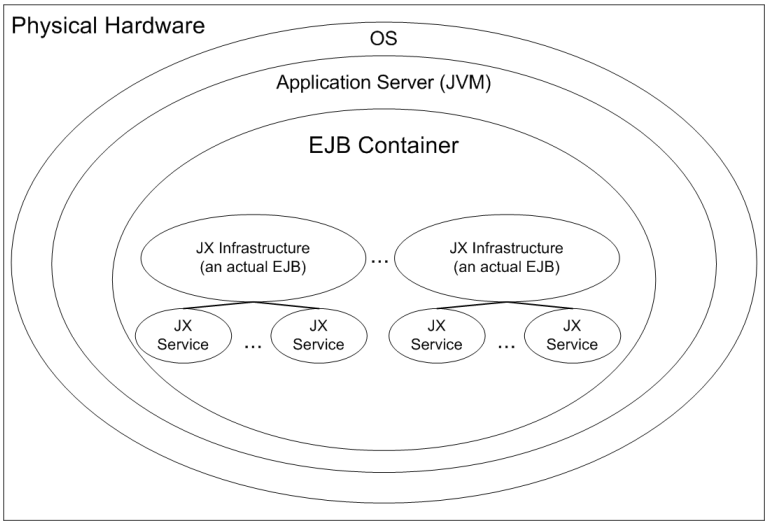


Figure 2-2: Enterprise Services Detail

Using Chordiant 5 Foundation Server, you implement custom services which are Java classes. This is in contrast to the procedure for developing a conventional EJB, which requires you to compile and deploy the EJB to a J2EE application server when adding a new service or changing service interfaces.

Since Chordiant 5 Foundation Server relies on a single EJB hosting all services running as Java classes, you can introduce new services without having to reconfigure the EJB. Also, since the service is simply an implementation of a Java class, you can develop, test, and run the class independently of J2EE. Additionally, the Chordiant 5 Foundation Server leverages the bean pool and thread pool model of J2EE.

For more information about the JX architecture and EJBs, refer to "Single-Bean Architecture" on page 19.

Web Application Component Detail

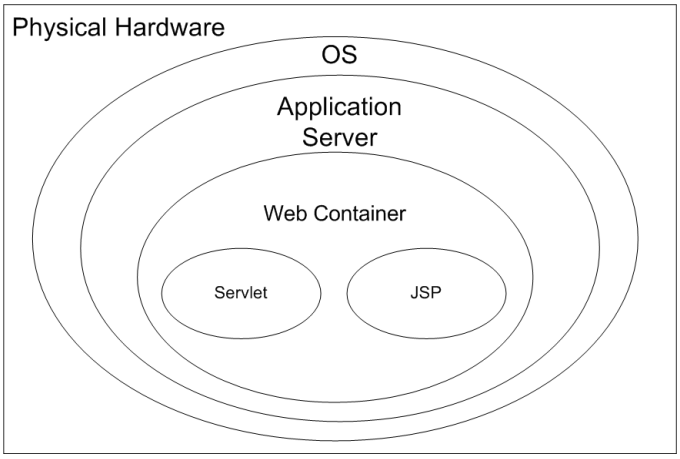


Figure 2-3: Web Application Component Detail

The web container facilitates:

- HTTP server
- JSP/Servlet runner

You can have the HTTP server and JSP/Servlet runner execute separately from the EJB container, for example Apache or Tomcat. Alternatively, you can combine the JSP/Servlet runner with the EJB container. This is a deployment decision based on the specific J2EE application server that you are using.

SERVICE AND WEB APPLICATION COMPONENT TOPOLOGY

Figure 2-4 illustrates the service and web application component topology supported by Chordiant 5 Foundation Server.

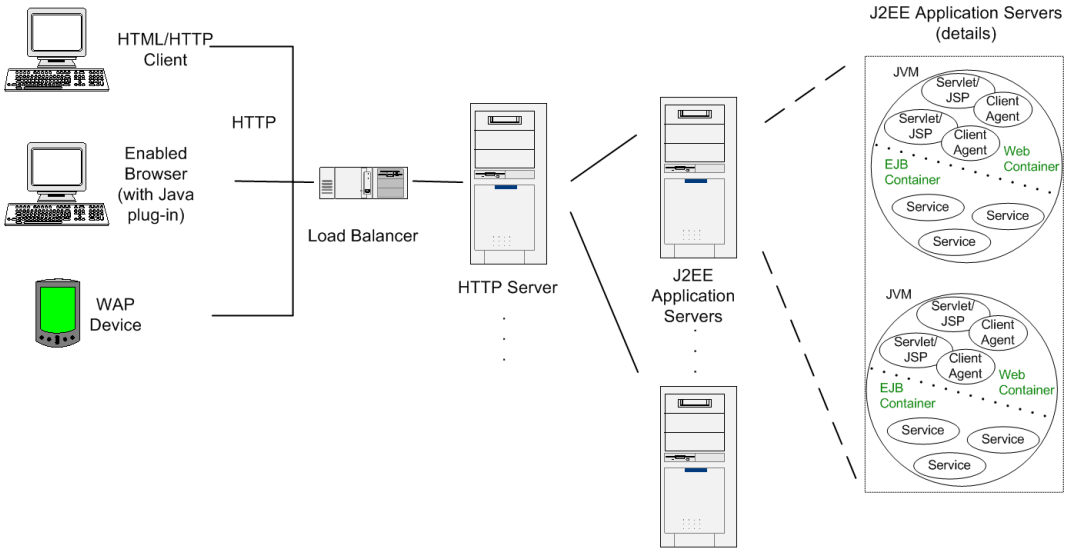


Figure 2-4: Service and Web Application Component Topology

The Service and Web Application component topology supported by the Chordiant 5 Foundation Server comprises:

- **One or more J2EE application servers**

  Each physical server can host multiple application server replicates, which are the containers for the JX services, running as Enterprise Java Beans.

  Application servers also handle JNDI, Java Messaging Service (JMS), Java Transaction API (JTA), Java Management Extensions (JMX), and connection pooling.

- **One or more web servers**

  The web server hosts the Request Server, responsible for interacting with thin clients and serving as a bridge to the application servers.

Note: Normally, the J2EE application server serves as the EJB and web container.

- **Load Balancer**

  One or more load balancers distribute incoming requests among HTTP servers.

- **Clients**

  These include HTML-based clients, browsers (optionally with Java plug-in), and mobile thin clients, such as wireless devices.

Similar to the conventional JX application model, described in "Enterprise Services Topology" on page 9, multiple servers enable the Chordiant 5 Foundation Server to offer a robust, fault-tolerant, and load-balanced execution environment for business applications. The system distributes client requests among application servers and JX application service replicates.

Web development is described in Chapter 12, "Request Server", and in the CAFE and Chordiant Interaction Server documentation sets.

DEPLOYMENT MODEL

This section contains a brief discussion of the Chordiant 5 Foundation Server deployment model. For more in-depth information, refer to the Chordiant 5 Performance Guide or consult your Chordiant representative.

Descriptions of the various components used in these deployment scenarios are described after Figure 2-4 on page 12. In these scenarios, note that you receive performance benefits when components run in the same address space (the same OS level process). There might, however, be drawbacks. These issues are discussed as they apply to each scenario.

Scenario 1: The JSP/Servlet Runner (the web container) is separate from the EJB runner (the EJB container).

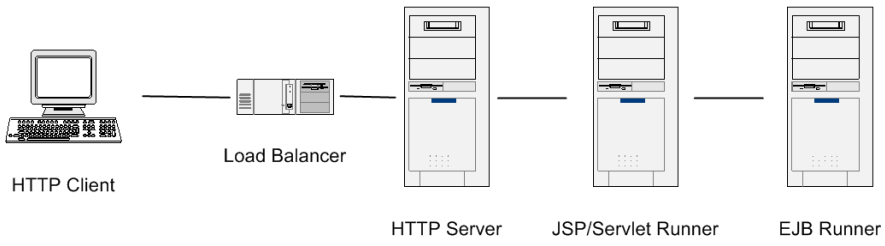


Figure 2-5: Scenario 1—All Components on Separate OS Processes and Separate Machines

You can run two instances of the application server—one to run the JSPs and Servlets and one to run the EJBs. This deployment model is typically chosen to separate the processing power required for the "application tier" from the processing power required by the "services tier".

Scenario 2: The HTTP Server, JSP/Servlet Runner and EJB runner all run in the same OS process.

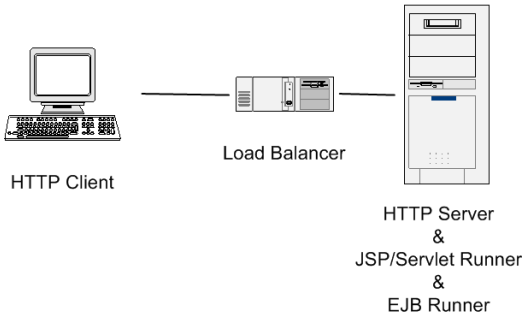


Figure 2-6: Scenario 2—Many Components Combined on One OS Process and One Machine

This scenario is useful for development because everything is running in one OS process. This makes it easy to manage. This is a good setup for trying out your code in a deployment setting.

Scenario 3: The JSP/Servlet Runner and the EJB Runner are in the same OS process, but the HTTP server is kept separate.

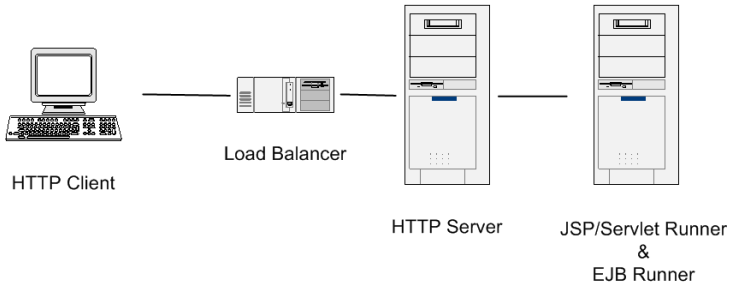


Figure 2-7: Scenario 3—JSP/Servlet Runner and EJB Runner Combined on One OS Process and One Machine

This scenario is typically the most common and optimizes JSP/Servlet to EJB communications.

JX Client Application Components

Chordiant 5 Foundation Server offers several application components you can use to develop both graphical and non-graphical client applications, as illustrated in Figure 2-8.

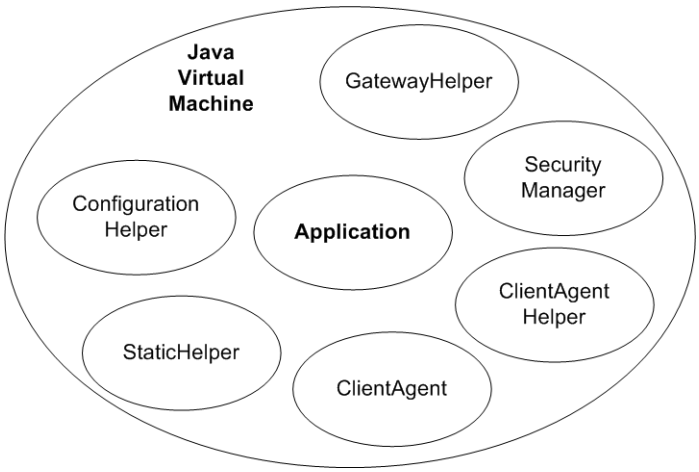


Figure 2-8: JX Client Application Components, running within the Java Virtual Machine

- **GatewayHelper —** This component enables a network presence for the client agent and client application, thus enabling a callback mechanism for the client agent and client application. The GatewayHelper automatically registers with the name service of the application server, and runs inside the client process. The client application is responsible for activating and deactivating the GatewayHelper as appropriate. In a thin client, the GatewayHelper runs as a Java applet.

- **SecurityManager —** The client application uses the SecurityManager to receive authentication before it can gain access to distributed resources. For example, the client application must complete authentication before communicating with a client agent. Upon doing so, the client application receives an authentication token back from the SecurityManager, which it uses to contact Chordiant's business services. When an application issues a call to the client agent, it passes the authentication token as one of the parameters. The JX infrastructure uses this token to determine access rights to the called service. You use the Chordiant 5 Foundation Server Administration Manager tool to configure the security information. For more information on security, refer to Chapter 11, "Security".
- **Client Agent —** Client agents serve as proxies to services, and are used by both client applications trying to contact services and by services trying to contact other services. The services then perform the desired work.

  To the client application, the JX client agent presents itself as a simple Java class with an arbitrary API. All remote features are hidden from the client application.

  A client agent is a subclass of the ClientAgentBaseClass that might implement the processCallback interface. Each client agent features individual network addressability through the GatewayHelper, assuming it has been enabled. This enables callbacks to be addressed to individual client agents.

  In addition to acting as a proxy to the services, client agents also fill the following roles:

  — Client agents contain implementations of callbacks.

  — Client agents are proxies for themselves. During a callback, they call their implementations to execute remotely on the application side.
- **ClientAgentHelper —** Vends a client agent to the client application, which is the only way that a client application can call a distributed service using the Chordiant 5 Foundation Server. You configure client agents using the Chordiant 5 Foundation Server configuration component. The ClientAgentHelper then locates the client agent configuration and creates a new client agent.

EXCHANGING INFORMATION THROUGH PAYLOAD

Chordiant 5 Foundation Server uses an extensible, XML-based or Java-object-based payload, coupled with transform technology, to offer a single distributed interface that enables components within the distributed application to exchange information in a seamless fashion. Once loaded, the payload can be passed to the services in one of three formats:

- A Java Object Graph (as qualified below)
- An XML document
- A String (java.lang.String)

In the case of a Java Object Graph, the graph must contain supported data types (see "Supported Data Types" on page 141) and each object in the graph must be serializable. A Java Object Graph can be arbitrarily deep, but must not contain circular references. Figure 2-9 illustrates a Java Object Graph and the equivalent XML Tree.

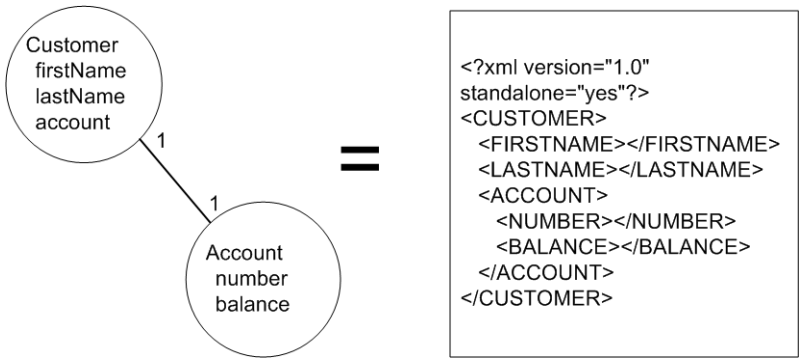


Figure 2-9: Mapping Object Trees to XML Trees

Using a client agent together with an extensible communication payload means that you do not need to recompile your applications when making changes to the type of information exchanged between the client and the server.

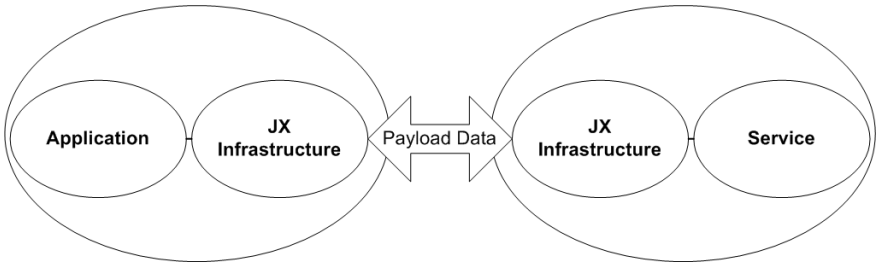


Figure 2-10: Transferring the Payload

In a typical J2EE application, on the other hand, the client interacts directly with the EJB. In addition, J2EE employs a strongly typed interface that requires recompiling when changes are made.

Note that the payload could be perceived differently in the client and the server. For example, because of the transforms involved, the client might pass an XML document containing a tree-structured representation of data. Meanwhile, the service could instead be configured to receive Java Object Graphs. The transform technology enables the service to receive the data in a format convenient for its purposes.

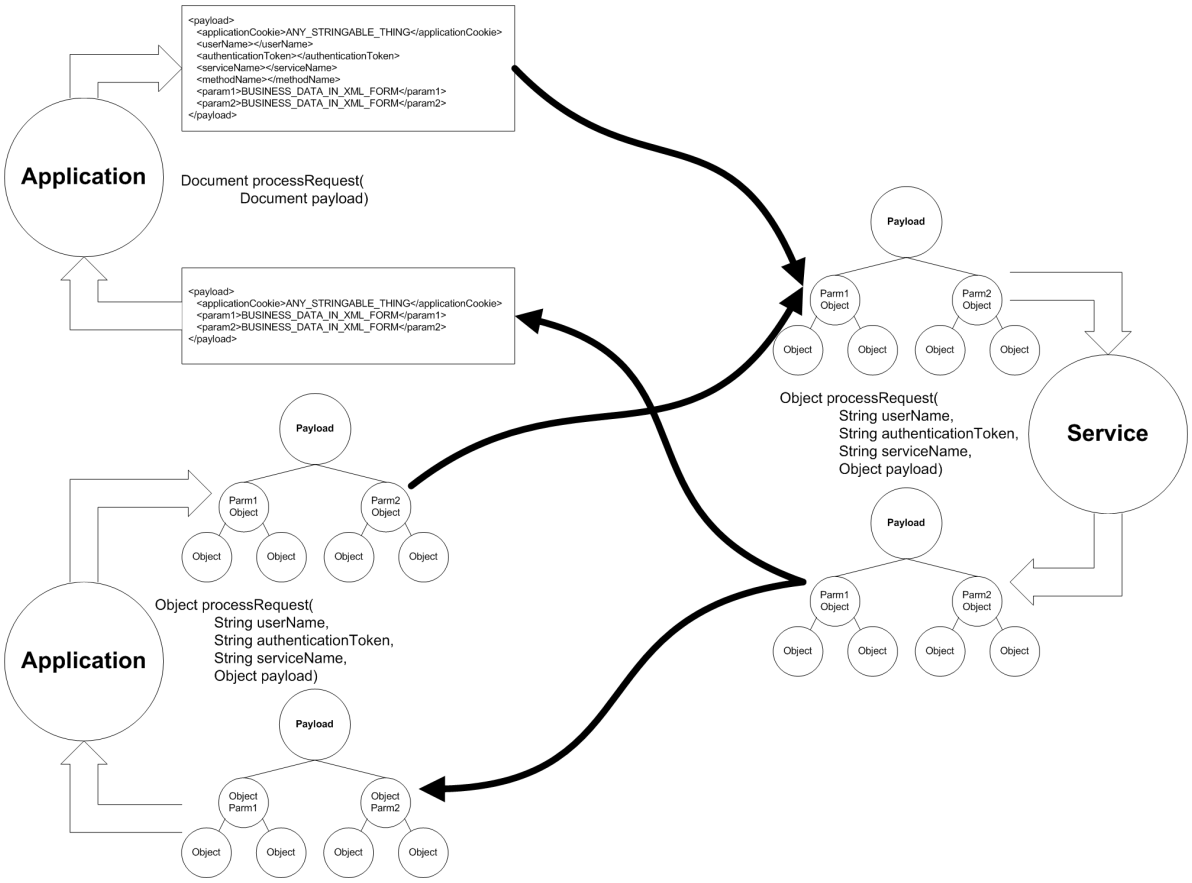


Figure 2-11: Distributed Business Data

The JX EJB has an interface, called processRequest, that expects object graphs or XML data. For information on processRequest, refer to "Building a Service" on page 112, "Building a Client Agent" on page 134, "Accessing Services without Client Agents" on page 149, and "processRequest Method: Client Agent vs. Service" on page 156.

processRequest requires that payload data be at the top of the object graph or, for XML, payload data must be at the root level. For more information on payload data, refer to "Passing Payload with PayloadData" on page 140.

CHORDIANT PERSISTENCE SERVER

The Chordiant Persistence Server component is the means by which business services perform persistence operations, enabling applications to store and receive data from common data stores, including RDBMS, WebSphere MQ, Java Connect Architecture (JCA), CICS, and IMS. The Persistence Server component offers an XML-based meta model, a set of object-oriented design tool plug-ins, an extensible code generator, and advanced plug-in connectors. For more information, see "Chordiant Persistence Server" on page 181 and the Business Component Generator section of the *Chordiant 5 Foundation Server Application Components Developer's Guide*.

Figure 2-12 illustrates the ***logical*** representation of the Persistence Server layer.

Note: In a deployed model, the Persistence Server, EJBs, and servlets all typically reside in the same JVM.

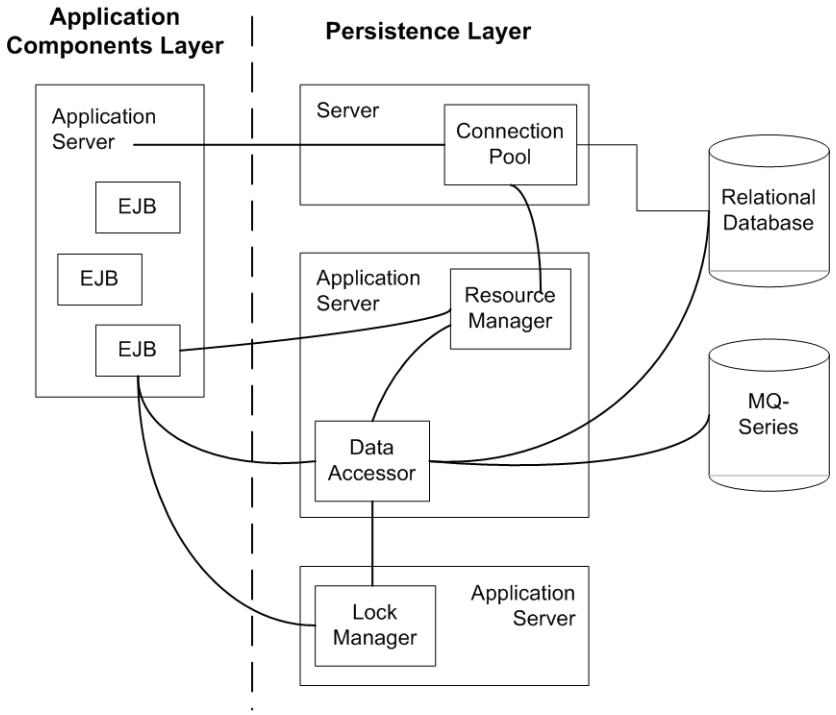


Figure 2-12: Logical Representation of Persistence Server Architecture

SINGLE-BEAN ARCHITECTURE

Chordiant 5 Foundation Server is based on a single-bean architecture. The JX bean is just like any other stateless session EJB and can co-exist and interact with any other EJBs you might have.

The JX single-bean, single-interface architecture is simple and powerful. With it, you can create and change services and data without affecting the distributed interface. Once you have integrated with the JX architecture, you have access to *all JX* services, including additional JX services you might create.

Communication

You can communicate between the JX EJB and other EJBs through JNDI lookup and standard bean-to-bean communications. The interface that you can access from the external EJB can be XML, object-oriented, or typed object-oriented. Once you are communicating with the JX EJB, you have access to all of the related JX services.

Services

JX services are Java classes that run as attributes of the JX EJB. Depending on the configuration, several (or all) services run as individual instances under each JX EJB instance. The JX EJB acts as a dispatcher for the services. A JX service essentially runs as an EJB, including having all EJB interfaces (like ejb_create and ejb_passivate) forwarded to it.

Pooling

The JX EJB can be pooled just like any other EJB. The underlying services are also pooled along with the JX EJB pools.

Assume that you have a single replicate (JVM) of the Application Server that includes a JX EJB, defined to be in a bean pool of 10. Within the JX XML configuration file, you have also defined two services: Service A and Service B.

In this case, there is a total of 30 class instances running in the single Application Server replicate, partitioned as:

- 10 instances of Foundation Server EJB (since bean pool size defined to be 10)
- 10 instances of Service A (run as instances under each Foundation Server EJB)
- 10 instances of Service B (run as instances under each Foundation Server EJB)

Note: There are other possible deployment configurations such as node-specific services and singleton services, but this example is the most basic (and most recommended) stateless service deployment configuration.

SessionContext

Since JX services run as EJBs, they also have a SessionContext attribute, just like an EJB. The SessionContext attribute is available to the JX service for general purpose use, such as obtaining a user transaction within a Bean Managed Transaction (BMT) service.

Transactional Disposition

The JX EJB is a SessionBean that is deployed as both a Bean Managed Transaction (BMT) and a Container Managed Transaction (CMT). For more information on transactions, see "Transactions with the JX EJB" on page 20.

TRANSACTIONS WITH THE JX EJB

In Chordiant 5 Foundation Server, the single JX EJB is deployed twice:

- once as a Bean Managed Transaction (BMT) EJB
- once as a Container Managed Transaction (CMT) EJB. Specifically, as a CMT EJB with the methods trans-attribute set to "Required".

Both BMT and CMT are defined by J2EE. You can use both types of transactions when designing your Foundation Server implementation. Each service within your solution can only use one transaction type—either BMT or CMT.

Background Information

Both Bean Managed Transactions and Container Managed Transactions are defined by J2EE. They are not specific to Chordiant. This section provides you with some general information on BMTs and CMTs. Chordiant-specific information begins with "Transactions in Chordiant Foundation Server" on page 118.

Bean Managed Transactions

Bean Managed Transactions (BMT) are handled manually within the Java code of the EJB using the J2EE UserTransaction interface.

Note: Bean Managed UserTransactions can *only* be active within a single EJB instance and will *not* span calls to other EJB instances. This is significant for JX services because JX services communicate across EJB instances when they interact with each other (for example, through client agents).

Contrast with "Container Managed Transactions" on page 22.

Example of Bean Managed Transaction

Code Sample 2-1 provides an example of code using the J2EE UserTransaction interface.

```
public String MyPublicBMTEJBFunction(String inputData)
{
    javax.naming.InitialContext initialContext = null;
    javax.transaction.UserTransaction myTransaction = null;
   try
   {
       myTransaction = myEJBSessionContext.getUserTransaction();
       myTransaction.begin();
       // Do XA compliant tasks here (i.e. JDBC, JMS, MQ, …)
       // across one or more XA compliant drivers/servers.
     myTransaction.commit();
   }
   catch (Throwable e)
   {
       myTransaction.rollback();
   }
}
```

Code 2-1: Using the J2EE UserTransaction Interface

For More Information

For more detailed information on bean managed transactions, refer to the following online documentation:

- http://java.sun.com/products/ejb/docs.html#specs
- http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Transaction4.html#63068

If these specific links do not work for you, go to http://java.sun.com/products/jta/ or go to http://java.sun.com and search for JTA (Java Transaction API).

For more information on transactions in Chordiant 5 Foundation Server, refer to "Performing Transactions" on page 211.

BMT Deployment

The JX EJB is automatically deployed as both a BMT and a CMT. Code Sample 2-2 shows the relevant sections of the code for the BMT Deployment Descriptor, for your reference. You can view the graphical interface for the full deployment descriptor for your application server. In your development environment, open `ejb-jar.xml`.

```
<?xml version="1.0"?>
<!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 id="EJBJar_1055181543930">
    <display-name>BMT</display-name>
   <enterprise-beans>
      <session id="Session_1055181544055">
       <ejb-name>EJBGatewayServiceBMT</ejb-name>
       <home>com.chordiant.service.ejb.EJBGatewayServiceHome</home>
       <remote>com.chordiant.service.ejb.EJBGatewayService</remote>
       <ejb-class>com.chordiant.service.ejb.EJBGatewayServiceBean</ejb-class>
       <session-type>Stateless</session-type>
       <transaction-type>Bean</transaction-type>
        ...
      </session>
   </enterprise-beans>
</ejb-jar>
```

Code 2-2: BMT Deployment Descriptor

Container Managed Transactions

Container Managed Transactions are used implicitly by the EJB. They are not referenced directly by the EJB Java code, but rather are controlled by the J2EE "container" (the application server) upon call/return from the EJB.

Unlike BMT UserTransactions, CMT transactions can be active/dependent across one or more CMT EJB calls/instances. So you can use CMT transactions across multiple CMT EJBs to process a business service call sequence in a single J2EE transaction.

For More Information

For more detailed information on container managed transactions, refer to the following online documentation:

- http://java.sun.com/products/ejb/docs.html#specs
- http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Transaction3.html#62910

If these specific links do not work for you, go to http://java.sun.com/products/jta/ or go to http://java.sun.com and search for JTA (Java Transaction API).

Example of Container Managed Transaction

Figure 2-13 illustrates Container Managed Transactions.

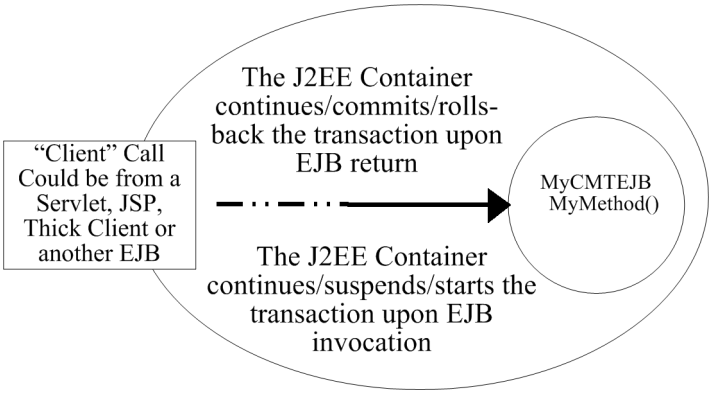


Figure 2-13: Container Managed Transaction

Note: With respect to JX client agent/service architecture, a service communicating to another service through a client agent will always communicate from one instance of the JX EJB to a different instance of the JX EJB. This mechanism follows J2EE standards for EJB communication.

CMT Deployment

The JX EJB is automatically deployed as both a bean managed transaction (BMT) and a container managed transaction (CMT). Code Sample 2-3 shows the relevant portions of the code for the CMT Deployment Descriptor, for your reference. The relevant sections are in bold. You can view the graphical interface for the full deployment descriptor for your application server. In your development environment, open `ejb-jar.xml`.

```
<?xml version="1.0"?>
<!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 id="EJBJar_1055181442577">
    <display-name>CMT</display-name>
   <enterprise-beans>
      <session id="Session_1055181443514">
       <ejb-name>EJBGatewayServiceCMT</ejb-name>
       <home>com.chordiant.service.ejb.EJBGatewayServiceHome</home>
       <remote>com.chordiant.service.ejb.EJBGatewayService</remote>
       <ejb-class>com.chordiant.service.ejb.EJBGatewayServiceBean</ejb-class>
       <session-type>Stateless</session-type>
       <transaction-type>Container</transaction-type>
        ...
   <assembly-descriptor id="AssemblyDescriptor_1">
      <container-transaction id="MethodTransaction_1">
        ...
           <trans-attribute>Required</trans-attribute>
      </container-transaction>
   </assembly-descriptor>
</ejb-jar>
```

Code 2-3: CMT Deployment Descriptor

CMT "trans-attribute" Options

J2EE defines several options for the "trans-attribute" setting on individual EJB methods. For Chordiant 5 Foundation Server, the only trans-attribute option is Required, as shown in the deployment descriptor code starting on page 24.

Although only the Required trans-attribute is supported in this release, here are all of the J2EE-defined options, for your reference:

- **Required —** If existing transaction is present then continue it, otherwise create new transaction. (Supported in this release.)
- **RequiresNew —** Always create new transaction. Existing transaction, if present, is suspended.
- **Mandatory —** Throws exception if existing transaction is not present.
- **NotSupported —** Existing transaction, if present, is always suspended. No new transaction is created.
- **Supports —** Continues existing transaction. Will not create a transaction if existing transaction is not present.
- **Never —** Throws exception if existing transaction is present.

Note: CMT EJBs can not access the UserTransaction interface. Only BMTs can access the UserTransaction interface.

MESSAGE DRIVEN BEANS

Chordiant 5 Foundation Server includes message driven beans (MDBs) for processing JMS messages asynchronously. These MDBs are part of the J2EE standards. The MDBs are used mainly for queuing and routing, as well as passing messages asynchronously in the Chordiant Event Server.

Refer to the following documentation for additional information:

- *Chordiant 5 Foundation Server Business Process Server Developer's Guide* for details on queuing and routing.
- Chapter 10, "Chordiant Event Server" for details on asynchronous messaging.

With the inclusion of MDBs, along with EJBs, the startup order of beans is especially important. Continue reading the next section, "Startup Order of Beans", for more information.

STARTUP ORDER OF BEANS

The startup order of beans is important, with the inclusion of CMT and BMT EJBs and Message Driven Beans. The CMT bean is responsible for setting up the static helpers and custom objects. If the CMT bean is not started first, custom objects and services which depend on it will not be able to function. The queue-based MDBs rely on the queue service to process their messages, as soon as the MDBs start up. But the queue service cannot function if the CMT has not yet started.

The development environment is set up with this order, and the Application Packaging Manager (APM) is set up to deploy in this order as well. You do not need to take any action on this order, but you should be aware of it — especially if you plan to add any custom objects or custom EJBs to your solution. Table 2-1 lists the proper startup order, along with some comments.

The beans must be loaded in this order because beans in later groups may either depend on or affect the beans listed before them.

ORDER	BEAN NAME	DESCRIPTION
1	CMT	Must be started first. Custom objects that are started when this EJB is created will try to reference the CMT EJB. If it is not there, the custom objects cannot function. CMT-based services, such as the queue service, also reference the CMT EJB.
2	BMT	Must be started after the CMT EJB is started.
Start the Topic listener MDBs after the CMT and BMT EJBs (described above) and before the Queue listener MDBs (described after this section), because messages processed by the Queue listener MDBs might alter state that must be shared by the Topic listener MDBs.		
3	QueueAdminTopicMDB	Manages JMS Queue state for the Queue Service across a J2EE cluster.
4	SessionTopicMDB	Shares UserSession state for the Session Service across a J2EE cluster.
5	UserProfileTopicMDB	Shares security state for the UserProfile Service across a J2EE cluster.
Start the Queue listener MDBs after the CMT and BMT EJBs and the Topic listener MDBs (all described above), because messages processed by the Queue listener MDBs might alter state that needs to be shared by the Topic listener MDBs and the messages processed by the Queue listener MDBs might need to call the client agents supported by the CMT and BMT EJBs. Note that only one MDB listening to a Queue is delivered a given message (point-to-point), even though there may be many such listeners in a J2EE cluster. All MDBs listening to a Topic get every message that is broadcast across a J2EE cluster.		
6	JXEMessageInboundQueueMDB	Processes messages for the Chordiant Event Server.
7	SessionQueueMDB	Processes session availability event messages from the Session service to the Queue Service, which may result in a QueueItem's getting pushed to an available session.

Table 2-1: Startup Order of Beans

ORDER	BEAN NAME	DESCRIPTION
8	SystemPullQueueMDB	Used by the Queue Service to asynchronously process QueueItems that have been injected by the queue, requeue, and transfer APIs.
9	SystemPushQueueMDB	Used by the Queue Service to asynchronously process QueueItems that have been injected by the route and reroute APIs.

Table 2-1: Startup Order of Beans (Continued)

Chapter 3

Life Cycle of a Foundation Server Application

Applications and services built using the Chordiant 5 Foundation Server follow an established life cycle that enables an orderly startup and shutdown of dependent software and services.

This chapter describes the details of using a client agent. You do not need to be aware of these details if you are using client agents. You might find them interesting if you are building client agents.

Client Application Startup and Shutdown

Thick Client Application

For an application to start and execute successfully, the system completes the following series of activities:

Note: For details on sections of this example, refer to "StaticHelper" on page 45 and "GatewayHelper" on page 60.

---

1. The JX infrastructure and system are set up through the FatClientStaticHelper's setup method.

Note: This must happen once for each Java Virtual Machine in an application's main routine.

---

2. The client application authenticates using a username and password.

   The client application uses the Security service to complete the authentication.

3. The client application *optionally* initializes its Network Presence through the GatewayHelper. This enables the client to accept callbacks.

4. The client application requests and receives one or more client agents from the ClientAgentHelper. The ClientAgentHelper might optionally initialize the client agent at this time.

5. While running, the client application interacts with the client agents.

6. When done, the client application *optionally* disables its Network Presence through the GatewayHelper.

7. The JX infrastructure and system shut down with the FatClientStaticHelper.shutdown method.

Thin Client Applications

Thin client application startup and shutdown is different from that of thick client applications in these ways:

- All Foundation Server static helpers are automatically set up and shut down in the web/EJB containers.
- Thin client authentication is handled through the Foundation Server login servlet. See "Using the Login Helper" on page 311 for more information.
- Thin client network presence is not established in the web/EJB container, but rather in the browser as a Java applet. See "The RegisterNetworkPresence Class" on page 306 for more information.

Thick Client to Service Interactions

The client application accesses services using an intermediary called the client agent. Figure 3-1 illustrates the interaction between the application and client agent.

Note: For details on sections of this example, refer to "ClientAgentHelper" on page 61 and "Building a Client Agent" on page 134.

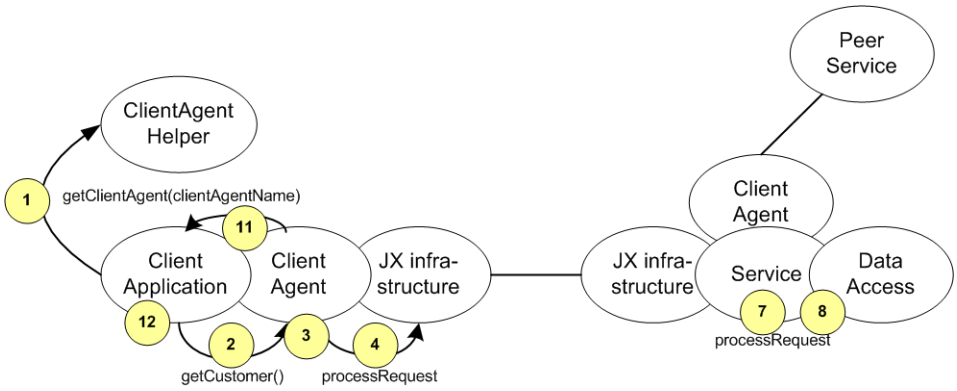


Figure 3-1: Thick Client to Service Interactions

In the process of the thick client application's interacting with the client agent, the system completes the following series of activities. Steps shown in italics are performed automatically by the JX infrastructure.

1. The thick client application requests a client agent from the ClientAgentHelper.
2. The thick client application invokes a method on the client agent that corresponds to a business operation on the service.
3. The client agent assembles the payload.

   The payload is the data communicated between the client application and the service. The data is required by the service operation to complete the business functionality requested by the client.
4. The client agent invokes the processRequest method in the client agent base class.
5. *The JX infrastructure invokes the configured client communication protocol for the service (sockets, RMI, or IIOP, for example).*
6. *The JX infrastructure invokes the JX service.*
7. The service processes the request.

   In the process, it applies the encoded business logic of the organization and performs data access operations (Chordiant Persistence Server), as required.
8. The service returns the results payload.
9. *The JX infrastructure returns the result through the configured server protocol.*
10. *The JX infrastructure returns the results to the client agent.*
11. The client agent returns the results to the client application.
12. The thick client application resumes control and continues execution.

SERVICE TO SERVICE AND THIN CLIENT TO SERVICE INTERACTIONS

Service to service interactions typically involve a service calling a peer to perform some useful work. These interactions assume the same general process flow as between a client application and service, modified to take into account special requirements of server-based software. Transactions (specifically Container Managed Transactions, or CMTs) can live across one or more service to service calls, however nested transactions are not allowed. See "Transactions with the JX EJB" on page 20 and documentation on J2EE Java Transaction API (JTA) for more information.

Thin client to service interactions are the same as service to service interactions because servlets/JSPs typically reside in the same JVM as the service they are connecting to, just as services reside in the same JVM as their peers.

Note: For details on sections of this example, refer to "ClientAgentHelper" on page 61 and "Implementing a Service to Service Call" on page 166.

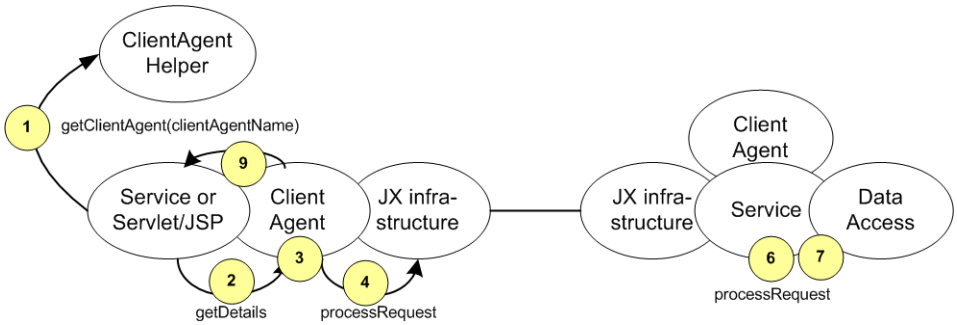


Figure 3-2: Service to Service or Thin Client to Service Interactions

In the process of a service or thin client interacting with a *target* service, the system completes the following series of activities. Steps shown in italics are performed automatically by the JX infrastructure.

1. The originating service or thin client application calls the ClientAgentHelper to get the client agent for the target service.
2. The originating service or thin client invokes the returned client agent.
3. The client agent assembles the payload.
4. The client agent invokes the processRequest method in the client agent base class.
5. *The JX infrastructure invokes the customized target service without performing a transform.*
6. The target service processes the service request.

   This involves applying the implemented business logic, and performing data access operations (Chordiant Persistence Server), as required.

7. The target service returns the result.
8. *The JX infrastructure returns the result to the client agent of the thin client or service.*
9. The target service client agent returns the result.
10. The originating service or thin client regains control and continues execution.

SERVICE TO CLIENT INTERACTIONS

Thick Client Scenario

A service might be required to interact with a client agent associated with the client application. It might do this, for example, when performing a callback to the client application, requesting it to perform some work, such as doWorkflowActivity or getMail.

Figure 3-3 illustrates the interaction between a service and the client agent of a client application. Processes completed automatically by the JX infrastructure are not displayed in the figure and are shown in italics.

Note: For details on sections of this example, refer to "GatewayHelper" on page 60, "ClientAgentHelper" on page 61, and "Implementing a Callback" on page 159.

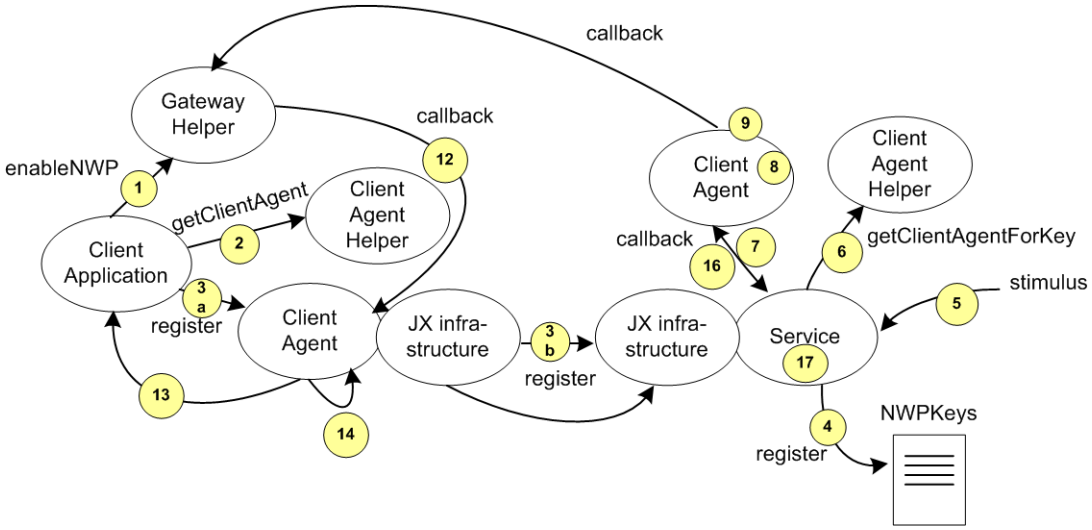


Figure 3-3: Service to Client Interactions in Thick Clients

In the process of a service interacting with a client, the system completes this series of activities:

1. The client application enables its Network Presence through the GatewayHelper.

   The Network Presence offers a mechanism for the client application to be network addressable and accept callbacks from the service. The protocol used by the network presence is configurable (RMI or sockets).

   The client application receives a Network Presence Key (NWPKey). The NWPKey is an arbitrary string that is unique in the namespace and can be registered with any services that need to perform callback functionality.

2. The application gets a client agent from the ClientAgentHelper.
3. The client application registers with a specific client agent (a) and service (b) using the NWPKey.

   The client agent must implement a callback function such as processCallback.

4. The service retains the registered NWPKey list.

   A service that needs to perform callbacks must have access to the NWPKeys for the clients it intends to call back.

5. On the server, some stimulus occurs, affecting the service.
6. The service calls the getClientAgentForKey method, specifying the NWPKey for the required client agent, and receives a client agent.
7. The service calls the returned client agent to perform a callback.
8. The client agent on the server assembles the payload.
9. The client agent on the server invokes the processRequest method of the client agent base class.
10. *The JX infrastructure invokes the configured protocol for the callback on the server.*
11. *The JX infrastructure invokes the callback routine,* processCallback.
12. The Gateway dispatches the callback to the client agent.
13. The client agent on the client posts an event (or similar) that is of interest to the application.

    For example, an application could post a mail arrival event for a specific class of application.

14. The customized client agent returns the result as part of the payload.
15. *The JX infrastructure returns the result to the client agent on the server.*
16. The customized client agent on the server returns the result to the service.
17. The service regains control and resumes execution.

Thin Client Scenario

Similar to the thick client architecture, the thin client architecture enables a service to interact with a specific thin client application running in the browser by using the NetworkPresenceApplet. Data is passed from the service to the NetworkPresenceApplet and then passed on to a JavaScript function specified by the thin client application—the Custom JavaScript Callback Handler. This JavaScript function can do any required processing and then return a result value that is returned to the service.

Figure 3-4illustrates the interaction between a service and a thin client application with network presence. Processes completed automatically by the JX infrastructure are not displayed in the figure and are shown in italics below.

This example scenario shows the Custom Callback JavaScript Handler in the browser performing local processing or making a synchronous request back to the Servlet/JSP layer before it returns to the JavaScript infrastructure—and therefore back to the callback originator on the server side. In practice, the Custom Callback JavaScript Handler might process the callback in any manner you desire, however the JavaScript processing should be non-blocking, as the call from the service is synchronous.

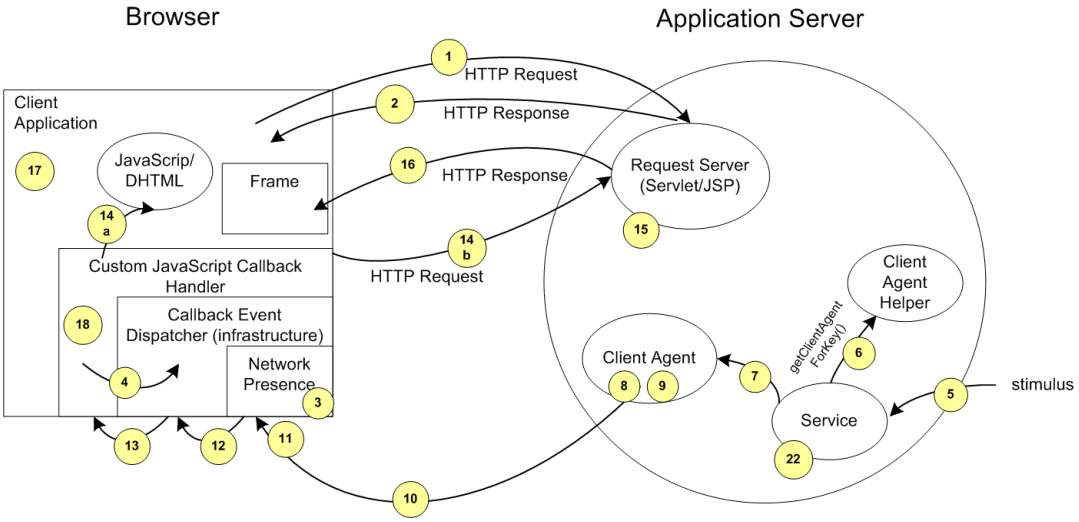


Figure 3-4: Service to Client Agent Interactions in Thin Clients

1. A browser sends an HTTP request to the Request Server.
2. The Request Server returns an HTTP response to the browser with HTML markup for the Network PresenceApplet frame.
3. The browser is initialized and loaded with Chordiant thin-client Network Presence components shown in the figure. The Network Presence Key (NWPKey) is created and registered in JNDI through the RegisterNetworkPresence servlet.
4. The Custom JavaScript Callback Handler is registered with the JavaScript infrastructure, typically when the HTML page is loaded (through the onLoad mechanism).

5. On the server, some stimulus—such as an incoming email—occurs, affecting the service.
6. The service calls the getClientAgentForKey method, specifying the NWPKey for the required client agent, and receives a client agent.
7. The service calls the returned client agent to perform a callback.
8. The client agent on the server assembles the payload.
9. The client agent on the server invokes the processRequest method of the client agent base class.
10. *The JX infrastructure invokes the configured protocol for the callback on the server.*
11. *The Network Presence in the browser receives the callback request.*
12. *The Network Presence in the browser passes the callback request to the JavaScript infrastructure.*
13. The JavaScript Infrastructure dispatches the information to the Custom JavaScript Callback Handler.
14. Once the Custom JavaScript Callback Handler has the information, it can process it in any appropriate manner. This scenario includes two examples:
    a. The browser can handle the callback information on the browser itself, using JavaScript and DHTML. Skip ahead to Step 18 on page 36.
    b. The browser can send an HTTP request back to the application server for server-side processing. The frame for the response must be specified within this HTTP request.
15. The server processes the HTTP request.
16. The server sends an HTTP response back to the specified frame on the browser.
17. The application receives the HTTP response.
18. Control is returned to the Custom JavaScript Callback Handler.
19. *Control is returned to the JX JavaScript infrastructure.*
20. *The JX Network Presence writes the response.*
21. *The JX infrastructure returns the result to the ClientAgent of the caller.*
22. The ClientAgent returns a result and the caller regains control.

Chapter 4

Managing State in JX Services

Chordiant 5 Foundation Server supports these models for services:

- "Stateless Service", including:
  - Multi-Instance
  - Multi-Instance, Central Persistent
- "Stateful Services", including:
  - Single Instance, Multi-Threaded
  - Single-Instance, Multi-Threaded, Persistent
  - Multi-Instance, State Propagated

STATELESS SERVICE

Stateless service means that a software program cannot take information about the last session into the next, such as settings the user made or conditions that arose during processing.

Stateless service in Foundation Server supports both horizontal and vertical scalability. This means that a program can be replicated multiple times on one system, or on many systems. Foundation Server utilizes the automated load balancing and failover capability of J2EE.

Dynamic state is not allowed in stateless service, but static state can be cached in memory. In static state, data can be viewed but not changed. An example of static state might be a lookup table of zip codes. An example of a dynamic state might be a customer's account data.

Stateless service also supports fault tolerance; if one replicate fails, other replicates immediately pick up the load, ensuring uninterrupted service and no loss of data. This is the preferred service design and deployment model.

Chordiant supports these two stateless models:

- "Multi-Instance Model"
- "Multi-Instance, Central Persistent Model"

Multi-Instance Model

Figure 4-1 illustrates a stateless model that has several replicates of a service, each of which might have static state.

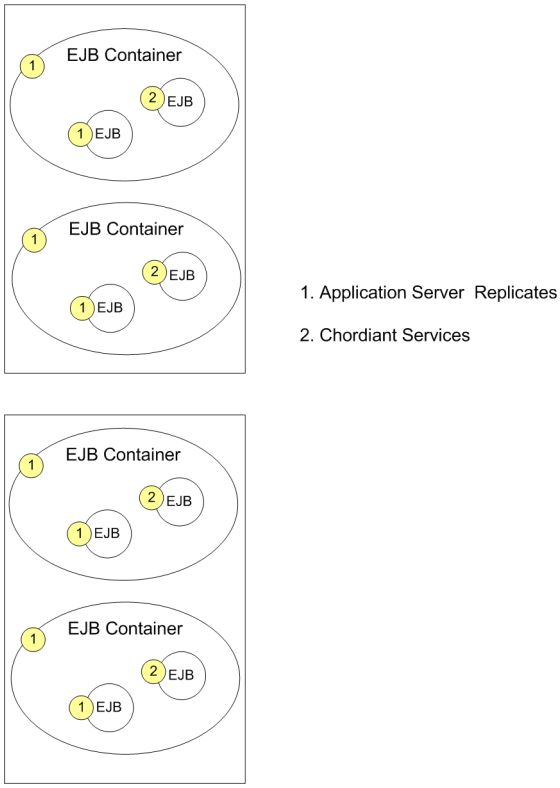


Figure 4-1: Stateless, Multi-Instance

This stateless service model is both horizontally and vertically scalable, so load balancing can be achieved. In the event a replicate fails, any of the replicates can pick up the load, insuring uninterrupted service. Since all source data is static, there is no loss of data when a replicate goes down.

JX Model

In the JX architecture, the multi-instance model can be accommodated using a stateless JX service.

Multi-Instance, Central Persistent Model

The model shown in Figure 4-2 shows multiple instances of the service, as in the first model. However, dynamic state is written to a database. Synchronization for this data can be handled by the database's record-locking, or other appropriate, mechanism, such as the Chordiant LockService.

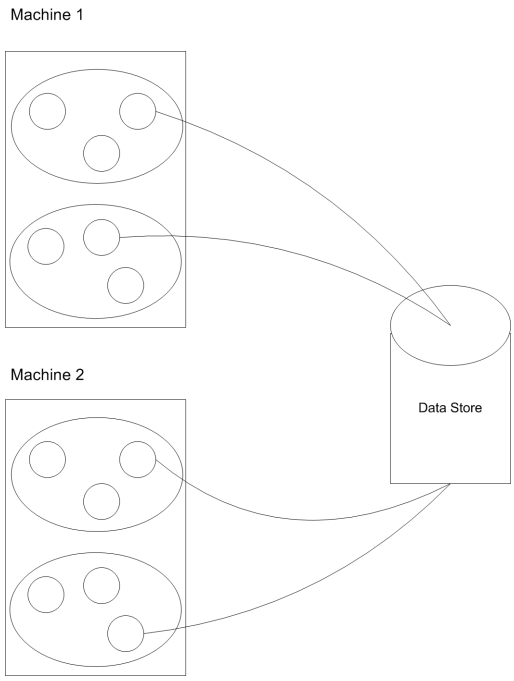


Figure 4-2: Stateless, Multi-Instance, Central Persistent

The multi-instance, central persistent stateless model offers the same advantages as the multi-instance stateless model with the added advantage of being able to utilize dynamic state. However, the need to constantly access the database can affect performance.

JX Model

In the JX architecture, the multi-instance, central persistence model can be accommodated using a stateless JX service in combination with Chordiant Persistence Server (JXP) for database access.

STATEFUL SERVICES

Stateful services enable utilization of dynamic state.

Note: Although there is currently no built-in JX support for stateful services, the following stateful models are provided as a guide for developers who want to create their own.

- "Single Instance, Multi-Threaded Model"
- "Single-Instance, Multi-Threaded, Persistent Model"
- "Multi-Instance, State Propagated Model"

Single Instance, Multi-Threaded Model

Figure 4-3 illustrates a single instance of a service in the entire J2EE application server namespace. The service can be accessed by multiple users at the same time. Dynamic state must be managed using an appropriate synchronization mechanism.

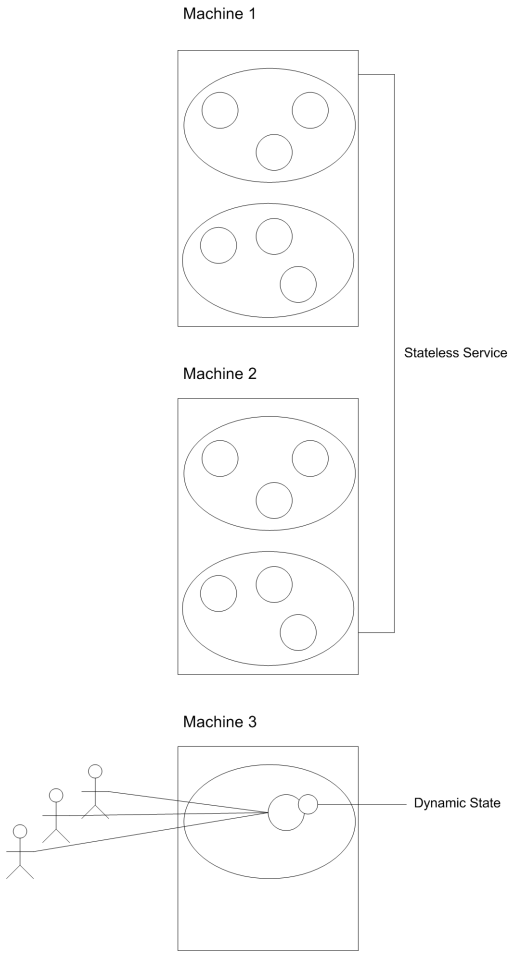


Figure 4-3: Stateful Model, Single Instance, Multi-threaded

A multi-threaded stateful service can work very well if the percentage of utilization is balanced accordingly, in relation to the rest of the services. Since there can be only one instance of the service, over-utilization can result in reaching a saturation limit that affects efficiency. This type of service can only be as fast as the machine on which it is running, and can only scale to the CPU capacity of that machine.

The disadvantage of this stateful model is that if the process fails, users cannot access the service until it is restarted, and all dynamic state that was held in memory will be lost. Developers can modify this model so that data is written to a database. Then, in the event of a failure, the cached

data last saved can be restored. High Availability software can also be implemented to act as a watchdog to restart the service. Then the service can restore its dynamic state, minimizing the amount of user down time.

JX Model

In the JX architecture, the single instance, multi-threaded model can be accommodated using a combination of one or more JX stateless services and a JX singleton CustomObject configured on a specific application server replicate.

In this case, the stateless front-end, or *facade*, JX services pass requests through to the singleton JX CustomObject. Here, the JX CustomObject could be an RMI object which is configured to run centrally on one of the Application Server replicates. The stateless JX facade services can look up this RMI object in JNDI.

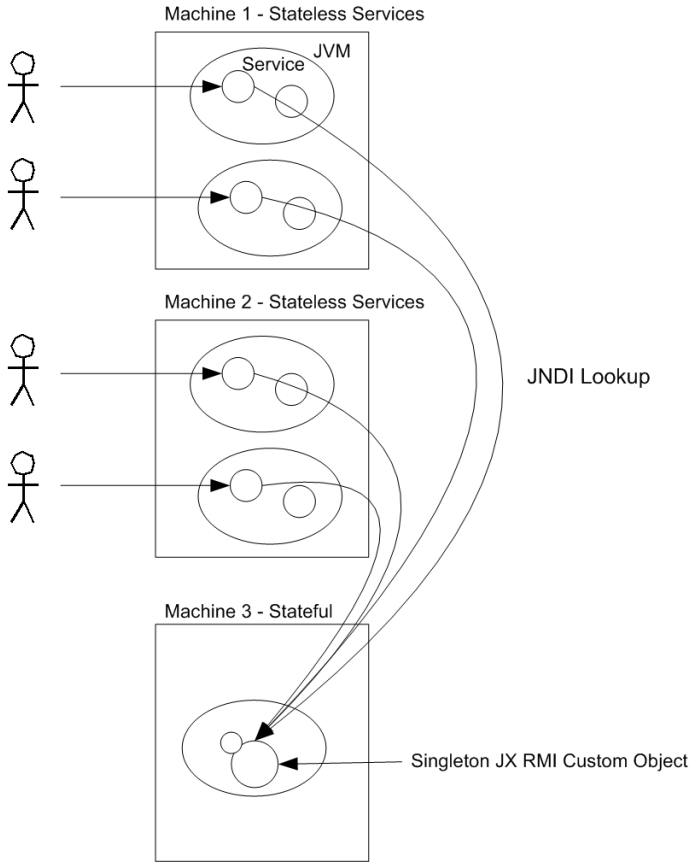


Figure 4-4: JX Version of Stateful Model, Single Instance, Multi-threaded

Single-Instance, Multi-Threaded, Persistent Model

The stateful model shown in Figure 4-5 is similar to the previous model. However, this model also provides fault tolerance because dynamic state is periodically written to a database. Therefore, if the process fails, the dynamic state can be restored from the database once the process is restarted. In terms of performance, the single-instance, multi-threaded, persistent service model sacrifices some speed to access the database.

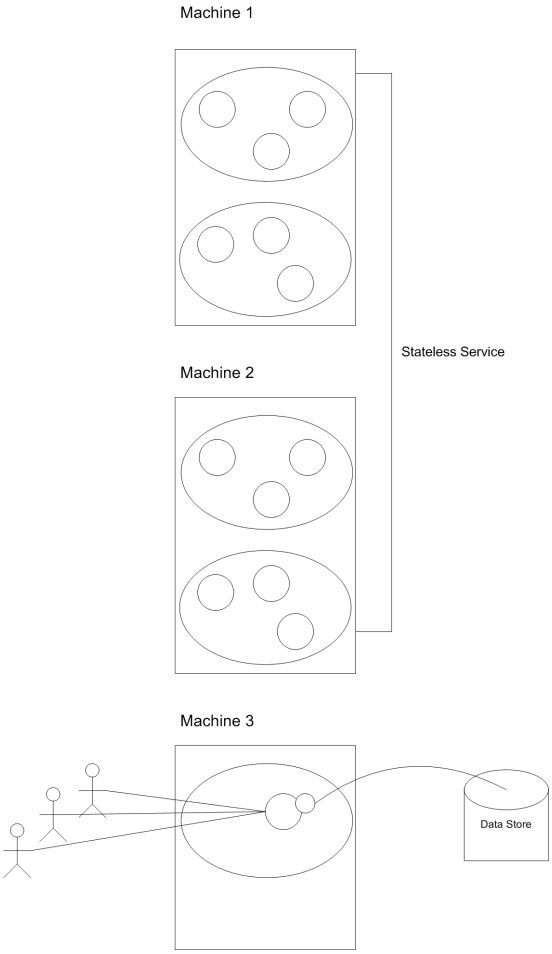


Figure 4-5: Single Instance, Multi-Threaded, Persistent

JX Model

In the JX architecture, the singleton instance multi-threaded persistent model can be accommodated using a combination of the single instance, multi-threaded model (described on page 41) and Chordiant Persistence Server (JXP) for database access.

Multi-Instance, State Propagated Model

The stateful, multi-instance service model, shown in Figure 4-6, enables multiple instances of a service with dynamic state held in memory. Statefulness is managed by dynamically propagating each instance's dynamic state to all of the other instances. The multi-instance, state-propagated service supports scalability, load balancing and fault tolerance.

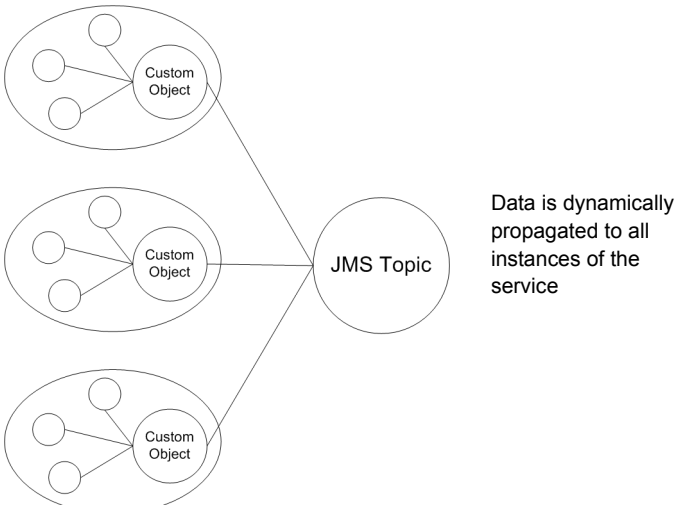


Figure 4-6: Stateful, Multi-Instance, State-Propagated

If one of the processes fails, the other instances will continue the service uninterrupted, as in the stateless, multi-instance model. And, because dynamic state is propagated to the other instances, no loss of data occurs. The service can be replicated as many times as needed, both horizontally and vertically.

The multi-instance, state propagated service model works well provided the amount of data that has to be propagated is not excessive. There is also some latency propagating the data to all instances, and large amounts of data being written to many instances will affect performance and efficiency.

JX Model

In the JX architecture, the multi-instance, state propagated model can be accommodated using stateless JX services in combination with some distributed event mechanism, such as Java Message Service (JMS). JX CustomObjects or J2EE message beans can be used to receive the JMS events.

Chapter 5

Chordiant 5 Foundation Server Helpers

The Chordiant 5 Foundation Server includes several utilities, or helpers, to assist you in creating your Chordiant 5 solution. These helpers assist with functions including logging, configuration, setup and shutdown.

StaticHelper

The StaticHelpers call the other helpers in the proper order to properly set up and maintain your environment.

There are two StaticHelpers to set up the appropriate JX infrastructure for the container:

- **application server —** for J2EE application servers, such as WebSphere or WebLogic.
- **thick client —** for Swing-based thick client JVM containers

Note: Within a J2EE Application Server, such as WebSphere or WebLogic, the StaticHelper is automatically set up and shut down by the Chordiant infrastructure. There is nothing to do for the Servlet/JSP, JX CustomObject, or JX Service Developer.

The StaticHelper has one method with different commands. They are described for the FatClient, but similar commands are available for the application server.

- The **SETUP** command is used at the beginning of a program. It calls the other helpers described in this chapter in the proper order to prepare your environment for use.

```
public static final String FatClientStaticHelper.serviceControl(
    StaticHelperBaseClass. SERVICE_CONTROL_COMMAND_SETUP );
```

- The **SHUTDOWN** command is used at the end of a program, just before exit. It shuts down the various helpers in the appropriate order to properly shut down your environment.

```
public static final String FatClientStaticHelper.serviceControl(
    StaticHelperBaseClass.SERVICE_CONTROL_COMMAND_ SHUTDOWN );
```

- The **REFRESH** command is used to refresh the cache of the various helpers.

```
public static final String FatClientStaticHelper.serviceControl(
StaticHelperBaseClass. SERVICE_CONTROL_COMMAND_REFRESH );
```

- The **STATUS** command is used to check the status of the helpers.

```
public static final String FatClientStaticHelper.serviceControl(
StaticHelperBaseClass. SERVICE_CONTROL_COMMAND_STATUS );
```

The StaticHelper base class is in the package com.chordiant.core.StaticHelperBaseClass. It includes:

- com.chordiant.core.FatClientStaticHelper
- com.chordiant.core.ThinClientStaticHelper
- com.chordiant.core.ApplicationServerStaticHelper

CONFIGURATIONHELPER

The ConfigurationHelper, used at runtime, accesses the `master.xml` file and any *`{component}`*`.xml`, *`{nodename}`*`.xml`, and `sitemaster.xml` files and returns information about the configuration the XML configuration file specifies. Refer to "Configuration Files" on page 95 for more information on `master.xml` and other configuration files.

The ConfigurationHelper provides fundamental configuration information and is thus called by other utilities within the Chordiant 5 Foundation Server.

Code Sample 5-1 provides an example of a generic configuration file to show the basic format of these files.

```
<section> name
    <tag> name
        <value> abc </value>
    </tag>
    <tag> name
        <value> def </value>
    </tag>
</section>

<section> name
    <tag> name
        <value> abc </value>
    </tag>
    <tag> name
        <value> def </value>
    </tag>
</section>
```

Code 5-1: Generic {component}.xml Configuration File

The ConfigurationHelper is in the package com.chordiant.core.configuration.ConfigurationHelper.

Configuration Files and the ConfigurationRootDirectory

Configuration files are located in the *`{CHORDIANT_ROOT}`*`/config/Chordiant/` directory of either the development or production environment. *`{CHORDIANT_ROOT}`* corresponds to the chordiant.configuration. configurationRootDirectory parameter in your application server. You must either have your configuration files in this directory, or point the ConfigurationRootDirectory parameter to the location of your configuration files. In a production environment, these directories are located within the EAR file directory.

ConfigurationHelper Methods

ConfigurationHelper includes the following methods:

- **getConfigurationValue**—Use this method, providing the desired section name and tag name, to receive a string with the associated value.

```
public static String getConfigurationValue( String sectionName, String tagName )
```

- **getConfiguration**—Use this method, providing the desired section name, to receive an array of the configuration items (tag names and values).

```
public static ConfigurationItem[] getConfiguration( String sectionName )
```

Code Sample 5-2 illustrates how to retrieve configuration values from an XML configuration file via the ConfigurationHelper.

```
String section = "TestService";
String tag = "classname";
String value = null;
value = ConfigurationHelper.getConfigurationValue( section, tag );
```

Code 5-2: Using ConfigurationHelper to Retrieve Configuration Values

Configuration Refreshing

Whenever you change configuration information, go to the Administrative Console to refresh the ConfigurationHelper. For information on the Administrative Console, refer to Chapter 6, "Chordiant 5 Foundation Server Administration", beginning on page 63.

LogHelper

The LogHelper is used at runtime to output a variety of messages concerning the application or the server. Logging is configured in the configuration XML files.

The LogHelper can be controlled by the service control, so it can be setup, shut down, refreshed, or return a status.

The LogHelper is in the package com.chordiant.core.log.LogHelper.

Logging Interfaces

The following interfaces are included in the LogHelper. Each is based on the package name, class name, and method name where the message is generated and displays a message string.

Error

Error messages are the most serious of all messages. They should always be turned on. They signify that the system is not working as it should and that these problems must be addressed. For example, any communication problems between the client and server are displayed as error messages.

Try to be as precise with your error message text as possible — down to where in the method an exception might have occurred. This will help you track down the error.

There are two kinds of error messages. The first is generated by an error with another program. The second is generated by an error within your own program.

1. Error with another program:

```
public static void error ( String packagename, String classname, String methodname,
    String msg, Throwable th )
```

   Sends an error message (String msg) to the configured log writer, based on the error that prompted the message (Throwable th).

   This Throwable is usually created by an error within a different application which interacts with your application. For example, if you are trying to enter a value in a database, but the database sees that value as invalid, the program accessing the database will throw an exception. That exception will be delivered to the user of the application who generated the error. It will also be sent to the log. In the message text you write, try to explain the nature of the error to the user, although it may not necessarily be a problem with your own program.

   You might also choose to call this error message from within your Chordiant-based application if you think the Throwable error information will be helpful for the user.

2. Error with your own program:

```
public static void error ( String packagename, String classname, String methodname,
    String msg )
```

   Sends an error message (String msg) to the configured log writer. The standard log writer is standardout. Refer to page 54 for more information.

   This error is logged for your own Chordiant-based program.

Warning

Warning messages are not as severe as errors. Warnings indicate that some known and repairable condition has occurred, but the system is still stable. They are usually not turned on for production.

```
public static void warning ( String packagename, String classname, String methodname,
    String msg )
```

Info

Informational messages are basic configuration and system settings messages. These are typically hit only once during startup. They are usually only turned on during development and not during production.

```
public static void info ( String packagename, String classname, String methodname,
    String msg )
```

Debug

Debug messages are basic messages letting you know where and what the program is executing. These can include parameter values in methods, operational state, and could have very large messages or complex message construction.

Debug messages are usually turned on only during initial testing and possibly during error recreation or error tracking. They are not used during production.

Because debug messages can be very large, they are controlled by isDebugLogOn. For more information, refer to "isDebugLogOn" on page 52.

```
public static void debug (String packagename, String classname, String methodname,
    String msg)
```

MethodEntry / MethodExit

Entry and exit messages show entry and exit points for methods. They are similar to info messages, but can be treated differently in the log. They are usually only used during development, error tracking, error recreation, or testing. They are not used during production. Notice there is no text message. This message only shows the entry and exit for a method, without further annotation.

```
public static void methodEntry ( String packagename, String classname, String methodname)
```

```
public static void methodExit ( String packagename, String classname, String methodname )
```

Performance

Performance logging is helpful when you are tuning your application. Performance logging is usually used only during development and testing. It is not used during production. Place performance statistics calls within your code to receive logged messages about the start and end times of your method call, its duration, and whether there were any errors.

Foundation Server has already instrumented performance logging on the base client agent and EJBs through distributed auditing. By turning on distributed auditing and setting proper filters, you can obtain the performance information for any function calls on any service. For details on distributed auditing, refer to "distributedaudit" on page 105. At times, however, it might be necessary to get information for a particular segment of code within a function call. To accomplish that, logPerformanceStatistics calls can be placed wherever desired in your code. With filters properly set, performance statistics for the code segments instrumented are printed out to standardout, or files where you have redirected standardout.

Code Sample 5-3 shows the logPerformanceStatistics method, the primary call you will use for core Foundation Server components, particularly EJBSmartStub and EJBGatewayServiceBean. Additional overloaded methods for using test harnesses are available in the LogHelper class and are described in the associated Javadoc.

```
public static void logPerformanceStatistics(
    String packagename,
    String classname,
    String methodname,
    String bopName,
    long startTime,
    long endTime,
    String userName,
    String serviceName,
    String functionName,
    String inputData,
    String outputData,
    boolean errorFlag)
```

Code 5-3: Performance Logging Call within a Method

where:

packagename = full source package name

classname = full source class name

methodname = full source method name

bopName = name of the Business OPeration, pick a unique one

startTime = starttime of the BOP, in milliseconds

endTime = endtime of the BOP, in milliseconds

userName = username like hmonroe, can be null

serviceName = name of the service

functionName = name of the function called on the service

inputData = serilialized inputdata

outputData = serilialized output data

errorFlag = true if error in operation, false if not.

When you run your code with the performance logging calls, you will receive logging output similar to Code Sample 5-4. The meaningful data starts with <PTH_STATISTICS. You can choose to trim the output to start with <PTH_STATISTICS. That portion of Code Sample 5-4 is shown in bold.

```
<Wed Sep 03 16:35:17 PDT 2003> <1062632117468> <  PERF  > <Thd=P=35240:O=0:CT>
<com.chordiant.core.test.fat tester.logstatistics()>
<PTH_STATISTICS,engsrv14.chordiant.com,hmonroe,0,6666,engsrv14.chordiant.com-nomanager-1234-6666,Login,09/03/2003,
16:35:17,09/03/2003,16:35:17,463,0,N>
```

Code 5-4: Output from Performance Logging

The comma-separated fields in the Performance Logging output, as shown in Code Sample 5-4, are:

<PTH_STATISTICS tag
hostname
username
iteration
client number on this machine
client id
BOP name
Start date
Start time
End time
Start time in milliseconds
Duration of the call
Error flag (N means no error)

You can analyze the performance data to evaluate your implementation. Performance logging is intended for studying single threads. Contact your Chordiant consultant for information on performance evaluation.

For additional performance logging through distributedaudit, refer to "distributedaudit" on page 105.

isDebugLogOn

This interface reads the configuration files to find the value of LOG_DEBUG_ON. If it is true, this value is true and you will see debug messages. True is the default value in the master configuration XML file. Since debug messages can be large, this variable enables you to turn them off.

Note: This setting supersedes any settings that you might make for your log filters within the configuration XML file. So although you might have set logging for the debug level, if isDebugLogOn is false, you will not see any debug messages, even though you are filtering for them.

```
public static boolean isDebugLogOn()
```

Suggested use of isDebugLogOn

In this example, isDebugLogOn will inform the method if it should construct a lengthy message for debug logging. If debug logging is turned off in the configuration file, unnecessary message generation can be avoided.

```
if (LogHelper.isDebugLogOn())
{
     for(int n=0; n < 1000; n++)
     {
            message = message + ", " + n;
     }
     LogHelper.debug(PACKAGE_NAME, CLASS_NAME, METHOD_NAME, message);
}
```

Caution When Using isDebugLogOn

Do not include other log messages within the brackets of the if (LogHelper.isDebugLogOn()) section of code. If isDebugLogOn is false, none of the code within the brackets will be run and you will not see any log messages contained within the brackets.

In the code sample below, if DebugLogOn is set to false, you will not see the methodEntry message in the log, even if the methodEntry/methodExit is turned on. To see the methodEntry message in the log, both methodEntry/methodExit and DebugLogOn would need to be turned on.

```
if(LogHelper.isDebugLogOn())
{
     LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
}
```

Logging Configuration

Configuration of logging during runtime is handled in the configuration XML files (mainly *{component}*.xml, but also *{nodename}*.xml, or sitemaster.xml files) under the Log section.

There are three sections within the loghelper.xml file:

1. **Enumeration** — The listing of the active filters which will be described later in the file.
2. **Configuration** — Consists of a single variable, LOG_DEBUG_ON, which controls whether you will see debug messages.
3. **Filters** — Describes the details of the filters enumerated in the first section.

Code Sample 5-5 is an annotated representation of the loghelper.xml configuration file. You can also refer to the loghelper.xml file itself, located in the *{CHORDIANT_ROOT}*/config/Chordiant/components/master directory.

{CHORDIANT_ROOT} corresponds to the chordiant.configuration.configurationRootDirectory parameter in your application server. Refer to "Configuration Files and the ConfigurationRootDirectory" on page 46 for more information.

CODE	COMMENTS
<Section> Log <Tag>log.Filter <Value>FilterOne</Value> </Tag> <Tag>log.Filter <Value>FilterTwo</Value> </Tag> </Section> <Section>LogConfiguration <Tag>LOG_DEBUG_ON <Value>true</Value> </Tag> </Section>	The start of the enumeration section. You can list as many filters as you want. The <Tag> must be "log.Filter" for each <Value>. If you create a *{nodename}*.xml or *{component}*.xml file exists, you can override the log.Filter by using log.Filter within your configuration. Otherwise, you can add filters by using filters with different names. The <Value> must be unique for each filter because it refers to a section later in the configuration file. This section only contains the LOG_DEBUG_ON variable. This value defines what isDebugLogOn will return. By default, it is set to "true" so you will see debug messages if you are filtering for debug messages in a filter section. If the value is set to "false", you will not see any debug messages, even if your filter includes them.

Code 5-5: Representation of loghelper.xml File

CODE	COMMENTS
<Section>FilterOne <Tag>filterclass <Value>com.chordiant.core. log.LogFilter</Value> </Tag>	This describes the FilterOne filter which was introduced earlier in the enumeration section. The filterclass value is the fully-qualified class path of the standard LogFilter class provided by Chordiant.
<Tag>criteria <Value>com</Value> </Tag>	The criteria defines the package.class.method name sequence for which you want to log messages. By default, the value is "com" to catch everything where the package name begins with "com". However, you can define more than one criteria value with different levels of precision, all the way down to the method name. Refer to "Multiple Criteria and Levels" on page 57 for more information.
<Tag>level <Value>error</Value> </Tag>	The level describes which type of messages to log: error, warning, info, debug, entry, and exit. You can have one or more levels defined per criteria. Here, both the error and warning levels are defined.
<Tag>level <Value>warning</Value> </Tag>	
<Tag>writer <Value>com.chordiant.core. log.LogWriterStandardOut </Value> </Tag> </Section>	The writer defines how the messages are processed. By default, the output is to *StandardOut* where messages are printed on your screen, or wherever they are written by your application server.
<Section>FilterTwo <!--same structure as FilterOne--> </Section>	*FilterTwo* has the same structure as *FilterOne*, with different values for the elements.

Code 5-5: Representation of loghelper.xml File (Continued)

Figure 5-1 shows the details of the LogHelper relationships:

- The LogHelper can have many filters.
- Each filter can have many criteria.
- Each criterion can have many levels. The available levels are:
  - error
  - warning
  - info
  - debug
  - method entry/exit
  - perf (for performance statistics)
- Each filter can have many writers.

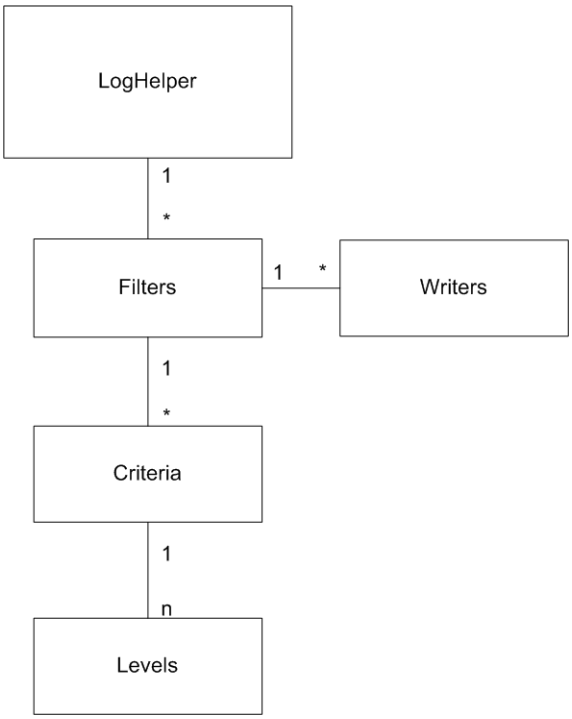


Figure 5-1: Relationships within LogHelper.xml

Creating a New LogFilter

To create a new log filer:

1. Develop your Java Class to implement your own filter logic and implement Chordiant-required Log Filter interfaces.
2. Copy `loghelper.xml` from the *{CHORDIANT_ROOT}*`/config/Chordiant/components/master` directory to *{CHORDIANT_ROOT}*`/config/Chordiant/components/` directory.

   {CHORDIANT_ROOT} corresponds to the chordiant.configuration.configurationRootDirectory parameter in your application server. Refer to "Configuration Files and the ConfigurationRootDirectory" on page 46 for more information.
3. Rename the `loghelper.xml` file in the `/components` directory and update it.

   Replace the default filter class name within the FilterOne section with your own filter class.

```
<Section>FilterOne
       <Tag>filterclass
          <value>com.chordiant.core.log.LogFilter </value>
       </Tag>
       …
</Section>
```

Configuring Message Formatting

You can customize the process method on the log filter to control how your message is formatted. Write you own code in the process method if you want custom behavior.

Criteria Details

Criteria Formatting

When working with filters, be aware of these formatting requirements:

- Criteria values are case-sensitive.
- When defining Criteria down to the method level, do not put the parentheses () on the method name.

  For example, use com.chordiant.bd.custom.myMethod, without any parentheses on the end.

Note: When specifying criteria down to the method level, be aware that the filter works on the "startsWith" algorithm. So in addition to the filter criteria, a path starting with that criteria will also be selected.

For example, if you specify the criteria com.chordiant.temp, you expect that com.chordiant.temp.method will be selected which is the case. However, you must consider that com.chordiant.temporary will also be selected.

Multiple Criteria and Levels

In your XML configuration files, you can specify a single filter with multiple criteria and levels. Here is an example of a filter you might specify.

```
<Section>FilterOne
   <Tag>filterclass
      <Value>com.chordiant.core.log.LogFilter</Value>

      <criteria>com</criteria>
         <level> error</level>
         <level> warning</level>

      <criteria> com.chordiant</criteria>
         <level> info</level>

      <criteria> com.chordiant.core</criteria>
         <level> debug</level>
```

Redundant Levels

Redundant levels do not have any affect on the LogHelper. Once you have listed a level within a filter, you will not receive duplicate messages by repeating the level within that filter.

Redundant Filters

Be aware of redundant filters. If you create several filters with redundant criteria and levels, for example two filters which look for errors in com components, you will receive multiple messages.

Creating a New LogWriter

To create a new log writer:

1. Develop your Java class to implement your own writer logic and implement Chordiant required com.chordiant.core.log.LogWriterHandler interface.
2. If you have not already done so, copy `loghelper.xml` from the *{CHORDIANT_ROOT}*`/config/Chordiant/components/master` directory to *{CHORDIANT_ROOT}*`/config/Chordiant/components/` directory.

   {CHORDIANT_ROOT} corresponds to the chordiant.configuration.configurationRootDirectory parameter in your application server. Refer to "Configuration Files and the ConfigurationRootDirectory" on page 46 for more information.

3. Update the `loghelper.xml` in the `/components` directory.

   Replace the default writer class name value with your own writer class.

```
<Section>FilterOne
    …
    <Tag>writer
        <value>com.chordiant.core.log.LogWriterStandardOut</value>
    </Tag>
    …
</Section>
```

Changing Logging Configuration

If you want to change the filter levels you have defined, change the settings in an XML configuration file. Then use the Chordiant Administrative Console to refresh the ConfigurationHelper, then refresh the LogHelper.

Note: You must refresh the ConfigurationHelper first.

Refer to "Using the Administrative Console" on page 66 for details.

For details on configuring logging, refer to "Configuration Files" on page 95.

Production Environment Settings

In most cases, within the production environment, you will only have error logging on. Other messages can be helpful during development, error tracking recreation, and testing, but they add extra overhead in a production environment.

For a production environment, change your production filter as described here:

- Include only the level Error. Remove all other levels.
- Change the setting of LOG_DEBUG_ON to FALSE. It is set to TRUE by default. This level of logging is not needed for the production environment.
- Set the criteria to com to catch all errors related to the Chordiant 5 Foundation Server.

Calling the LogHelper

Code Sample 5-6 illustrates how to use LogHelper within your code.

```
import com.chordiant.core.log.*;

public static final String PACKAGE_NAME = ”com.chordiant.packageTest”;
public static final String CLASS_NAME = ”classTest”;
public static final String METHOD_NAME = ”methodTest”;
//construct Debug message
for (int i=0; i< 100; i++)
{
if (LogHelper.isDebugLogOn())
{
          message = “Debugging this routine at loop “ + i;
          LogHelper.debug(PACKAGE_NAME, CLASS_NAME,METHOD_NAME, message);
}
.
.
.
}
```

Code 5-6: Using LogHelper

Log File Output

The standard log writer writes the log messages to your screen, or to wherever your application server writes to standardout. Log messages appear in this order:

1. Time Stamp—MMM DD, YYYY HH:MM:SS - date
2. Milliseconds—MSS - long
3. Severity—string (Debug, Informational, Warning, Error, Entry, Exit)
4. ThreadID—string (the process running on the machine at the time)
5. Sub-System Marker—string (The package, class, and method name taken from the interface)
6. Message Text—string (The value of the message you wrote.)
7. Exception Name (optional)—string (The value of the Throwable)

Note: Be aware of management issues with logging. Log files can become quite long if you redirect the output of the LogHelper to a file. If you choose, you can create a batch file to pipe the messages to a file and manage their size.

Log File Output Example

Here is an example of error message output. Note that if you expand the width of the viewing window, this will all be on one long line on your monitor, so all the columns (including time and message type) line up.

```
<Fri Sep 21 14:30:25 PDT 2001> <1001107825160> <  ERROR  > <Thd=clientHandlerThread
[127.0.0.1]/[1598]> <com.chordiant.persistence.test.JXPTestService.processRequest()>
<Exception occurred <java.sql.SQLException: DataSourceRunner.getConnection() ERROR:
Connection pool is empty or all connections are in use> >
```

Multi-Threaded Logging

Log files can become large, and different logging threads become intermingled.

To find all logs for a specific thread, search (GREP) for the ThreadID. If you find an error message, search backward from it to find the events leading up to the error.

GATEWAYHELPER

The GatewayHelper enables the network presence for a client "application" instance. When the GatewayHelper is set up, it creates a callback server to process callbacks and also generates a unique network presence key (NWPKey) which is registered environmentally with the name service of the Application Server. This identifier enables services to connect with the application or client agent through a callback.

- For the **thick client**, the GatewayHelper resides in the client application's JVM.
- For the **thin client**, the GatewayHelper resides on the HTTP client. That is, in the browser.

The GatewayHelper is in the package com.chordiant.service.GatewayHelper.

For the thick client, the main GatewayHelper methods are:

- **enableNetworkPresence**—Generates a network presence ID and creates and sets up a gateway with that presence

  ```
  public static String enableNetworkPresence( String username,
      String authentication ) throws CommandException
  ```

  Here is an example usage of the enableNetworkPresence method:

  ```
  String networkPresenceKey = GatewayHelper.enableNetworkPresence(
      userName,authenticationToken);
  ```

  Notice that the user name and authentication token are required for enabling network presence. This is done for security purposes, ensuring that the user has permission to have network presence. In addition, the user name is used to generate the NetworkPresenceKey.

  You can only assume that the NetworkPresenceKey is unique within the system. Its actual content is subject to change.

- **disableNetworkPresence**—Removes the network presence when it is no longer needed.

```
public static void disableNetworkPresence() throws java.lang.Exception
```

Here is an example usage of the disableNetworkPresence method:

```
GatewayHelper.disableNetworkPresence();
```

CLIENTAGENTHELPER

The ClientAgentHelper vends client agents. Clients use client agents to access JX Services. All Chordiant services should have corresponding client agents.

The ClientAgentHelper loads the configuration at startup, so it knows which client agents are available for use. Client agents are also loaded on demand, as required. The client agents are cached locally.

The ClientAgentHelper is in the package com.chordiant.service.clientagent.ClientAgentHelper.

Methods

- **getClientAgent**—Returns the specified client agent, given the client agent class's name. This class name is also how the client agent is registered in the XML configuration file.

```
ClientAgent getClientAgent(String ClientAgentName)
```

- **getClientAgentForKey**—Returns a client agent for callback purposes, given the client agent class's name and its network presence key. The class name is also how the client agent is registered in the XML configuration file.

```
ClientAgent getClientAgentForKey(String ClientAgentName, String keyString)
```

Note: The ClientAgentHelper returns an Object, not an actual client agent. You must cast the result of the ClientAgentHelper.getClientAgent method to the client agent you desire, as shown here:

```
myClientAgent = (AccountClientAgent)
ClientAgentHelper.getClientAgent(AccountClientAgent)
```

SECURITY SERVICE

The Security Manager Service (formerly the SecurityHelper) provides an authentication token and checks the authorization of the user when a user name and password are entered.

The Security Manager Service is documented in "Security" on page 253.

CUSTOMOBJECTHELPER

This component enables you to put any CustomObject Java class into the J2EE application server container and have it work with your system.

The CustomObjectHelper is documented in "CustomObjects and the CustomObjectHelper" on page 176.

Chapter 6

Chordiant 5 Foundation Server Administration

The Chordiant Administrative Console enables you to monitor and control Chordiant services.

Information on exceptions and error handling is available at the end of this chapter, starting on page 92.

MONITORING THE CHORDIANT 5 FOUNDATION SERVER SYSTEM

Monitoring the system—including startup, shutdown, refresh, and status methods—is done through the Chordiant 5 Foundation Server Administrative Console.

The Administrative Console enables you to:

- Visualize the layout of your distributed system
- Issue service control commands from the namespace all the way down to the subcomponent level

Figure 6-1 on page 65 shows the basic layout of a generic distributed system, including the namespace, groups, JVMs, components, and subcomponents.

The following components are present in Figure 6-1:

- **Namespace —** Includes all servers and JVMs in a system.
- **Group —** A set of JVMs that you configured.
- **JVM —** A J2EE application server replicate which contains J2EE and Foundation Server components.
- **Components and subcomponents**:
  - **CustomObjectHelper:** A manager for CustomObjects.

    CustomObjects are the subcomponents.

    For more information, refer to "CustomObjects and the CustomObjectHelper" on page 176.

    Do not use the commands in the Administrative Console on the CustomObjectHelper. See the note on page 73.
  - **StaticHelper:** A manager for static objects within the JX architecture.

    Static Objects are the subcomponents.

    For more information, refer to "StaticHelper" on page 45.
  - **EJBs:** A J2EE entity which manages JX services.

    Services are the subcomponents.

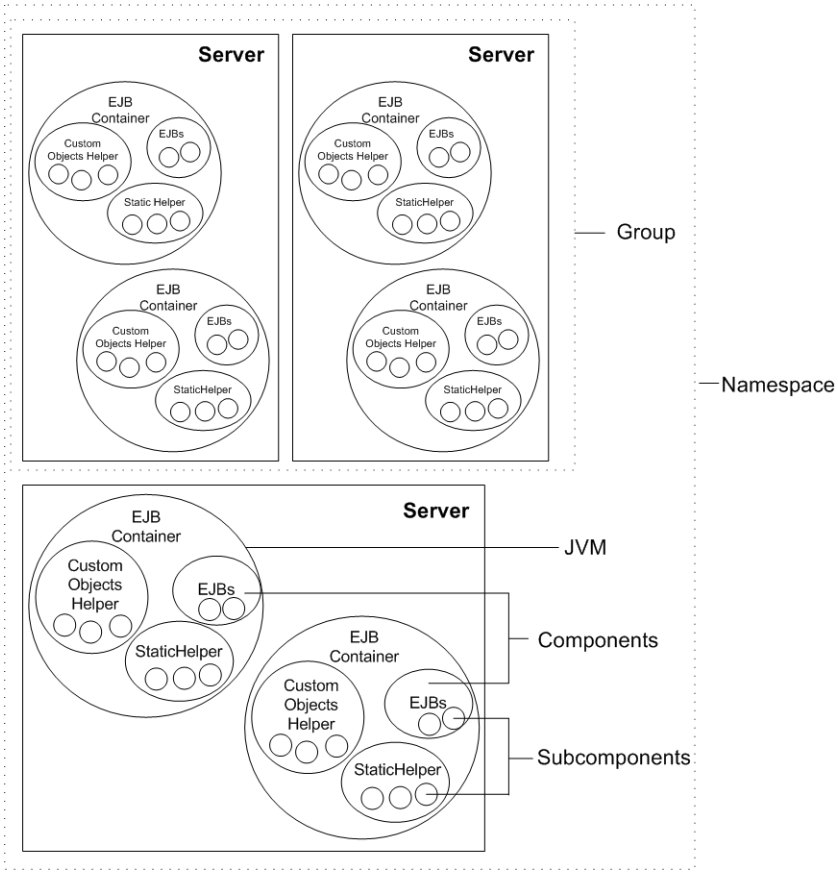


Figure 6-1: Distributed System Layout

Using the Administrative Console

To send a command:

1. Open the Administrative Console from the Chordiant Tools Platform. From the **Admin** menu, select **JX Admin**.

Note: The Administrative Console is a Java Swing application that can be run in UNIX by creating a shell executable file.

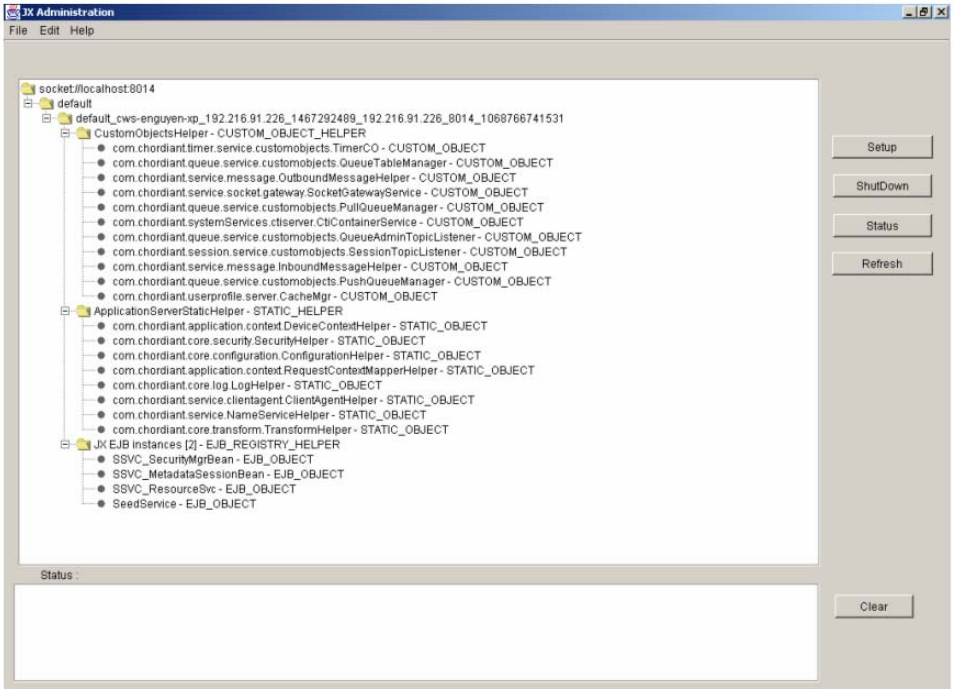


Figure 6-2: Administrative Console

2. Select the target you want to control. The target can be at any level described in Figure 6-1.

Note: When you send a command, it affects that level, and any level below it. So if you send a command to a component, it affects both the component and any subcomponents it contains. If you send a command to a namespace, everything within it is affected.

---

3. Click the appropriate command button.
   - — **Setup**: Sends the setup command
   - — **Shutdown**: Sends the shutdown command
   - — **Status**: Retrieves status for the selected objects
   - — **Refresh**: Refreshes the selected objects
4. View the status information in the Status panel.

   You can copy and paste status information into other applications. Click **Clear** to clear the status panel.
5. To reset the information in the Administrative Console, select **Reset** from the **File** menu.

Service Control API

You can also control these service control functions programmatically through the API of the AdministrationHelper Java class. Refer to the Javadoc for the `AdministrationHelper.java` class.

Service Control through the Command Line

Chordiant also provides a command line interface to the service control functions. The `ServiceControlCommander.java`, located in the com.chordiant.queue.test.ServiceControlCommander package, can be used across JVMs to control services, custom objects, and static helpers.

Code Sample 6-1 shows how the commands are structured.

```
-DadministrationURL=socket://{ADMIN_IP_ADDRESS}:{ADMIN_PORT}
-DserviceControlCommand={setup | refresh | shutdown | status}
-DnamedComponentType={EJB_REGISTRY_HELPER | CUSTOM_OBJECT_HELPER | STATIC_HELPER}
-DnamedSubComponent={SUB_COMPONENT_NAME}
[-DparameterData={PARAMETER_DATA}]
```

Code 6-1: Commands for Service Control

Where:

- administrationURL is the socket and port number where the Administrative Console is running.
- serviceControlCommand is a choice of setup, refresh, shutdown, and status. These commands are discussed on "Standard Behavior" on page 69.
- namedComponentType is whether the target object is an EJB registry helper, a custom object helper, or a static helper.
  - — EJB registry helper: a service within the JX EJB.
  - — custom object helper: specified for custom objects, like the asynchronous messaging helpers.
  - — static helper: the helpers provided with Chordiant, including the ConfigurationHelper and LogHelper, as described in "Static Helpers" on page 70.

- namedSubComponent is:

  For a namedComponentType of:

  — EJB_REGISTRY_HELPER: The CLASS_NAME of the Service.

  — CUSTOM_OBJECT_HELPER: The PACKAGE_NAME plus "." plus the CLASS_NAME of the CustomObject.

  — STATIC_HELPER: The PACKAGE_NAME plus "." plus the CLASS_NAME of the StaticHelper.

- PARAMETER_DATA is:

  If provided, will be sent to the target subcomponent and be available within the ServiceControlRequest.getBuffer() attribute.

The `jxcore.jar` file must be in the Java classpath for this to work.

Code Sample 6-2 provides an example of a command to refresh the ConfigurationHelper sent through the command line utility.

```
java -DadministrationURL=socket://localhost:7014 -DserviceControlCommand=status
-DnamedComponentType=STATIC_HELPER
-DnamedSubComponent=com.chordiant.core.configuration.ConfigurationHelper
com.chordiant.queue.test.ServiceControlCommander
```

Code 6-2: Using the Command Line Utility

Make sure that `jxcore.jar` is in the Java classpath.

On the server side, add these arguments to the command line (iii=IP address. ppp = port):

-Dchordiant.service.socketGatewayServiceIPAddress=iii

-Dchordiant.service.socketGatewayServicePort=ppp

Refer to "Using the Foundation Server SocketGatewayService" on page 151 and the annotations within the utility file for additional information.

For other administrative tasks, refer to the *Chordiant 5 Tools Platform Administration Manager Guide*.

Security and the Administrative Console

The Administrative Console uses the SocketGatewayService, which provides secure access to clients within your internal trusted network. For additional information, refer to "Security and the SocketGatewayService" on page 153.

Behavior of Services within the Administrative Console

Standard Behavior

As described above, here is a list of the standard behavior of services within the Administrative Console:

- **Setup** — Sends the setup command.
- **Shutdown** — Sends the shutdown command.
- **Status** — Retrieves status for the selected objects.
- **Refresh** — Refreshes the selected objects.

Note: When you send a command, it affects that level, and any level below it. So if you send a command to a component, it affects both the component and any subcomponents it contains. If you send a command to a namespace, everything within it is affected.

Clicking one of the four buttons on the Administrative Console—**Setup**, **Shutdown**, **Refresh**, **Status**—effectively sends the corresponding service control command to the target service, custom object, or static object. Within the target, this command can perform standard functionality, as described above, or other behavior as written by the creator of the service, custom object, or static object.

Actual Behavior of Chordiant-Provided Services

This section provides a list of actual behavior from Chordiant-provided services within the Administrative Console.

Notes: Services must be instantiated to appear within the Administrative Console.

The Administrative Console does not filter the components it displays. Not all components that appear in the Administrative Console actually require use of the Administrative Console. Components which you might use more frequently are noted with an asterisk (*).

Any component which uses configuration is dependent on the ConfigurationHelper. Refresh the ConfigurationHelper before refreshing the other component.

Core Components

Static Helpers

- **ClientAgentHelper** — com.chordiant.service.clientagent.ClientAgentHelper
  Refer to "ClientAgentHelper" on page 61 for details on this component.
  - — **Setup**: Reads the configuration values.
  - — **Shutdown**: Calls shutdown on all client agents within the cache.
  - — **Refresh**: Clears the client agent cache.
  - — **Status**: Returns "OK".
  - — **Dependencies**: Must refresh ConfigurationHelper before refreshing the ClientAgentHelper.

    Uses: You might want to refresh the ClientAgentHelper if you have added or modified any services and don't want to have to bring down the whole server. You must refresh the ConfigurationHelper first to see the new client agents and services.

- **ConfigurationHelper*** — com.chordiant.core.configuration.ConfigurationHelper
  Refer to "ConfigurationHelper" on page 46 for details on this component.
  - — **Setup**

    Client side: Sets up communication to the server configuration.

    Server side: Reads the configuration.
  - — **Shutdown**

    Client side: Closes communication to the server.

    Server side: Clears out the local cache of configuration values.
  - — **Refresh**

    Client Side: Reconnects to server.

    Server Side: Shuts down the server, then calls setup methods to clean out the local cache. Then rereads the local configuration files.
  - — **Status**

    Client Side: Returns nothing.

    Server Side: Returns nothing. The client-side status command is not passed to the server side.

    Uses: Any time you change any configuration file, you can refresh the ConfigurationHelper to reload the configuration settings without bringing down the server. You must usually refresh the ConfigurationHelper before refreshing other helpers.
- **DeviceContextHelper*** — com.chordiant.application.context.DeviceContextHelper
  See also RequestContextMapperHelper on page 72.
  Refer to "Understanding the Device Context Mapper Helper" on page 304 for details on this component.
  - — **Setup**: No action.
  - — **Shutdown**: Not implemented.
  - — **Refresh**: Clears the device context mapping.
  - — **Status**: Not implemented.
  - — **Dependencies:** Must refresh the ConfigurationHelper first before you can refresh this service.

    Uses: You can change the mapping for a certain device, for example, how some content will appear for web-based clients, you can refresh the DeviceContextHelper to implement these changes, rather than having to restart the server. You should refresh the ConfigurationHelper before refreshing the DeviceContextHelper.

- **LogHelper*** — com.chordiant.core.log.LogHelper
  Refer to "LogHelper" on page 47 for details on this component.
  - — **Setup**: Reads configuration and reconfigures filters and writers.
  - — **Shutdown**: Clears out all log handlers.
  - — **Refresh**: Calls the shutdown, configurationLogSetup, then setup methods.
  - — **Status**: Returns "OK".
  - — **Dependencies**: You must refresh the ConfigurationHelper before refreshing the LogHelper.

    Uses: You can change the level of logging in the `LogHelper.xml` file if, for example, you want to debug a certain section of code. You can then refresh the ConfigurationHelper and the LogHelper in the Administrative Console without having to restart the server. When you have finished debugging, you can change the configuration file and refresh again.
- **NameServiceHelper** — com.chordiant.service.NameServiceHelper
  Refer to "Exploring the Primary Classes" on page 305 for details on this component.
  - — **Setup**: Initializes a new NameServiceJNDIHelper which makes a connection to JNDI.
  - — **Shutdown**: No action.
  - — **Refresh**: No action.
  - — **Status**: Returns "OK".
- **RequestContextMapperHelper*** — com.chordiant.application.context.RequestContextMapperHelper
  See also DeviceContextMapper on page 71.
  Refer to "Understanding Request Context Mapping" on page 290 for details on this component.
  - — **Setup**: No action.
  - — **Refresh**: Clears the request context mapping.
  - — **Dependencies:** Must refresh the ConfigurationHelper first before you can refresh this service.

    Uses: In the RequestContextMap, you can change the mapping of an ActionID to a specific functionality. You can refresh the RequestContextMap to implement these changes, rather than having to restart the server. You should refresh the ConfigurationHelper before refreshing the RequestContextMapperHelper.
- **SecurityHelper** — com.chordiant.core.security.SecurityHelper
  Refer to "Security Service" on page 61 for details on this component.
  - — **Setup**: Calls configuration then sets up security resources.
  - — **Shutdown**: Not supported for security reasons. Nothing happens when this command is called.
  - — **Refresh**: Not supported for security reasons. Nothing happens when this command is called. You can refresh (keep up-to-date) cached security data by using the CacheMgr. See page 73 for information.

  - — **ServiceControl: Status**: Returns “OK”.
  - — **Dependencies**: SecurityHelper depends on the Cache Manager to be set up successfully.

- **TransformHelper** — com.chordiant.core.transform.TransformHelper
  Refer to “Transformation Helper” on page 284 for details on this component.
  - — **Setup**: Reads configuration values.
  - — **Refresh**: Makes new instance of the cache.
  - — **Shutdown**: Clears out template cache.
  - — **Status**: Returns “OK”.
  - — **Dependencies:** Must refresh the ConfigurationHelper first before you can refresh this service.

    Uses: If you change the way that data is transformed from the payload, refresh the TransformHelper to implement your change without shutting down the server.

Custom Objects

- **CacheMgr** — com.chordiant.userprofile.server
  - — **Setup**: No action.
  - — **Shutdown**: No action.
  - — **Status**: Shows “OK” if everything is running normally.
  - — **Refresh**: Forces the cache manager instance you selected to discard any cached data and re-read data from database. Therefore, after this action, all cache data within this instance become synchronized with data in the database. To refresh cache across the cluster, you must refresh the cluster.

    Uses: Use the CacheMgr to refresh any cached user profile data and access control information. Currently, CacheMgr only manages cached security data, including certain user profile data.

- **CustomObjectHelper**
  Refer to "CustomObjectHelper" on page 62 for details on this component.

Notes: Do not use the Administrative Console with the CustomObjectHelper. All commands you send through it ripple through all of the custom objects in the system, which include the socket gateway. If you shut down or refresh (which involves shutting down) the CustomObjectHelper, you will shut down the socket gateway. The Administrative Console works through the socket gateway, so it will be shut down too, cutting off that communication. Nobody will be able to send service controls after the socket gateway is shut down. The socket gateway will restart itself, but you must restart the Administrative Console manually.

For information on security of the Administrative Console, refer to "Security and the SocketGatewayService" on page 153.

- **SocketGatewayService** — com.chordiant.service.socket.gateway.SocketGatewayService
  Refer to "Using the Foundation Server SocketGatewayService" on page 151 for details on this component.
  - — **Setup**: Reads configuration and sets up communication socket.
  - — **Shutdown**: Shuts down communication socket and deletes administration file. Automatically calls the setup function, so the socket is not down for long.
  - — **Status**: Returns detailed operational data.
  - — **Refresh**: Not supported.

Note: The Administrative Console runs through the SocketGatewayService. If you shut it down, it will cut the connection from the Administrative Console to the server. The SocketGatewayService will restart automatically, but you will have to start the Administrative Console manually.

Custom Objects for Asynchronous Messaging

- **OutboundMessageHelper** — com.chordiant.service.customobjects.OutboundMessageHelper
  Refer to "Outbound Messages" on page 242 for details on this component.
  - — **Setup**: Reads the configuration and initializes the instance.
  - — **Shutdown**: Shuts down owned objects.
  - — **Refresh**: Calls shutdown method, followed by the setup method.
  - — **Status**: Sends a detailed operational message.

  JMS sessions can be created when this custom object is setup and destroyed when it is shutdown. This custom object can be enabled or disabled in the `outboundMessageHelper.xml` configuration file.

Custom Objects for Queueing Services

The following four services are in the com.chordiant.queue.service.customobjects package. All four services support the standard service control commands.

- **PullQueueManager**
  - — Processes service control events and responds with status messages. JMS sessions are created when this custom object is set up and destroyed when it is shut down. The JMS MessageListener interface is implemented by the PullQueueListener objects that this custom object manages in an array, so there are threads actively listening for and processing queue items (awaiting QueueDetermination having been injected by the Queue API) when the PullQueueManager has been set up (and not when it is shut down).
  - — Exercise extreme caution when changing the state of this custom object.

    Uses:

    If you lose your connection to JMS, you might want to send the Refresh command to this custom object or use the Shutdown command followed by the Setup command.

    In a development environment, if your database space fills up, an exception is thrown. This causes the transaction to rollback. The queue will shut itself down and report an error. You can use the Setup or Refresh commands to start the queue again. Note that a well-managed production environment should not allow such database problems.

    Sending the Status command will list of the number of queue items processed asynchronously for each of the System Pull Queue replicates, along with other queue information.

- **PushQueueManager**
  - — Processes service control events and responds with status messages. JMS sessions are created when this custom object is set up and destroyed when it is shut down. The JMS MessageListener interface is implemented by the PushQueueListener objects that this custom object manages in an array so there are threads actively listening for and processing queue items (awaiting QueueDetermination having been injected by the Route API) when the PushQueueManager has been set up (and not when it is shut down).

    Exercise extreme caution when changing the state of this custom object.

    Uses: Same uses as "PullQueueManager".
- **QueueAdminTopicListener**
  - — Processes service control events and responds with status messages. JMS sessions are created when this custom object is set up and destroyed when it is shut down. The JMS MessageListener interface is implemented so this custom object actively listens for and processes administration messages including the Queue count state information when it has been set up (and not when it is shut down).

    Exercise extreme caution when changing the state of this custom object.

    Uses: Same uses as "PullQueueManager".
- **QueueTableManager**
  - — Processes service control events and responds with status messages. Many JMS sessions are created when this custom object is set up and are destroyed when it is shut down. The QueueTableManager might count the JMS queues when it receives a setup or refresh service control event or it might try and recover this state from another Application Server replicate when running in a J2EE clustered environment.

Note: The queue items that reside in the JMS Queues are not accessible when this custom object is shut down, starting up, or refreshing so Chordiant recommends doing this as infrequently as possible in a live production environment.

Uses: If you change the queue definitions, you can refresh this custom object to bring those changes into effect. However, care should be used in a production environment because it can affect system performance and availability.

- **SessionTopicListener** — com.chordiant.session.service.customobjects.SessionTopicListener
  - — Supports all four standard service control commands.
  - — Processes service control events and responds with status messages. JMS sessions are created when this custom object is set up and destroyed when it is shut down. The JMS MessageListener interface is implemented so this custom object actively listens for and processes session messages including the session availability state information when it has been set up (and not when it is shut down).

    Exercise extreme caution when changing the state of this custom object.
- **TimerCO** — com.chordiant.timer.service.customobjects.TimerCO
  - — Processes service control events minimally with default messages.
  - — **Setup**: The timer engine is created when this custom object is set up, which depending upon the configuration, might create sessions with the database through JDBC. Timers can be created and/or fired when the TimerCO has been set up (and not when it is shut down).
  - — **Shutdown**: Not functional.
  - — **Refresh**: Not functional.
  - — **Status**: Not functional.

    Exercise extreme caution when changing the state of this custom object.

    Uses: This custom object is started, like all other custom objects, when the application server is started. The other service controls are not available on it. If you lose a connection to the database, you must check it manually through standard J2EE practices.

Services

- **HelloWorldService** — com.hello.world.service.HelloWorldService
  Refer to the *Chordiant 5 Tools Platform Installation and Configuration Guide* for details on this component.
  - **Setup**: Sends back an acknowledgement with the parameter data received.
  - **Shutdown**: Sends back an acknowledgement with the parameter data received.
  - **Status**: Sends back an acknowledgement with the parameter data received.
  - **Refresh**: Sends back an acknowledgement with the parameter data received.
- **PersistentCacheManager** — com.chordiant.core.persistentcache.PersistentCacheManager
  See also XMLStorageService on page 84.
  - **Setup**: Gets configuration values and then resets state.
  - **Shutdown**: Shuts down local connections and values.
  - **Refresh**: Gets configuration values and then resets state.
  - **Status**: Not implemented.

    Uses: If you change the data source, or any of its parameters (like the password), where Java objects are persisted, you can make the change in the configuration file, then refresh the ConfigurationHelper and the PersistentCacheManager to implement the change without restarting the server.

Chordiant Way Service

- **CwapiService** — com.chordiant.cwapi.service
  Refer to the *Chordiant 5 Foundation Server Business Process Server Developer's Guide* for details on this component.
  - **ServiceControl Messages**: Standard behavior for all.

    Uses: You should not need to use service control on this service.

Chordiant Business Services

Refer to the *Chordiant 5 Foundation Server Application Components Developer's Guide* for details on these Chordiant-provided business services.

Note: Any service that sets up a Resource Manager has an indirect dependency on ConfigurationHelper because the Resource Manager calls ConfigurationHelper to read configuration data. You must refresh the ConfigurationHelper before you refresh the target service, if you have updated that service's configuration file.

- **Uses**:

  Most Chordiant Business Services respond similarly to service control commands. This section describes why you might want to issue commands to Chordiant business services:

  — **Setup:** This command is automatically sent on initialization of the business service. It gets the service ready for use, including setting up its corresponding Resource Manager.

  — **Shutdown:** Give this command to stop a specific service from running. You might want to do this if, for example, you lose a connection and need to shut down a service and set it up again. It is very rare that you would want to shut down a service and most services do not respond to the shutdown command.

  — **Status:** Give this command to find out if a service is still up and running. Services that implement this command usually return "OK".

  — **Refresh:** Give this command if you have added something to the database directly and want to reload the cache.

- **AccountService** — com.chordiant.bd.services.AccountService
  EJBCMTRequired

  — **Setup:** Initializes Resource Manager, gets an instance of account number generator, gets a lock service, client agent, and a PartyRole client agent.

  — **Refresh**: Not implemented

  — **Shutdown**: No action.

  — **Status**: No action.

  — **Refresh**: No action.

  — **Dependencies:** Must refresh the ConfigurationHelper first before you can refresh this service, if you have updated that service's configuration file.

- **BusinessObjectFactoryService** — com.chordiant.bd.services.BusinessObjectFactoryService
  EJBCMTRequired

  — **Setup:** Initializes Resource Manager, parses and loads CMI file.

  — **Shutdown**: No action.

  — **Refresh**: Not implemented.

  - — **Status**: No action.
  - — **Refresh**: No action.
  - — **Dependencies:** Must refresh the ConfigurationHelper first before you can refresh this service, if you have updated that service's configuration file.
- **DeliveryService** — com.chordiant.bd.services.DeliveryService
EJBCMTRequired
  - — **Setup**: Not implemented.
  - — **Refresh**: Not implemented.
  - — **Shutdown**: Not implemented.
  - — **Status**: Not implemented.
  - — **Refresh**: Not implemented.
  - — **Dependencies:** com.chordiant.bd.services.OrderGenerationService references one its fields. (See also page 82.)
- **EbcInteractionService** — com.chordiant.bd.services.EbcInteractionService
EJBCMTRequired
  - — **Setup**: Sets up Resource Manager, configures PMFCustomer and other client agents.
  - — **Refresh**: Not implemented.
  - — **Shutdown**: Not implemented.
  - — **Status**: Not implemented.
  - — **Refresh**: Not implemented.
- **GenericService** — com.chordiant.bc.services.GenericService
EJBCMTRequired
  - — **Setup**: Initializes Resource Manager.
  - — **Shutdown**: No action.
  - — **Refresh**: Not implemented.
  - — **Status**: No action.
  - — **Refresh**: Refreshes Behavior Factory and Validator Factory. Clears cache.
  - — **Dependencies:** Must refresh the ConfigurationHelper first before you can refresh this service, if you have updated that service's configuration file.
- **GuideService** — com.chordiant.bd.services.GuideService
EJBCMTRequired
  - — **Setup**: Sets up ResourceManager.
  - — **Refresh**: Standard behavior.
  - — **Shutdown**: Standard behavior.
  - — **Status**: Standard behavior.
  - — **Refresh**: Clears cache.

- **InventoryService** — com.chordiant.bd.services.InventoryService
  EJBCMTRequired
  - **Setup:** Sets up ResourceManager.
  - **Refresh**: Not implemented.
  - **Shutdown**: Not implemented.
  - **Status**: Not implemented.
  - **Refresh**: Not implemented.
  - **Dependencies:** `com.chordiant.bd.services.OrderGenerationService`
- **LocationService** — com.chordiant.bd.services.LocationService
  EJBCMTRequired
  - **Setup:** Sets up ResourceManager.
  - **Refresh**: Standard behavior.
  - **Shutdown**: Standard behavior.
  - **Status**: Standard behavior.
  - **Refresh**: Clears cache.
- **LockService** — com.chordiant.lock.service.LockService
  EJBBMT
  - **Setup:** Loads configuration data and gets a database connection pool.
  - **Shutdown**: Releases database connection pool.
  - **Refresh**: Not implemented.
  - **Status**: Lists current locked objects.
  - **Refresh**: Releases database connection pool, reloads configuration data, and gets a database connection pool.
  - **Dependencies:** Must refresh the ConfigurationHelper first before you can refresh this service, if you have updated that service's configuration file.
- **NumberGenerationService** — com.chordiant.bd.numberGeneration.NumberGenerationService
  EJBBMT
  - **Setup:** Sets up ResourceManager.
  - **Shutdown**: Not implemented.
  - **Refresh**: Not implemented.
  - **Status**: No action.
  - **Refresh**: Clears cache.
  - **Dependencies:** Its own client agent has a reference to one of its fields.

- **OfferingService** — com.chordiant.bd.services.OfferingService
  EJBCMTRequired
  - **Setup:** Sets up Resource Manager, reads transaction DSN, and creates connection pool.
  - **Shutdown**: Clears connection pool.
  - **Refresh**: Not implemented.
  - **Status**: Not implemented.
  - **Refresh**: Clears cache.
- **OrderFullfillmentService** — `com.chordiant.bd.services.OrderFullfillmentService`
  EJBCMTRequired
  - **Setup:** Initializes Resource Manager, caches instance of PmfCustomer client agent.
  - **Shutdown**: Not implemented.
  - **Refresh**: Not implemented.
  - **Status**: Not implemented.
  - **Refresh**: Clears cache.
  - **Dependencies:** Must refresh the ConfigurationHelper first before you can refresh this service, if you have updated that service's configuration file.
- **OrderGenerationService** — `com.chordiant.bd.services.OrderGenerationService`
  EJBCMTRequired
  - **Setup**: Initializes Resource Manager, caches instance of delivery/product/inventory client agent, obtains number generator for order item and return merchandise.
  - **Shutdown**: Nothing.
  - **Refresh**: Not implemented.
  - **Status**: Nothing.
  - **Refresh**: Clears cache.
  - **Dependencies**: Must refresh the ConfigurationHelper first before you can refresh this service, if you have updated that service's configuration file. See also Delivery Service on page 80.

- **OrderTrackingService** — `com.chordiant.bd.services.OrderTrackingService` EJBCMTRequired
  - **Setup**: Initializes Resource Manager.
  - **Shutdown**: Not implemented.
  - **Refresh**: Not implemented.
  - **Status**: Not implemented.
  - **Refresh**: Not implemented.
  - **Dependencies:** Must refresh the ConfigurationHelper first before you can refresh this service, if you have updated that service's configuration file.
- **PartyRoleService** — `com.chordiant.pmf.service.PartyRoleService` EJBCMTRequired
  - **Setup**: Sets up Resource Manager, makes data cache, stores it with Resource Manager.
  - **Shutdown**: No action.
  - **Status**: No action.
  - **Refresh**: No action.
- **PmfCustomerService** — com.chordiant.customer.service.PmfCustomerService EJBCMTRequired
  - **Setup**: Sets up Resource Manager.
  - **Shutdown**: No action.
  - **Status**: No action.
  - **Refresh**: No action.
- **ProductService** — com.chordiant.bd.services.ProductService EJBCMTRequired
  - **Setup**: Sets up Resource Manager, user name, password, and PartyRoleClientAgent
  - **Shutdown**: No action.
  - **Status**: No action.
  - **Refresh**: Clears cache.
  - **Dependencies**: Referenced by OrderGenerationService for its constants fields.(See also page 82.)

- **XMLStorageService** — com.chordiant.xmlstorage.service.XMLStorageService
  EJBCMTRequired
  See also PersistentCacheManager on page 78.
  - **Setup**: Sets up Resource Manager.
  - **Shutdown**: No action.
  - **Refresh**: Standard behavior.
  - **Status**: Not functional.

    Uses: If you change the data source where XML objects are persisted, or any of its parameters (such as the password), you can make the change in the configuration file, then refresh the ConfigurationHelper and the XMLStorageService to implement the change without restarting the server.

CTI Services

- **CTI Container Service** — com.chordiant.systemServices.ctiserver.CtiContainerService
  - **Setup**: Creates the number of CTI containers specified in the `TelephonyService.xml` configuration file. Registers the CTI containers with JNDI and stores the containers in a Vector. Initializes the next available container attribute.
  - **Shutdown**: Deregisters the CTI containers from JNDI. Calls shutdown on the CTI containers and destroys the object.
  - **Refresh**: Performs a shutdown, followed by a setup.

    Making a change to the number of CTI containers in the `TelephonyService.xml`, requires a refresh of the ConfigurationHelper first.
  - **Status**: Returns the String "running *{time stamp}*" if a connection to the CTI Container Service is established.
  - **Dependencies**: Refreshes the ConfigurationHelper before refreshing the CTI Container Service.

    Uses: You might want to refresh this service if you have added extensions to your system and this requires an increase in the number of CTI containers. The number of CTI containers is set in the `TelephonyService.xml` file. Refresh the ConfigurationHelper first, then refresh this service.

- **VRU Socket Service** — com.chordiant.businessServices.ctiServices.VruSocketService
  - **Setup**: Starts up the VRU Socket Service. When this is called, it creates a socket server that waits for client socket connections from the VRU.
  - **Shutdown**: Shuts down the socket server created in the setup command.
  - **Refresh**: Shuts down the socket server, re-initializes all variables, and starts up the socket server.

    To change the port number for the socket service, it must be changed on the command line and the server must be brought down and back up again.
  - **Status**: Returns the String "running *{time stamp}*" if a connection to the VRU Socket Service is established.

    Uses: If you have changed any offerings, you might want to refresh this service to refresh the offerings cache. You might also want to refresh this service if you require a new VRURequestHandler.

Queuing Services

- **QueueService** — com.chordiant.queue.service
  `EJBCMTRequired`
  - **ServiceControl Messages**: Standard behavior for all
- **SessionService** — `com.chordiant.session.service`
  `EJBBMT`
  - **ServiceControl Messages**: Standard behavior for all
- **TimerService** — com.chordiant.timer.service
  `EJBCMTRequired`
  - **ServiceControl Messages**: Standard behavior for all

    Uses: These services depend on the custom objects listed in "Custom Objects for Queueing Services" on page 75. Refer to the individual custom objects in that stationery more information.

Multiple Application Server JVMs and SocketGatewayService

If you have multiple application server JVMs, the administrative console must be aware of all the JVMs that exist in the namespace—that is, all of the JVMs within the application server cluster. For this to happen, all the JVMs should have access to a common mounted directory, so a unique file with associated connection information for each JVM can be created in this directory. Therefore, the number of files is this directory is the same as the number of JVMs in that cluster. This directory can be specified using the following server-side command line property:

-Dchordiant.administration.directory=xxx

Each JVM in the cluster listens on a different IP address and port number. The Administrative Console needs to connect to only one of the JVMs, so any IP address and port number will be sufficient. From that information, the other JVMs can be retrieved. Refer to the next section, "Configuring a Cluster Environment for Use with the Administrative Console" on page 88, for details on setting this up.

Each time you do a cold start on the cluster, a time-stamped instance file for a JVM will be created in that directory. If you bring down the system and restart it, other unique instance files will be created. The Administrative Console will only process the most recently time-stamped instance file for a JVM, so if you restart with fewer JVMs in your cluster the Administrative Console will think there are more replicates than there really are, unless the old ones are cleaned up first. Regardless, the number of these files will grow over time and you should periodically clean up the instance files.

As shown in Figure 6-1 on page 65, multiple arbitrary JVMs can exist in a group. The name of the group for a specific JVM can be specified using the following server-side command line property:

```
-Dchordiant.administration.group=xxx
```

If nothing is specified, the name of the group is default. For example, if you chose abc as your group name, -Dchordiant.administration.group=abc, the hierarchy would look like this:

```
NameSpace
    -abc
        -JVM1
             -Custom Object
              -Static helpers
              -Services
        JVM2
            -Custom Object
             -Static helpers
             -Services
```

You could also have multiple groups.

```
NameSpace
       -abc
       -JVM1
         -Custom Object
         -Static helpers
         -Services
       JVM2
         -Custom Object
         -Static helpers
         -Services

   NameSpace
   -xyz
      -JVM1
        -Custom Object
        -Static helpers
        -Services
      JVM2
        -Custom Object
        -Static helpers
        -Services
```

Figure 6-3 on page 88 is for the namespace socket://engibm03:7014.

The group is default and there are two JVMs. They are:

- default_engibm03_204.162.92.117_1516095122_204.162.92.117_7015
- default_engibm03_204.162.92.117_1516095370_204.162.92.117_7014

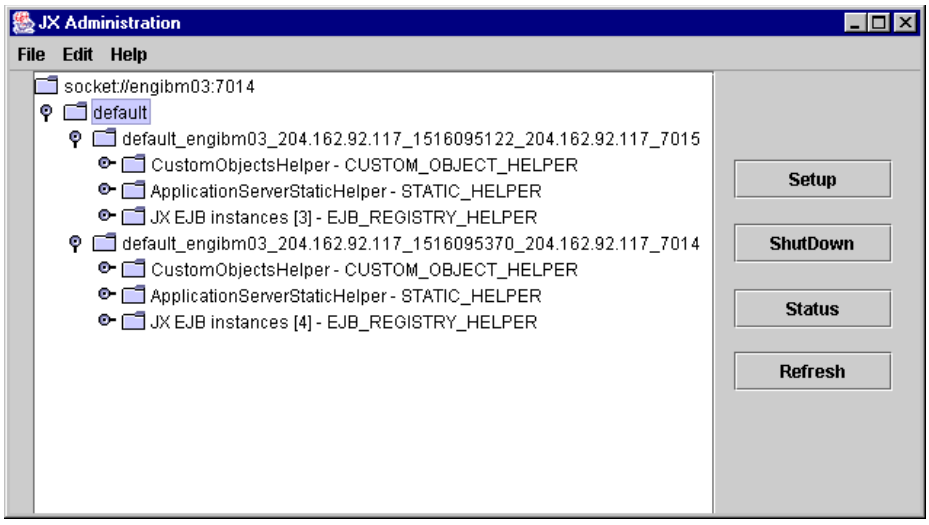


Figure 6-3: Administrative Console Showing a Specific Namespace Socket

Configuring a Cluster Environment for Use with the Administrative Console

Chordiant's Administrative Console and other applications look in the specified location that is given in the custom parameters of the JVM. During this setup, you will set up each JVM to enable unique administration through Chordiant's Administrative Console. Reading the previous section, "Multiple Application Server JVMs and SocketGatewayService" on page 85, will give you background on this setup.

To configure a cluster environment that can be under administrative control:

1. On one computer in the cluster, share the existing `JXAdmin` directory, for example, *`{CHRD_EAR_DEPLOYED}`*`/jxadmin`.
2. Mount this directory on all the nodes in the cluster in the same location: *`{CHRD_EAR_DEPLOYED}`*`/jxadmin`.

3. After making the clones for the cluster, bring up the custom parameters for each application server clone. To open the custom parameters, select: **Application Servers | *{clone number}* | Process Definition | Java Virtual Machine | Custom Parameters.**

   Coordinate these custom parameters:

   — **Administrative Console port [administrationURL]**: This URL is used by the Chordiant Administrative Console and must be unique for each application server.

   — **VRU port [Chordiant.ctiServices.vruSocketServerPort]:** This is used by the CTI Service. By default, this number is one greater than the value for the Administrative Console port, described above, but can vary depending on your system setup.

   — **Administrative Service Port: [Chordiant.service.socketGatewayServicePort]:** This should match the Administrative Console URL's port number described above. It is used by the Chordiant Administrative Console to send requests, like Refresh Queues, across the cluster. This port is also used by the CTI Service.

   — **Administrative Console Directory [chordiant.administration.directory]**: The directory where the Administrative Console resides on the computer. By default, this value is *{CHRD_EAR_DEPLOYED}*/jxadmin.

   Set these configuration parameters:

   — **instance number [com.chordiant.instance]**: Each application server clone must have a unique instance number. This number should be incremented by one for each application server clone, in the order of starting sequence.

   The number of clones in the cluster must be reflected in the master.dtd file for the application EAR. In the master.dtd file for the application EAR, update the default entry value

   ```
   <!ENTITY MAXIMUM_INSTANCES "1">
   ```

   to include the actual number of clones in the cluster.

   The master.dtd file is located in the *{APP_SERVER_INSTALL}*/AppServer/installedApps/*{Cell Name}*/ ChordiantEAR/config/Chordiant/ directory.

   For example, for Cell_01 on WebSphere, the master.dtd would be located in /usr/Was51/WebSphere/AppServer/installedApps/Cell_01/ ChordiantEAR/config/Chordiant

   — **Number of CTI Managers [NumberOfCtiManagers]:** This is related to the number of application server clones and is specified in the TelephonyService.xml configuration file.

This example shows use of the ports and instances involved:

The default name of the Chordiant Application Server is ChordiantAppServer, but for clarity we will call it Node01_Clone01.

On Node01, there are two clones:

NODE01 CLONES	
Node01_Clone01	
administrationURL	http://localhost:8010
Chordiant.ctiServices.vruSocketServerPort	8011
Chordiant.service.socketGatewayServicePort	8010
com.chordiant.instance	1
chordiant.administration.directory	*{CHRD_EAR_DEPLOYED}/* jxadmin
Node01_Clone02	
administrationURL	http://localhost:8012
Chordiant.ctiServices.vruSocketServerPort	8013
Chordiant.service.socketGatewayServicePort	8012
com.chordiant.instance	2
chordiant.administration.directory	*{CHRD_EAR_DEPLOYED}/* jxadmin

Code 6-3: Node01 Clone

On Node02, there is one clone:

NODE02 CLONE	
Node02_Clone01	
administrationURL	http://localhost:8014
Chordiant.ctiServices.vruSocketServerPort	8015
Chordiant.service.socketGatewayServicePort	8014
com.chordiant.instance	3
chordiant.administration.directory	*{CHRD_EAR_DEPLOYED}/* jxadmin

Code 6-4: Node02 Clone

In addition, we took these actions:

1. For each node, update the maximum number of instances to equal the total number of clones in the cluster. In our example, there are two nodes with a maximum of three clones. On Node01 and Node02 in the *{APP_SERVER_INSTALL}*/AppServer/installedApps/*{Cell Name}*/ChordiantEAR/config/Chordiant/master.dtd file. We updated the maximum instances to the total count in our example below using this value:

   ```
   <!ENTITY MAXIMUM_INSTANCES "3">
   ```

2. The number of containers in the CTI container system is related to performance: with too few containers, the performance degrades as the system looks for a free container. The number of containers created in the system should be at least the maximum number of agents using CTI at any one time, plus 10%. Creating 20% to 25% more containers than needed ensures a fast response when attempting to find a free container.

   In our example, let's say we will have 1500 users and we already have three clones set up. Each clone will have its own set of CTI containers, so we'll divide 1500 users by 3 clones. Each clone requires 500 containers. So we will add 10% more, for a total of 550 CTI containers.

   For each clone, in the TelephonyService.xml file, located in *{APP_SERVER_INSTALL}*/AppServer/installedApps/*{Cell Name}*/ChordiantEAR/config/Chordiant/components/master, we set this value:

   ```
   <Tag>NumberOfCtiManagers
        <Value>550</Value>
   </Tag>
   ```

   For more information on CTI settings, refer to the *Chordiant 5 Foundation Server Telephony Integration Guide*.

For more information on the SocketGatewayService, refer to "Using the Foundation Server SocketGatewayService" on page 151.

EXCEPTIONS AND ERROR HANDLING

In general, the JX infrastructure communicates errors to the user via exceptions. Exceptions can travel from service to client, and vice versa. If there is an error in a remote service, it is made apparent to the client.

ChordiantBaseException

Chordiant has a defined base exception for use with the JX EJB.

```
package com.chordiant.service.ejb;

import javax.ejb.EJBOblect;
import java.rmi.RemoteException;

import com.chordiant.service.ChordiantBaseException

public interface EJBGatewayService extends EJBObject
{
    String processRequestXMLString(String request)
        throws RemoteException, Exception, ChordiantBaseException;
    Object processRequestObject(Object inputData)
        throws RemoteException, Exception, ChordiantBaseException;
}
```

Within your services, we recommend that you subclass from the ChordiantBaseException for all of your errors to be marshalled through the application server between the client and server. If you do not extend the ChordiantBaseException, your errors might be treated differently, depending on the application server you are using. Only by extending the ChordiantBaseException can you depend on your errors getting through the application server intact.

This is true for all bean managed transactions (BMTs). Container managed transactions (CMTs) should also use the ChordiantBaseException, but must follow additional exception handling. Refer to "Configuring for Rollbacks" on page 122 for more information.

Business Service Client Agent Error Handling

The processRequest method of the ClientAgentBaseClass returns a java.lang.Throwable object if an exception is raised. If the Throwable is one of the exceptions defined in the business service client agent's API, that client agent will recast the Throwable as the appropriate exception. If that is not possible, and the Throwable is not a simple Error or RuntimeException, the client agent returns a com.chordiant.service.ChordiantRuntimeException object that wraps the Throwable object.

EJB Exceptions

If you want to throw an EJB exception from a service, for example during a container managed transaction (CMT), subclass from the com.chordiant.service.ChordiantBaseEJBException class.

Socket Protocol Exceptions

If you are contacting the JX EJB through sockets, exceptions are handled differently than those encountered when working through the native Application Server protocol, as described above.

The Foundation Server socket protocol does not support the transport of the entire native server-side exception. It repackages the server-side exception in a ServiceException and preserves the original message text.

This only affects development environments. Production environments would not use sockets, but use native Application Server protocol.

For more information on using sockets, refer to "Using the Foundation Server SocketGatewayService" on page 151.

Chapter 7

Configuration Files

Configuration files control all Chordiant 5 Foundation Server applications. They are simple files, written in XML, not in a proprietary format. There is a single `master.xml` file and several additional files under the *{CHORDIANT_ROOT}*`/config/Chordiant/components/master` directory which are provided with Chordiant 5 Foundation Server. You can use additional `components/`*{component}*`.xml` files to overlay or add to the master XML configuration files. There are also an unlimited number of *{nodename}*`.xml` files which you can add for each node to further customize your applications. `Sitemaster.xml` overrides the master configuration files (including `master.xml` and `components/master` files) and any `components/`*{component}*`.xml` files.

Notes: Do not to alter the `master.xml` or `components/master` files. Make any modifications in `components/`*{component}*`.xml`, *{nodename}*`.xml`, or `sitemaster.xml` files.

{CHORDIANT_ROOT} corresponds to the chordiant.configuration.configurationRootDirectory parameter in your application server. Refer to "Configuration Files and the ConfigurationRootDirectory" on page 46 for more information.

For details on editing the configuration files, refer to "Adding Components through Configuration" on page 101.

Configuration files are located in *{CHORDIANT_ROOT}*`/config/Chordiant` and *{CHORDIANT_ROOT}*`/config/Chordiant/components` directories.

A few of the many uses of the configuration files are:

- *Services* look in the configuration files for specific sections where they can find information.
- *Client agents* look in the configuration files for a list of all the available services.
- The *ClientAgentHelper* looks in the configuration files for a list of all available client agents.

CHORDIANT XML CONFIGURATION FILE STYLE

All Chordiant XML configuration files (including `master.xml`, `components/master` files, `components/{component}.xml`, `{nodename}.xml`, and `sitemaster.xml`) have the following styles in common:

- All are well-formed XML files.
- All are case-sensitive.
- All XML files are nested, consisting of Sections, Tags, and associated Values. Sections contain Tags, which include configuration Values for various elements. Elements for service, clientagent, and Log Sections also need entries in enumeration Sections at the start of the configuration files (see next bullet). Other elements do not need enumeration Sections. (Refer to Code Sample 7-1 for an example of enumeration and configuration Sections.)
- **All XML files dealing with service, clientagent, and Log must contain enumeration Sections.** These are special Sections which list all of the service, clientagent, and log elements which will be Sections later in the configuration file. These enumerations are almost like a table of contents to the rest of the Sections. There are currently three enumeration Sections: service, clientagent, and Log. Other types of configuration information do not require enumeration. (Refer to Code Sample 7-1.)

Code Sample 7-1 is a sample Section of the `loghelper.xml` file. Labels for Sections, Tags, and Values are shown in **bold**. The Log enumeration Section is shown first, followed by an attributes Section.

```
<Section> Log
    <Tag>log.Filter
        <Value>FilterOne</Value>
    </Tag>
</Section>
<Section>FilterOne
    <Tag>filterclass
        <Value>com.chordiant.core.log.LogFilter</Value>
    </Tag>
    <Tag>criteria
        <Value>com</Value>
    </Tag>
    <Tag>level
        <Value>error</Value>
    </Tag>
</Section>
```

Code 7-1: Excerpts from the loghelper.xml *File*

- **All Section names must be unique across all configuration files, unless you are intentionally overriding settings** (refer to the note on page 97). Within the enumeration Section, Tags should begin with *{XML file}*, not master, as shown here and in Code Sample 7-2:
  - **services**: *{XMLfile}*.name
  - **client agents**: *{XMLfile}*.agent
  - **logging**: *{XMLfile}*.filter

```
<Section> Log
   <Tag>mycomponentXML.Filter
      <Value>FilterTwo</Value>
   </Tag>
</Section>
```

Code 7-2: Sample Configuration File Section Showing Proper Naming Style

Note: When you want to override settings in one configuration file with Values from another configuration file (like `sitemaster.xml`), use the same Section and Tag names, but assign a different Value for the setting you are overriding. For information on overriding settings, refer to "components/{component}.xml" on page 98.

- **For some attributes, there are length limitations.** Service and client agent Tags are limited to 80 characters.

MASTER CONFIGURATION FILES

There are two types of master configuration files: `master.xml` and `components/master`.

Notes: Do not modify the `master.xml` or `components/master` files. Use additional files described in the following sections to specify configuration settings.

The `master.dtd` contains values that are used throughout the XML configuration files. This file is meant to be altered. For more information, refer to "master.dtd" on page 99.

The `master.xml` file is the standard starting point for the Chordiant 5 Foundation Server. It contains the basic configuration for the system.

`components/master` is a directory full of standard configuration files, grouped by functionality. As described below, you can add configuration files to the `components` directory, but you should not add to or modify the contents of the `components/master` directory.

components/*{component}*.xml

We recommend that you make all modifications outside the `master.xml` and `components/master` configuration files. You can use component files to override or append to master XML configuration files. For example, if you want to change the way the logging Section works, you can make a `logging.xml` file (or another name of your choosing) to override that Section of master XML configuration files. Component files can have one or more Sections.

Notes: When adding your own XML files, be sure to conform to the styles described in "Chordiant XML Configuration File Style" on page 96.

To override a Section in a master configuration file, use the same Section name and Tag name, but assign a different Value to the Tag.

You might choose to overwrite most of a *{component}*`.xml` file. In which case, you can copy the configuration file from the `components/master` directory to another directory, and make your minor modifications there. In this case, be sure to update the relative path to the `master.dtd` file. Refer to "master.dtd" on page 99 for more information.

Add your *{component}*`.xml` files in the *{CHORDIANT_ROOT}*`/config/Chordiant/components` directory. At startup, `master.xml` and `components/master` configuration files are loaded into cache, then all of the files within the component directory are loaded into cache.

- If the component files have Sections which appear in the master configuration files (`master.xml` and `components/master`), those master configuration Sections are *overwritten* with the component settings.
- If the component files have Sections which *do not appear* in master configuration files (`master.xml` and `components/master`), those Sections are *appended* to the configuration in the cache.

Note: The last configuration file to be read overlays any previously-read values. Refer to "ConfigurationHelper" on page 101 for the order in which all configuration files are read and overlaid.

Component files are overridden by the `sitemaster.xml` file, described in the next section.

sitemaster.xml

The sitemaster.xml file overrides the master configuration files (master.xml and components/master) and any {component}.xml files. It gives the site system administrator final control over the configuration. This administrator can specify any settings that are essential to the site, even if that means resetting a {component}.xml file's specifications.

If you choose to create a sitemaster.xml file, add it to the {CHORDIANT_ROOT}/config/Chordiant directory.

You do not have to include all Sections and Tags from the master.xml or {component}.xml files in the sitemaster.xml file. You should only include the specific Sections and Tags that you want to specify as different from those in the other XML configuration files.

Notes: Settings in the sitemaster.xml file cannot be overridden by any other XML configuration file.

When adding your own XML files, be sure to conform to the styles described in "Chordiant XML Configuration File Style" on page 96.

{nodename}.xml

The nodename is the name of a computer. You can make as many additional {nodename}.xml files as you want — each named after the computer which it affects. For example, you might have files on the main server called Cupertino1.xml or BostonServer5.xml. This customizes users' experiences from different locations. These additional XML files add to and override the information in cached configuration files (including master.xml, components/master files, {component}.xml, and sitemaster.xml) at runtime. {nodename}.xml files are located in {CHORDIANT_ROOT}/config/Chordiant.

Notes: When adding your own XML files, be sure to conform to the styles described in "Chordiant XML Configuration File Style" on page 96.

master.dtd

The master.dtd (document type definiton) file contains information that can be referenced in one or more XML configuration files. To make changes to the configuration, you only need to change the definition in the master.dtd file. The change will then ripple through any XML configuration files which point to the master.dtd.

The master.dtd file is located in {CHORDIANT_ROOT}/config/Chordiant/components/master, along with the master.xml file.

Referencing master.dtd

To have an XML configuration file point to the `master.dtd`, you must specify these lines at the top of the XML file, before the <ROOT> tag.

```
<?xml version="1.0"?>
<!DOCTYPE Root SYSTEM "../../master.dtd">
<Root>
```

Be sure to indicate the relative location of the `master.dtd` file by using the appropriate number of dots and slashes. In this example, the XML file is two directories below the `master.dtd`.

Syntax

Within the XML configuration file, to specify a value from the `master.dtd` file, you must use this syntax:

```
    <Value>&entityName;</Value>
```

Each entity name must begin with an ampersand (&) and end with a semicolon (;). If you have more than one entity name within the value, the names are separated by an additional semicolon, as shown here:

```
    <Value>&entityName;;&entityName;</Value>
```

Example

The `SecurityManager.xml` file points to several values within the `master.dtd` file. Here is one Tag for the authentication port number.

```
<Tag>authenticationPortnumber
    <Value>&SECURITY_AUTHENTICATE_PORT;</Value>
</Tag>
```

The corresponding tag can be found in the `master.dtd` file:

```
<!ENTITY SECURITY_AUTHENTICATE_PORT "1389">
```

So the value for the authentication port number in the Security Manager will be 1389.

Note: You can see the real values substituted for the `master.dtd` entities if you open the XML configuration files within a web browser. If you open them in a text editor, you will see the pointers to the `master.dtd` file.

If you change values in the `master.dtd` file, go to the Administrative Console to refresh the ConfigurationHelper. For information on the Administrative Console, refer to "Chordiant 5 Foundation Server Administration" on page 63.

ConfigurationHelper

This utility reads information directly from the combined view of the configuration files and returns it to other tools within the Chordiant 5 Foundation Server.

On startup, the ConfigurationHelper **performs these steps to create the combined configuration view:**

1. Reads `master.xml` into memory.
2. Overlays and merges the in-memory `master.xml` with all the XML files in *`{CHORDIANT_ROOT}`*`/config/Chordiant/components/master` directory.

   `{CHORDIANT_ROOT}` corresponds to the chordiant.configuration.configurationRootDirectory parameter in your application server. Refer to "Configuration Files and the ConfigurationRootDirectory" on page 46 for more information.
3. Overlays and merges the in-memory `master.xml` with all the XML files in *`{CHORDIANT_ROOT}`*`/config/Chordiant/components` directory.
4. Overlays and merges the in-memory `master.xml` with `sitemaster.xml`.
5. Overlays and merges the in-memory `master.xml` with *`{nodename}`*`.xml`, if available.

Refer to "GatewayHelper" on page 60 for more information.

Adding Components through Configuration

You must configure the Chordiant 5 Foundation Server system to include information about new applications, services, and other components that you develop to run with the system. You do this by adding component files to override the `master.xml` and `components/master` configuration files.

The `master.xml` and `components/master` files control the configuration for the entire system. You can also create an unlimited number of `components/`*`{component}`*`.xml` and *`{nodename}`*`.xml` files which add to and override the settings in the master XML configuration files. For more information, refer to "Configuration Files" on page 95.

Note: Do not change the `master.xml` or `components/master` files. Make your modifications by adding *`{component}`*`.xml` files in the *`{CHORDIANT_ROOT}`*`/config/Chordiant/components` directory or adding a `sitemaster.xml` file to the *`{CHORDIANT_ROOT}`*`/config/Chordiant` directory.

To avoid modifying the `master.xml` or `components/master` files, use a text editor to copy the Section you want to modify or enhance. Paste it into a new document and save it as *`{component}`*`.xml`, where *`{component}`* is a name you choose to describe this Section, often the name of a service or functionality. If you do this, be sure to update the relative path to the `master.dtd` file, as described in "Referencing master.dtd" on page 100.

You can also edit the *{nodename}*.xml or sitemaster.xml file directly.

To configure the system for your new components:

1. Locate <Section>clientagents in the configuration file. Copy the entire Section and paste it into a new XML document which you can modify. Here, we'll call that file TestClientAgents.xml.

   To add Services, proceed directly to Step 4 on page 103.

   For example, to add information about two Client Agents, TestClientAgent and TestClientAgent2, add Code Sample 7-3 to a new TestClientAgents.xml file.

```
<Section>clientagents
   <Tag>TestClientAgents.agent
      <Value>TestClientAgent</Value>
   </Tag>
   <Tag>TestClientAgents.agent
      <Value>TestClientAgent2</Value>
   </Tag>
</Section>
```

Code 7-3: Adding ClientAgent Configuration Information

Notes: Tags within this enumeration Section should be named after the name of your XML configuration file. Here, they are named after the file TestClientAgents.XML. This ensures that they will not conflict with standard client agents provided in master.xml or components/master files.

Values for client agents are limited to 80 characters.

2. Add new Sections (child sections of root) to the `TestClientAgents.xml` configuration file to further define the client agents.

   For example, to include additional information about TestClientAgent and TestClientAgent2, add Code Sample 7-4 to the new `TestClientAgents.xml` configuration file, below the enumeration Section you created in Step 1.

```
<Section>TestClientAgent
    <Tag>classname
        <Value>test.jx.simple.TestClientAgent</Value>
    </Tag>
    <Tag>stubtype
        <Value>EJBStub</Value>
    </Tag>
</Section>
<Section>TestClientAgent2
    <Tag>classname
        <Value>test.jx.simple.TestClientAgent2</Value>
    </Tag>
    <Tag>stubtype
        <Value>EJBStub</Value>
    </Tag>
</Section>
```

Code 7-4: Adding Detailed ClientAgent Configuration Information

   Notice that the stubtype must be one of the configured stub types, in this case EJBStub. For more information, refer to "Transactions in Chordiant Foundation Server" on page 118 and "Configuring SmartStubs" on page 124.

3. Save and close the `TestClientAgents.xml` file.

4. Locate <Section>services in the `master.xml` configuration file or in one of the `config/components/master` configuration files. Copy the Section and paste it into a new XML document which you can modify. Here, we'll call that file `TestServices.xml`.

   If you are not adding services, continue with Step 7 on page 104.

   For example, to add information about two services, TestService and TestService2, add Code Sample 7-4 to the new `TestServices.xml` configuration file.

```
<Tag>TestServices.name
    <Value>TestService</Value>
</Tag>
<Tag>TestServices.name
    <Value>TestService2</Value>
</Tag>
```

Code 7-5: Adding Service Configuration Information

Notes: Tags within this enumeration Section should be named after the name of your XML configuration file. Here, they are named after the file `TestServices.xml`. This ensures that they will not conflict with standard services provided in `master.xml`.

Values for services are limited to 80 characters.

5. Add new Sections (child sections of root) to the `TestServices.xml` configuration file to further define the services.

   For example, to include additional information about TestService and TestService2, add Code Sample 7-6 to the `TestServices.xml` configuration file, below the enumeration Section you created in Step 4.

```
<Section>TestService
   <Tag>classname
      <Value>test.jx.simple.TestService</Value>
   </Tag>
   <Tag>ConnectionName
      <Value>EJBBMT</Value>
        -- OR --
      <Value>EJBCMTRequired</Value>
   </Tag>
</Section>
<Section>TestService2
   <Tag>classname
      <Value>test.jx.simple.TestService2</Value>
   </Tag>
   <Tag>ConnectionName
      <Value>EJBBMT</Value>
        -- OR --
      <Value>EJBCMTRequired</Value>
   </Tag>
</Section>
```

Code 7-6: Adding Detailed Service Configuration Information

   Notice that you must specify whether the service is to be in the JX EJB deployed as a Bean Managed Transaction (BMT) or a Container Managed Transaction Required type (CMTRequired). For more information, refer to "Transactions with the JX EJB" on page 20.

6. Save and close the `TestServices.xml` file.

Note: The default security setting enables all users to call all APIs on all services. Therefore, without doing anything, your new service is accessible to all users, so you should be able to test it easily. To add access control to your services, follow the steps defined in "Adding a New Service as a Resource" on page 271.

7. **At this point, you need to deploy your service code (as JAR files or class files) and make it available to the application server.** You might have to restart the application server, depending on how you deployed your service code and how you defined the class path on the application server.
8. Run your client application.
9. Verify the output of your application, if applicable.

10. Verify the output of the J2EE Application Server.

Note: How you deploy your project depends on the application server you are using. Refer to your application server's documentation for details on deploying.

AUDITING FOR PERFORMANCE

When you are testing your new additions to the system, you will probably want to use auditing to test your additions. This is done through configuration.

Note: We advise that you do not alter the `master.xml` or `component/master` files. Make any additions and modifications in `component/{component}.xml`, `{nodename}.xml`, or `sitemaster.xml` files. Refer to "Configuration Files" on page 95 and "Adding Components through Configuration" on page 101 for details.

---

distributedaudit

The distributedaudit function provides performance information in addition to standard performance logging. For information on performance logging, refer to "Performance" on page 50.

Note: You should only use the distributedaudit function as an initial indicator of performance problems. It should *not* be turned on in a deployment environment.

Use the distributedaudit Section of the `performance.xml` configuration file to create a log message that includes the service, function, duration, and message size (in characters).

Figure 7-1 illustrates an application calling a client agent. The client agent calls across the network to a service in an EJB. As part of its function, the service then queries a database.

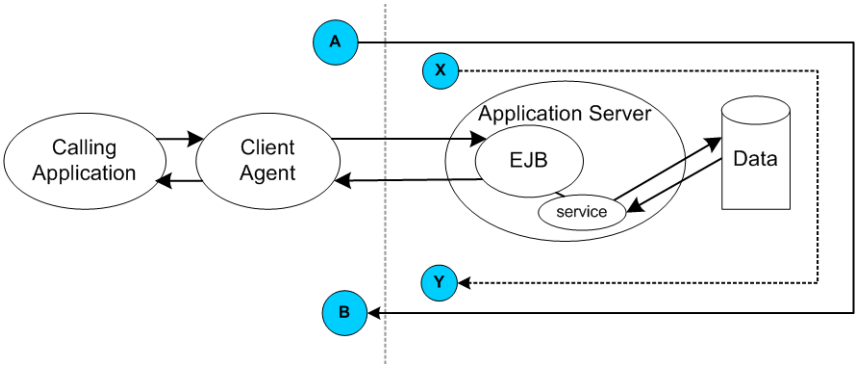


Figure 7-1: Sample Interaction Audited by distributedaudit

It is helpful to use distributedaudit to measure:

- **Client-Side Timing** — Client agent calling the service and then receiving a call back, illustrated in Figure 7-1 as Point A to Point B. This includes time spent querying the database as well as the time spent over the network between the client agent and the service.
- **Server-Side Timing** — Service receiving a call from the client agent and then returning the call back to the client agent, illustrated in Figure 7-1 as Point X to Point Y. This isolates the time spent performing the function on the service.

Controlling distributedaudit

You control the distributedaudit functionality in two places:

- **on or off —** `{CHORDIANT_ROOT}/config/Chordiant/component/master/Performance.xml`
- **settings, if turned on —** `{CHORDIANT_ROOT}/config/Chordiant/component/master/loghelper.xml`

Turning distributedaudit On and Off

To specify whether distributedaudit is turned on, open the `performance.xml` file and modify the Value of the ClientCall Tag, ServerCall Tag, or both. These two Tags operate independently, enabling you to audit calls on just the client side, just the server side, or both sides together. A Value of True means that distributedaudit functionality is turned on. Code Sample 7-7 shows that server-side auditing is available, while client-side auditing is not.

```
<Section>Performance
    <Tag>ClientCall
        <Value>False</Value>
    </Tag>
    <Tag>ServerCall
        <Value>True</Value>
    </Tag>
</Section>
```

Code 7-7: Performance Section of performance.xml Configuration File

Filtering the Output

If distributedaudit is turned on, as described in "Turning distributedaudit On and Off", you can control which messages are logged using entries in the `loghelper.xml` file. Filters are separated for client-side and server-side auditing.

Note: If distributedaudit is not turned on, you will not receive any output, regardless of which filters you specify here.

Specifying the criteria for distributedaudit is similar to specifying the criteria for other types of logging. However, instead of giving a package name, as shown in "Criteria Details" on page 56, you specify a hard-coded string for either client-side or server-side auditing. The criteria value for auditing has three parts: *{hard-coded string}.{servicename}.{methodname}*

- **hard-coded string for either client-side or server-side auditing:**
  - SMARTSTUB_DIST_AUDITING: client side
  - EJB_DIST_AUDITING: server side
- **service name** — specifies the service to audit.
- **method name** —specifies the method to audit.

These three parts work together to provide targeted auditing. The more you specify, the more targeted the auditing information you receive. The fewer you specify, the more output you receive.

- If no *service* is specified, *all services* on the specified client or server side are audited.
- If no *method* is specified, *all methods* on the specified service are audited.

Once you have specified the criteria, you must set the Value for this Criteria to perf.

Code Sample 7-8 shows performance filtering for the client side.

```
<Tag>criteria
     <Value>SMARTSTUB_DIST_AUDITING.{servicename}.{methodname}</Value>
</Tag>
<Tag>level
     <Value>perf</Value>
</Tag>
```

Code 7-8: Section of loghelper.xml for Filtering Client-Side Performance Statistics

Code Sample 7-9 shows performance filtering for the server side.

```
<Tag>criteria
     <Value>EJB_DIST_AUDITING.{servicename}.{methodname}</Value>
</Tag>
<Tag>level
     <Value>perf</Value>
</Tag>
```

Code 7-9: Section of loghelper.xml for Filtering Server-Side Performance Statistics

Format of distributedaudit Message

The distributedaudit output contains the following parts. The important information begins after the <PTH_STATISTICS portion.

```
<PTH_STATISTICS
user
service.function name
data size of request
output data size
<field for internal use only>
EJB_PERF_LOGGING or SMARTSTUB_PERF_LOGGING, showing which side is audited
Startdate
Starttime
Enddate
Endtime
starttime in ms
duration in ms
error (n = no error)
```

Code Sample 7-10 shows a sample distributedaudit message for the client side, the important information shown in bold.

```
06_clone1.log:[6/17/04 13:34:01:825 PDT] 629a1cd5 SystemOut     O <Thu Jun 17 13:34:01 PDT 2004> <1087504441825>
<PERF> <Thd=Servlet.Engine.Transports : 5> <com.chordiant.session.service.SessionService.removeSession()>
<PTH_STATISTICS,ccagent1,SessionService.removeSession,100,45,1,
SMARTSTUB_PERF_LOGGING,06/17/2004,13:34:01,06/17/2004,13:34:01,763,61,N>
```

Code 7-10: DIstributedAudit Message for Client Side (SMARTSTUB)

Code Sample 7-11 shows a sample distributedaudit message for the server side, the important information shown in bold.

```
06_clone1.log:[6/17/04 13:34:01:825 PDT] 629a1cd5 SystemOut     O <Thu Jun 17 13:34:01 PDT 2004> <1087504441825>
<PERF> <Thd=Servlet.Engine.Transports : 5> <com.chordiant.session.service.SessionService.removeSession()>
<PTH_STATISTICS,ccagent1,SessionService.removeSession,100,45,1,EJB_PERF_LOGGING,06/17/2004,13:34:01,06/17/2004,
13:34:01,773,50,N>
```

Code 7-11: DIstributedAudit Message for Server Side (EJB)

Auditing and Debugging Transactions

There might be times when something with a transaction doesn't seem to be working as you would expect. With both Container Managed Transactions (CMTs) and Bean Managed Transactions (BMTs) available, it can be useful to see if the JX Infrastructure is routing a particular service call to either the CMT or BMT EJB that you are expecting.

You can see exactly which EJB the call is going through by using a transaction debug configuration file. This configuration will print a debug message to standardout for every call that is made to every service.

Code Sample 7-12 shows a sample *{component}*`.xml` file, called `service_tx_debug.xml`.

```
<Section>Log
    <Tag>service_tx_debug.Filter
        <Value>service_tx_debug1</Value>
    </Tag>
    <Tag>service_tx_debug.Filter
        <Value>service_tx_debug2</Value>
    </Tag>
</Section>
<Section>service_tx_debug1
    <Tag>filterclass
        <Value>com.chordiant.core.log.LogFilter</Value>
    </Tag>
    <Tag>writer
        <Value>com.chordiant.core.log.LogWriterStandardOut</Value>
    </Tag>
    <Tag>criteria
        <Value>com.chordiant.service.ejb.EJBGatewayServiceBean.processRequestObject</Value>
    </Tag>
    <Tag>level
        <Value>debug</Value>
    </Tag>
</Section>
```

Code 7-12: Sample service_tx_debig.xml File

```
<Section>service_tx_debug2
    <Tag>filterclass
        <Value>com.chordiant.core.log.LogFilter</Value>
    </Tag>
    <Tag>writer
        <Value>com.chordiant.core.log.LogWriterStandardOut</Value>
    </Tag>
    <Tag>criteria
        <Value>com.chordiant.service.ejb.EJBGatewayServiceBean.processRequestXMLString</Value>
    </Tag>
    <Tag>level
        <Value>debug</Value>
    </Tag>
</Section>
```

Code 7-12: Sample service_tx_debig.xml File (Continued)

Format of Transaction Auditing Message

Code Sample 7-13 shows the output lines for the audit. Output is written to standardout.

```
...<DEBUG>...<Transaction management state is: [BMT] for service [HelloWorldService], method [DOIT]>
...<DEBUG>...<Transaction management state is: [BMT] for service [HelloWorldService],
      method [DOIT_CALLBACK]>
```

Code 7-13: Transaction Auditing Format

For more information on CMTs and BMTs, refer to "Transactions with the JX EJB" on page 20.

Chapter 8

Creating Foundation Server Components

When you create a Chordiant application, you use many components, including services and client agents, and possibly custom objects. You must also use the Chordiant Resource Manager to make the system aware of new components that you create. There are special interactions between these components — callbacks and service to service calls — that you will likely want to use in your solution.

You can hand-code the services and client agents, as described in this chapter, or you can model them in Rational Rose and then use Chordiant's Business Component Generator to create the services, client agents, constants class, and configuration file. You must still fill in the logic for the service, but the process is much quicker. Even if you choose to use the Business Component Generator, the details in this chapter can help you understand how services and client agents work, as well as how they interact with the Resource Manager.

BUILDING AN APPLICATION

Figure 8-1 illustrates the process of building an application using Chordiant 5 Foundation Server.

Building an Application Using Chordiant 5 Foundation Server

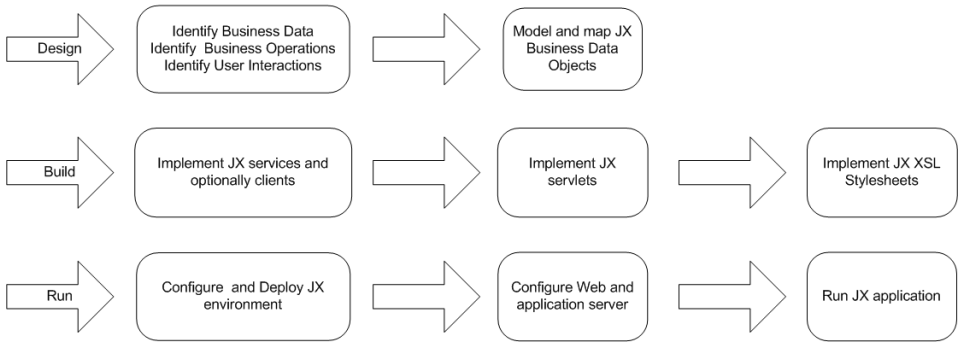


Figure 8-1: Using the Chordiant 5 Foundation Server

Customization Philosophy

The model for customizing components within the JX architecture is to subclass existing classes and make your modifications there. Do not alter the functionality in existing classes. For information on customizing JX services and application components, refer to the *Chordiant 5 Foundation Server Application Components Developer's Guide*.

Generating Java Components from Design Tools

You can also create components quickly and easily from within Rational Rose using Chordiant's UML Extender for Rational Rose along with Chordiant's Business Component Generator. Refer to the *Chordiant 5 Foundation Server Application Components Developer's Guide* for details. This process eliminates the need for most of the hand-coding described in this chapter.

Javadoc

Refer to Javadoc for details on public components.

Javadoc is installed on your computer through the installation under `{WORKSPACE}/documentation/FoundationServer/Javadoc`.

Different Javadoc files are available for different areas of the Chordiant 5 Tools Platform and Foundation Server. Locate the correct project within the `Javadoc` directory.

Example Code

Example code is provided for services, client agents and a sample application. Example code is included in the `Documentation/Samples/Services` directory on the Installation CD. You can also access this directory through the Chordiant Tools Platform under **Help | Help Contents**.

BUILDING A SERVICE

This section describes the process of building a service using Chordiant 5 Foundation Server, in the context of a sample service and application.

All business services should have a corresponding client agent to expose the API on the client side. For information on building a client agent, refer to "Building a Client Agent" on page 134.

Business Service Structure

Before you build a service, it is helpful to have an overview of its structure.

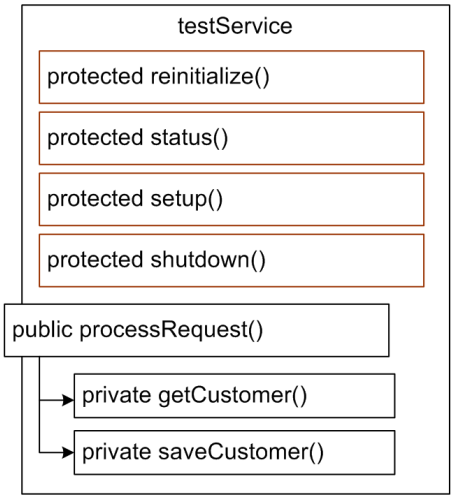


Figure 8-2: Structure of a Business Service

Services consist of these main parts:

- **protected service control methods**— These include setup, reinitialize, status, and shutdown. They are called automatically by the infrastructure.
- **processRequest method**—This public method serves as an entry point into the service. It calls the private methods.
- **private methods**—Do the work of the service. They are called by the single entry point, processRequest.

Creating a Service

Tip: Remember to familiarize yourself with the existing business services before creating your own service. You might be able to use some functionality from an existing business service.

You can also create business service skeletons automatically using Rational Rose and the Business Component Generator. For details, refer to the *Chordiant 5 Foundation Server Application Components Developer's Guide.*

To create a service:

1. Create a public class that extends the ServiceBaseClass, as shown in Code Sample 8-1.

```
public class TestService extends ServiceBaseClass{
    . . .
}
```

Code 8-1: TestService Extends the ServiceBaseClass

Note: The example services created in this chapter extend the ServiceBaseClass. All Chordiant-provided business services extend the BusinessDataServiceBaseClass.

The BusinessDataServiceBaseClass has extra features used for proper business services, like Resource Managers and caching. You can use the extra features of this base class or use the ServiceBaseClass to create your own business or system services.

2. Define the CLASS_NAME and the PACKAGE_NAME as constants.

   The `CLASS_NAME` constant is used to hold the value of the service name specified in the XML configuration files. Client agents use this constant to specify the desired service class name within the processRequest method. The package name is commonly used for logging purposes.

   The class name constant must match the service name specified in the `master.xml` or other configuration file. Client applications use this name when calling the processRequest method in the client agent.

   Code Sample 8-2 shows constants you might add within the service class definition.

```
public final static String CLASS_NAME = "TestService";
public final static String PACKAGE_NAME = "john.simple.jx.test";
```

Code 8-2: Sample Constants

3. Define the method name constants.

   Client applications use these method name constants when calling the processRequest method in the client agent. The constants are used by processRequest to dispatch the request to the proper method.

   You can choose to define the method name constants in a separate class, or within the service class definition itself.

   Defining constants within a separate class:

   Client agents can then use this same class to access the method names for this service. A separate class for method constants is created for you when using the Business Component Generator.

   To define the method constants, create a new class, named *{yourservice}*`constants.java`. Code Sample 8-3 shows an example of a constants class.

```
package john.simple.jx.test.constants;


public class ServiceHistoryConstants {

public final static String FUNCTION_DOHELLOECHO = "DOHELLOECHO";
public final static String FUNCTION_DOCALLBACK = "DOCALLBACK";
public final static String FUNCTION_DOSERVICE2SERVICECALL = "DOSERVICE2SERVICECALL";

}
```

Code 8-3: New Constants Class

   Defining constants within the service class definition:

   If you define the method constants within the service class definition, the client agent cannot access these definitions. You must also define them within the client agent.

   Your entry in the service class definition might look similar to Code Sample 8-4.

```
public final static String FUNCTION_DOHELLOECHO = "DOHELLOECHO";
public final static String FUNCTION_DOCALLBACK = "DOCALLBACK";
public final static String FUNCTION_DOSERVICE2SERVICECALL = "DOSERVICE2SERVICECALL";
```

Code 8-4: Constants in the Service Class Definition

4. Implement the processRequest method.

   The public processRequest method serves as the single entry point for the service. A typical implementation for this method is to dispatch the incoming request to local private methods defined within the class, based on the function name parameter.

   Code Sample 8-5 shows an example segment within the processRequest method.

```
public Object processRequest(
    String username,
    String authentication,
    String serviceName,
    String functionName,
    Object payload)
    throws Throwable
{
    final String METHOD_NAME = "processRequest";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    Object retval = null;
    PayloadData requestPayload = null;
    PayloadData responsePayload = null;

    if ((functionName != null) && (functionName.length() > 0))
    {
        try
        {
            // Do a simple "if statement" dispatcher
            if (functionName.compareToIgnoreCase(FUNCTION_DOHELLOECHO) == 0)
            {
                // Cast the payload as needed.
                requestPayload = (PayloadData)(payload);

                // Pull any required parameters out of the payload
                // as needed for the typed local function.
                String theInputData = (String)(requestPayload.
                    getDataWithName("theParameterName"));

                // Call the specific local function
                String theOutputData = dohelloecho(theInputData);

                // Reuse the input payload for the return payload by clearing it out.
                requestPayload.removeAllData();
                responsePayload = requestPayload;

                // Fill the return payload with the appropriate parameters.
                responsePayload.putDataWithName("theParameterName",theOutputData);

                // Assign the response payload to the return value of this method.
                retval = responsePayload;
            }
            else if (functionName.compareToIgnoreCase(FUNCTION_DOCALLBACK) == 0)
    // additional if statements and dispatching to follow...
```

Code 8-5: Sample processRequest Method

When coding the dispatch logic, use the method name constants that you defined in Step 3.

Tip: JX architecture enables method authorization. Users can be granted or denied permission for specified methods. Refer to "Security" on page 253 for more information.

5. Optionally, implement the remaining interface for the class.

   These four service control methods are called automatically by the infrastructure. You do not write code to call these methods, but you can control what they do. It is up to you how you want to use them, if at all.

   - **setup**—initializations, including start caching and getting a Resource Manager. This is called once when the application starts up.

     The BusinessDataServiceBaseClass includes functionality to code caching for static data.

   - **reinitialize**— to reset the service to its original starting state without shutting it down, for example refreshing the cache. This method can be called several times.
   - **status**—to assess status while the service is running. This method can be called several times.
   - **shutdown**—to provide a clean shut down when the service has finished its function. This is called once just before exit.

   For an example implementation of these methods, refer to the example code in the `Documentation/Samples/Services` directory on the Installation CD. You can also access this directory through the Chordiant Tools Platform under **Help | Help Contents**.

   Refer to Chapter 6, "Chordiant 5 Foundation Server Administration" for details on how the Administrative Console uses these service control methods.

6. Implement the local methods that perform the useful work of the service.

   For example, based on the method constants defined in Step 3, you would implement these local methods:

   - docallback
   - dohelloecho
   - doservice2servicecall

   For an example implementation of these methods, refer to the example code in the `Documentation/Samples/Services` directory on the Installation CD. You can also access this directory through the Chordiant Tools Platform under **Help | Help Contents**.

7. **Define the transaction type in the XML configuration file.** The transaction can be either bean managed (Value=EJBBMT) or container managed (Value=EJBCMTRequired). For details, refer to "Transactions in Chordiant Foundation Server" on page 118.

8. **Register the service class name into an XML configuration file for instantiation.** Refer to "Adding Components through Configuration" on page 101 for instructions.

Exceptions

For details on exceptions, refer to "Exceptions and Error Handling" on page 92.

Locking

By default, all Chordiant business services implement optimistic locking at the top of the object graph. You can choose to change this locking strategy if you want. Services which are not subclassed from the business service base class do not have a default locking strategy. For more information on locking strategies, refer to "Optimistic and Pessimistic Locking" on page 197.

Accessing Data Stores

Refer to the "Chordiant Persistence Server" on page 181 for information on how business services can access persistent data.

Server-Side Business Object Behavior

Business objects are primarily used to carry data between the client and server sides of the application. Server-side business object behavior (BO behavior or BOB) objects contain behavior that is specific only to the business object data. In other words, the behavior required for the business object is located in the BO behavior class. Business objects, together with Business Object behavior classes and Business Object Criteria classes can be designed using object-oriented tools. Server-side business object behaviors are used for the Party Management Facility.

For more information on creating server-side business object behavior, refer to the *Chordiant 5 Foundation Server Application Components Developer's Guide*.

Transactions in Chordiant Foundation Server

The single JX EJB has two public interfaces: one for Java object-oriented calls and one for XML-oriented calls.

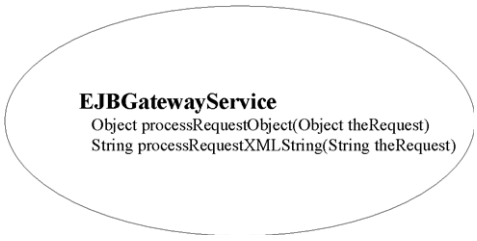


Figure 8-3: The JX EJB

As described in "Transactions with the JX EJB" on page 20, this single Chordiant JX EJB is deployed twice: once as a BMT EJB and once as a CMT EJB, with the J2EE Required attribute on both public interfaces shown above.

This enables individual JX services to:

- Configure for BMT and then use the J2EE UserTransaction interface explicitly within Java code.

  Dependent transactions across BMT JX service instances are *not* supported.

  -OR-

- Configure for CMTRequired and then use the J2EE implicit application server transaction functionality which will either continue an existing transaction across JX service instance calls or start a new transaction if one is not present.

  Dependent transactions across CMT JX service instances *are* supported.

Note: A BMT JX service that starts a transaction explicitly using UserTransaction can have that transaction continue (in a dependent manner) if it calls to one or more CMT JX Services.

Transaction Control Mechanism

Client agents interact with JX services through processRequest method, passing payload data through to the service.

- Some client agents, like an account management client agent, are strongly typed and interact with only one type of EJB — either BMT or CMT.
- Other client agents, like the XML client agent, are "typeless" and can interact with both BMT and CMT EJBs.

By default, all client agents use the EJBStub. (Other stub types are also available. Refer to "Configuring SmartStubs" on page 124 for more information.) The EJBStub has connections to all JX EJB deployments. The EJBStub determines which JX EJB deployment to call based on:

- the servicename tag passed in the client agent's payload data,

  and

- the service's transactional configuration.

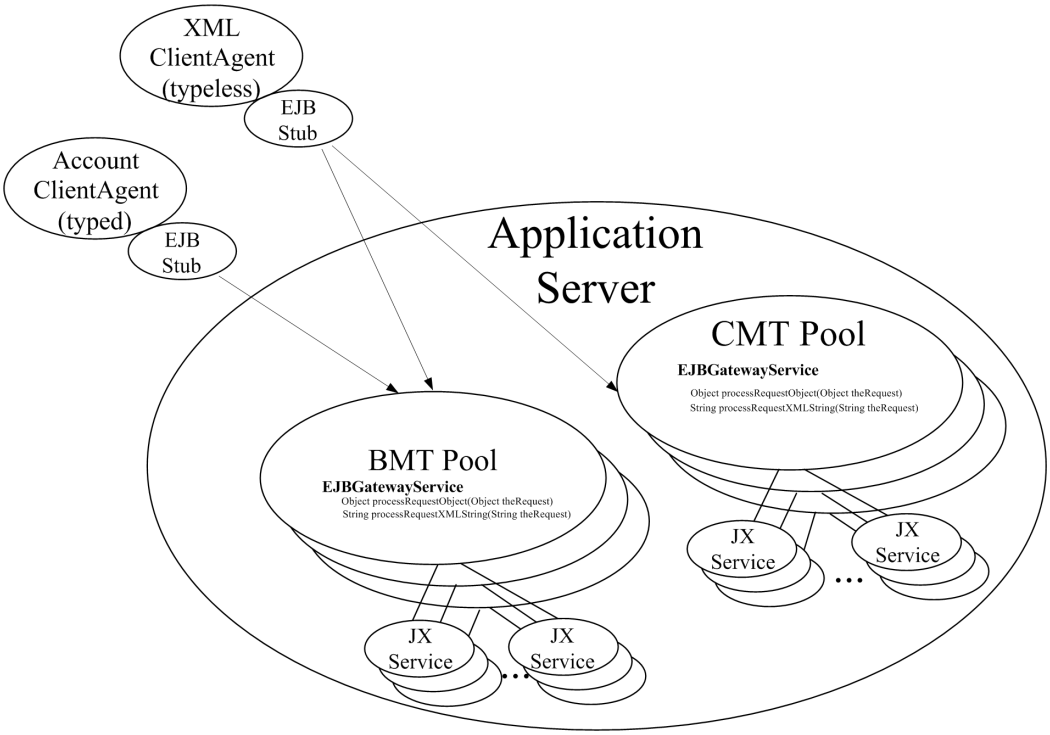


Figure 8-4: EJBStub Makes Calls to Appropriate Deployment

Therefore, in the Foundation Server *{component}*.xml configuration files, you must specify that:

- the client agent use EJBStub (this is the default stubtype)

```
<Section>clientagents
       <Tag>component.name
          <Value>MyClientAgent</Value>
       </Tag>
</Section>
<Section>MyClientAgent
   <Tag>classname
       <Value>com.chordiant.service.MyClientAgent</Value>
   </Tag>
   <Tag>stubtype
       <Value>EJBStub</Value>
   </Tag>
</Section>
```

- the service is either BMT or CMTRequired

```
<Section>services
       <Tag>component.name
          <Value>MyService</Value>
       </Tag>
</Section>
<Section>MyService
   <Tag>classname
       <Value>com.chordiant.service.MyService</Value>
   </Tag>
   <Tag>ConnectionName
       <Value>EJBBMT</Value>
         -- OR --
       <Value>EJBCMTRequired</Value>
   </Tag>
</Section>
```

Note: All Chordiant-deployed EJBs must use the EJBStub stubtype. You can use other stubtypes for other services, if you choose. Refer to "Configuring SmartStubs" on page 124 for more information on additional stub types.

Notice that this is done in the configuration files, so you do not have to make changes to your individual services.

Refer to "Adding Components through Configuration" on page 101 for more information on configuration files.

Rollbacks

If an exception is allowed to escape from a JX service under the JX CMT EJB, and the Transaction Rollback Strategy in the service's configuration is set to ALL (refer to "Configuring for Rollbacks" below), the JX CMT EJB top-level exception handler will call sessionContext.setRollbackOnly, which causes the J2EE container to call the rollback method on the active transaction. The original JX service exception is still thrown and delivered to the caller.

- If a CMT JX service wishes to "announce" an error condition to a calling client *without causing a rollback* on the active transaction, it must do so by returning a natural error code in the returned PayloadData. It should not throw an exception.
- If a CMT JX service wishes to "announce" an error condition that will cause the active transaction to roll back, it must do so by throwing an exception from its processRequest method.

Configuring for Rollbacks

You can set the way you want to handle rollbacks in your Container Managed Transaction (CMT) through configuration. In the services Section of the `{component}.xml` file, you can use the Transaction_Rollback_Strategy tag. This tag specifies how the CMT will handle a rollback.

If an exception is thrown in a CMT service and escapes to the top layer JX CMT EJB, there are two behaviors that can occur:

- ALL—All exceptions will cause a transaction rollback
- J2EE—Only system exceptions will cause a transaction rollback, as per the J2EE EJB specification.

Notes: ALL is the original default value and is the default if this tag is missing from the configuration file.

If a tag has a value other than the two valid values, an error message is written and the behavior defaults to ALL.

Regardless of the setting, the service can still call sessionContext.setRollbackOnly to roll back the entire transaction, even if there isn't an exception.

Here is an example of the tag in a sample service Section of a *{component}*.xml configuration file:

```
<Section>MyService
    <Tag>classname
        <Value>com.chordiant.service.MyService</Value>
    </Tag>
    <Tag>Transaction_Rollback_Strategy
        <Value>ALL</Value>
    </Tag>
    <Tag>ConnectionName
        <Value>EJBCMTRequired</Value>
    </Tag>
</Section>
```

For more information on error handling, refer to "Exceptions and Error Handling" on page 92. For more information about configuring services, refer to Chapter 7, "Configuration Files", especially "Adding Components through Configuration" on page 101.

Tip: When debugging your CMT services, you might want to specify a longer JTA timeout in your application server. This way, transactions will not be rolled back automatically before you have completed your testing.

Configuring SmartStubs

Smartstubs help the client agent communicate with the JX service. The client agent does not need to know what kind of service it is contacting— SmartStubs handle the communication information.

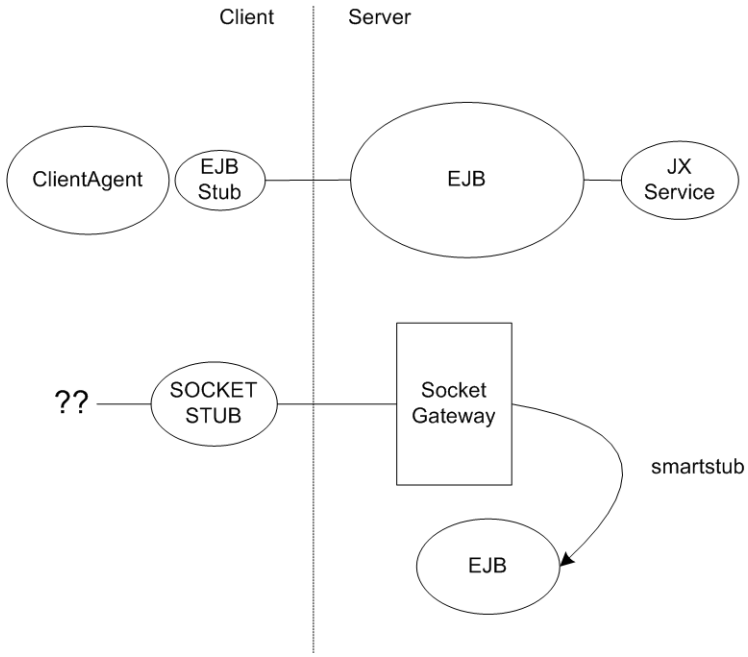


Figure 8-5: Client Agents Using SmartStubs to Contact Services

SmartStubs are configured in both the `smartstubs.xml` file and in the clientagent and services Sections of the *`{component}`*`.xml` files.

Chordiant provides four types of SmartStubs:

- EJBStub (default – used by Chordiant-provided business services)
- RemoteEJB
- SOCKETSTUB
- RMIService

EJBStub

The EJBStub is the default smartstub configuration. It is used for normal communications between

- thick clients and services
- servlet JSPs and services
- services and other services

These entities must be within the same JNDI space.

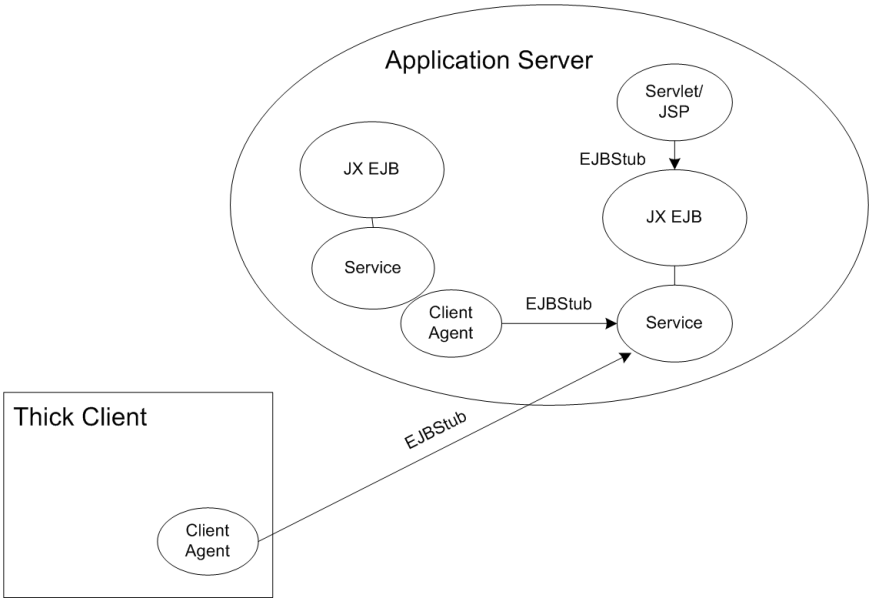


Figure 8-6: Using EJBStub Smartstub

RemoteEJB

RemoteEJB is used when the entities mentioned above (in "EJBStub") are in different JNDI spaces, perhaps due to two deployments in two different (heterogeneous) application servers.

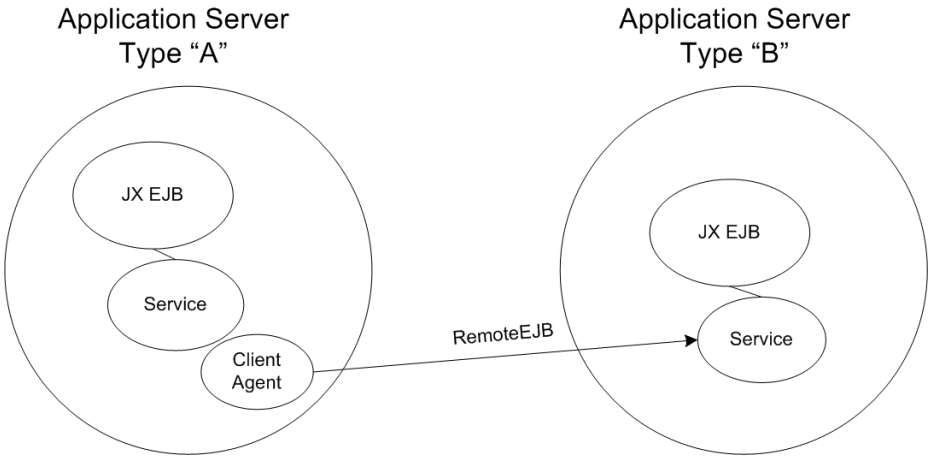


Figure 8-7: Using RemoteEJB SmartStub

SOCKETSTUB and RMIService

For callbacks, Chordiant 5 Foundation Server uses network presence.

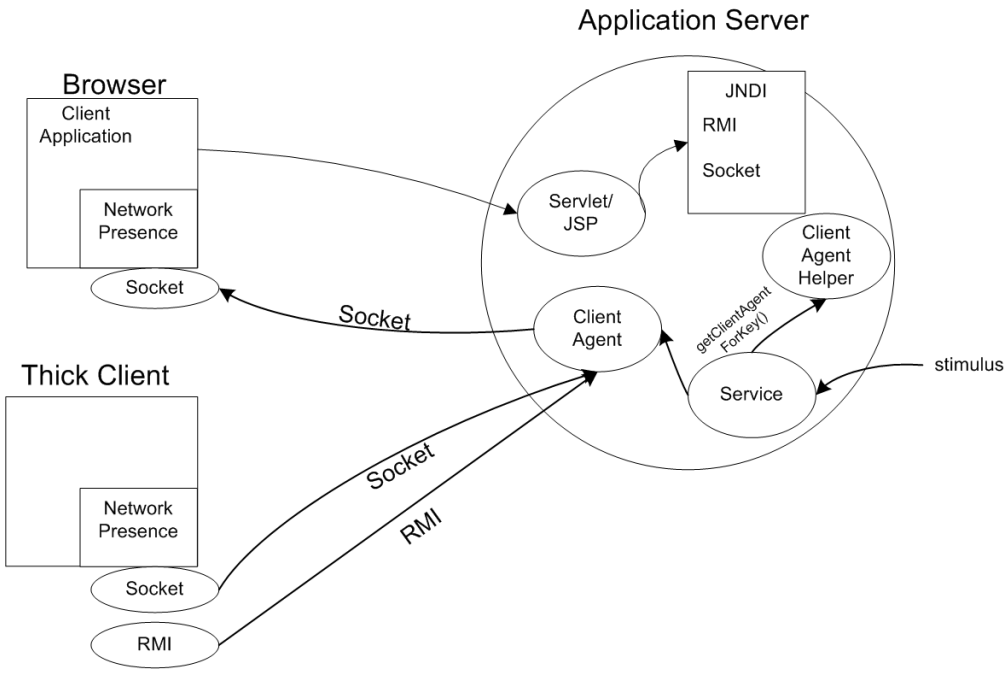


Figure 8-8: Communication for Callbacks

- On thin clients using browsers, the network presence is a socket server and is registered with JNDI as a socket entity.
- On thick clients, network presence is established using the GatewayHelper, which can be configured (for thick clients only) to use a socket server or an RMI server, and will register with JNDI as appropriate. For server-side components which perform callbacks to thick clients, the callback client agent will automatically be initialized to use the SOCKETSTUB or RMIService as appropriate.

Another reason you might use SOCKETSTUB is that you might not be able to use the application server's ORB. This can be caused by dissimilar versions of JDK. Here are two examples.

- If you are developing a thick client with JDK "A" and want to test with an application server with JDK "B", you might not be able to use EJBStub. Since socket communication is a common denominator between all JDK versions (and the fact that Chordiant provides this facility), you can communicate with the application server through sockets and SOCKETSTUB. One example where this can be useful is if you are developing a thick client in an IDE that cannot

support native communications with the application server (perhaps a JDK version mismatch), yet you still want to be able to run and debug your client from within the thick client IDE.

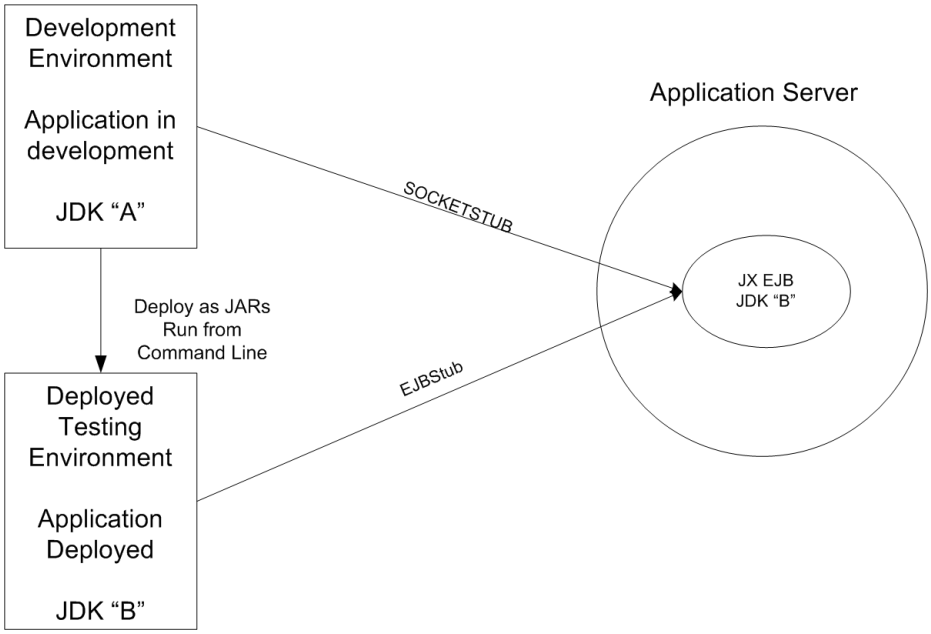


Figure 8-9: Using Sockets During Development

- Another example of where you might use the SocketSmartStub is if you have a legacy Java application on an old JDK version that must communicate with a newer application running a newer version of JDK.

Creating Your Own Smartstub Type

By default, all client agents use EJBStub. But if your system requires communication through sockets, you can specify that client agents use the SOCKETSTUB option. You can also create your own smartstub type.

To create your own smartstub type:

1. Develop your own communication service Java class to implement the new protocol (for example, com.xxx.service.IIOPServiceSmartStub) and implement the SmartStubInterface interface from Chordiant.
2. Create a new *{service}*`SmartStub.xml` file in the *{CHORDIANT_ROOT}*`/config/chordiant/components` directory, based on the `SmartStubs.xml` in *{CHORDIANT_ROOT}*`/config/chordiant/components/master` directory.

   {CHORDIANT_ROOT} corresponds to the chordiant.configuration configurationRootDirectory parameter in your application server. Refer to "Configuration Files and the ConfigurationRootDirectory" on page 46 for more information.

3. Within your new *{service}*`SmartStub.xml` file, create a **smartstubs** Section and a **smartstubs.type** Tag for your new smart type (for example, IIOPStub) in the enumeration Section of the file. Be sure to name it after your service, shown here as *{myservice}*.

```
<Section>smartstubs
       <Tag>{myservice}.type
          <Value>IIOPStub</Value>
       </Tag>
</Section>
```

4. Create a new **stub** Section and a **classname** Tag for your new smart type (for example, IIOPStub) for the attributes Section of the *{service}*`SmartStub.xml` file.

```
<Section>IIOPStub
       <Tag>classname
          <Value>com.xxx.service.IIOPSmartStub</Value>
       </Tag>
</Section>
```

5. Create a new *{component}*`.xml` configuration file in the *{CHORDIANT_ROOT}*`/config/chordiant/components` directory, based on the *{service}*`.xml` configuration file for the service the client agent will communicate with. (This file is located in the *{CHORDIANT_ROOT}*`/config/chordiant/components/master` directory.)

6. Create a *{component}***ClientAgent** Section and a **stubtype** Tag.

```
<Section>myClientAgent
…
       <Tag>stubtype
          <Value>IIOPStub</Value>
       </Tag>
</Section>
```

Note: Remember to follow the general rules for working with all configuration files, including not changing any of the `master.xml` or `components/master` files. Refer to "Configuration Files" on page 95 for details on working with configuration files.

Code Sample 8-6 is an example of the EJBStub definition in the `smartstubs.xml` file.

```
<Section>smartstubs
   <Tag>smartstubs.type
      <Value>EJBStub</Value>
   </Tag>
   ...
</Section>
<Section>EJBStub
   <Tag>classname
      <Value>com.chordiant.service.clientagent.EJBSmartStub</Value>
   </Tag>
   <Tag>Definition
      <Value>EJBStubBMT</Value><! You can have multiple definitions
```

Code 8-6: Defining EJBStub in the smartstubs.xml File

```
    </Tag>
    <Tag>Definition
        <Value>EJBStubCMTRequired</Value> <! You can have multiple definitions
    </Tag>
</Section>
<Section>EJBStubBMT
     <Tag>ConnectionName
         <Value>EJBBMT</Value>
     </Tag>
     <Tag>JNDIName    <! The JNDI name is configurable. Note the BMT ending
         <Value>com_chordiant_service_ejb_EJBGatewayServiceBMT</Value>
     </Tag>
</Section>
<Section>EJBStubCMTRequired
    <Tag>ConnectionName
        <Value>EJBCMTRequired</Value>
    </Tag>
    <Tag>JNDIName     <! The JNDI name is configurable. Note the CMTRequired
        <Value>com_chordiant_service_ejb_EJBGatewayServiceCMTRequired</Value>
    </Tag>
    <Tag>NameServiceHostURL <! Optional external JNDI configuration
        <Value>xxx://yyy:zzz</Value> <! for each connection
    </Tag>
    <Tag>InitialContextFactory
        <Value>some.vendor.specific.initial.context.factory</Value>
    </Tag>
</Section>
```

Code 8-6: Defining EJBStub in the smartstubs.xml File (Continued)

Creating Another EJB Deployment

As described in "CMT "trans-attribute" Options" on page 25, Chordiant provides the CMTRequired trans-attribute. If you want to use a different EJB deployment, you can define it yourself.

Note: Chordiant-provided JX services are set up to use either BMT or CMTRequired. Do not change the transaction model or transaction attribute for any existing JX services.

To use a custom EJB deployment:

1. Develop your Java class to implement the new transaction attribute, for example, CMT Mandatory.
2. Create a new *{service}*`SmartStub.xml` file in the *{CHORDIANT_ROOT}*`/config/chordiant/components` directory, based on the `SmartStubs.xml` in *{CHORDIANT_ROOT}*`/config/chordiant/components/master` directory.

3. Create an EJBStub Section and a Definition Tag named for your service for the new transaction attribute. In this example, EJBStubCMTMandatory. Note that EJBStubCMTMandatory is the Section name in Step 4.

```
<Section>EJBStub
       <Tag>[myservice]Definition
          <Value>EJBStubCMTMandatory</Value>
       </Tag>
</Section>
```

4. Create a new Section for the new transaction attribute with ConnectionName and JNDIName Tags.

```
<Section>EJBStubCMTMandatory
   <Tag>ConnectionName
      <Value>EJBCMTMandatory</Value>
   </Tag>
   <Tag>JNDIName
      <Value>com_xxx_service_ejb_EJBGatewayServiceCMTMandatory</Value>
   </Tag>
</Section>
```

   The ConnectionName used here needs to match the ConnectionName used in the service configuration file.

5. Create a new *{component}*`.xml` configuration file in the *{CHORDIANT_ROOT}*`/config/chordiant/components` directory, based on the *{service}*`.xml` configuration file for the service. (This file is located in the *{CHORDIANT_ROOT}*`/config/chordiant/components/master` directory.)

6. Within the service Section of the new *{component}*`.xml` file, update the ConnectionName Tag to match the ConnectionName in *{component}*`SmartStub.xml` file.

```
<Section>xxxService
       …
       <Tag>ConnectionName
          <Value>EJBCMTMandatory
       </Tag>
       …
```

INTEGRATING WITH CHORDIANT SERVICES

Now that you have created your channel-independent enterprise business logic, you can choose from many different styles of integration. Figure 8-10 illustrates four styles of integration.

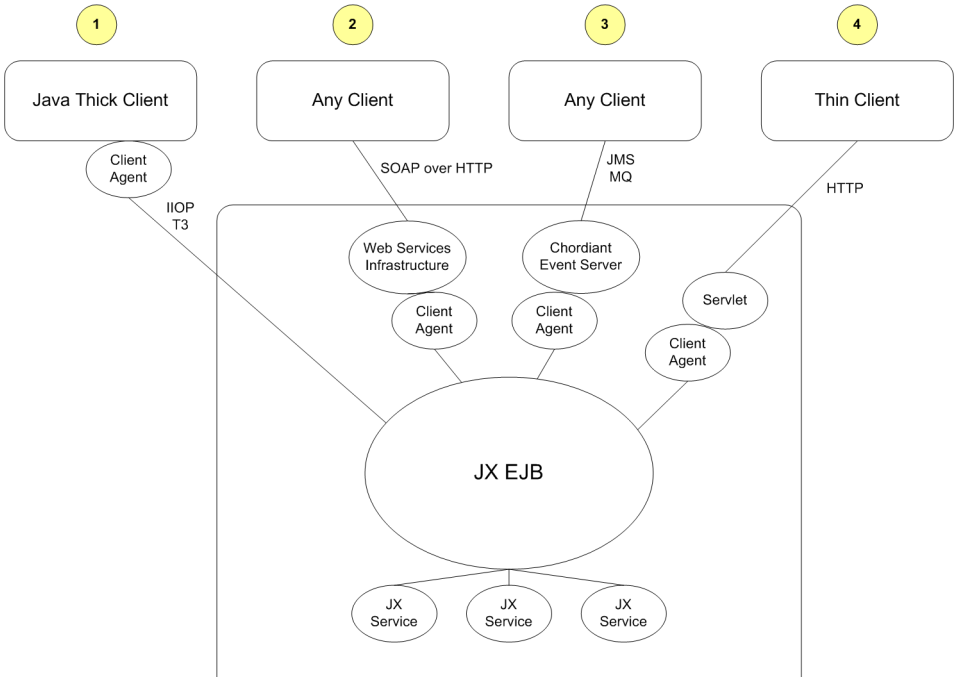


Figure 8-10: Styles of Integrating with Chordiant Services

The four styles of integration illustrated in Figure 8-10 include:

1. Java thick clients can use a Java client agent to contact the JX EJB and its services. For more information on client agents, refer to "Building a Client Agent" on page 134.
2. Any client that can use SOAP over HTTP can use the web services infrastructure to access the JX EJB and its services. For more information on web services, refer to "Using Web Services" on page 133 and the "Web Services" chapters in the *Chordiant 5 Foundation Server Application Components Developer's Guide*.
3. Any client that can interface with JMS or MQ can access JX services asynchronously through the Chordiant Event Server. For more information on interacting with JX services through asynchronous messaging, refer to "Chordiant Event Server" on page 239.

4. Thin clients using HTTP can access servlets which, in turn, contact Chordiant client agents, which are proxies to the JX services. For more information on how thin clients can interact with JX services, refer to "Request Server" on page 283.

All of these styles are valid. You can choose one of these styles based on many factors, including:

- type of client
- deployment
- performance

For example, if you have a Java client, using Java client agents is your best choice, since they are so performant. If you have a C# client, your choices are more limited and your best choice might be to use the web services interface. Consider your needs when determining which style of interaction is best for your enterprise solution.

USING WEB SERVICES

Chordiant offers web services functionality for your enterprise. Through web services, you can share Chordiant functionality with remote or non-Chordiant applications within your enterprise — regardless of the platform. You can also take advantage of non-Chordiant functionality within your Chordiant deployment.

- You can implement any Chordiant JX service as a web service, so it can be securely accessed by other systems over HTTP or other protocols.
- You can use Chordiant JX services to make calls to external web services, outside the Chordiant system.

Many of the Chordiant-provided services are also available as web services, complete with WSDL (Web Service Description Language) files and WSDD (Web Services Deployment Descriptor) files. These web services are ready for you to deploy and share with other users. A list of available web services files is published in the *Chordiant 5 Foundation Server Application Components Developer's Guide.*

If you model your own service or customize a Chordiant service model, after you create your service components, you can choose to generate WSDLs and WSDDs. For details on generating and using WSDLs, WSDDs, and task descriptors, refer to the *Chordiant 5 Foundation Server Application Components Developer's Guide*.

Web Services Security

Chordiant web services are secure because, as you've already read on page 138, the authentication token is embedded in each request to every Chordiant service, rather than just being part of the request container. Each Chordiant service call requires username and authenticationToken parameters. Without these parameters, you cannot access the service. So before you can call any Chordiant web service, you must first call the Chordiant Security Manager web service, specifically the authenticate method, to receive an authentication token. Once that token is acquired, use that token in your subsequent calls to any Chordiant web service. For more information on security, refer to Chapter 11, "Security".

If you are using Secure Sockets Layer (SSL) for web security, web services will still be fully accessible and functional.

BUILDING A CLIENT AGENT

This section describes the process of building a client agent using Chordiant 5 Foundation Server, in the context of a sample service, client agent, and application. The methods that you implement in the client agent correspond to the methods you define in the business service. For more information on building business services, refer to "Building a Service" on page 112.

Tip: It is possible to have one aggregate client agent that calls multiple business services. There does not have to be a one to one correlation between client agents and different business services.

Note: All logic, including caching, is implemented in the business services and server-side business object behaviors, not within the client agents, since it is possible for services and business object classes to be accessed without a client agent. Do not put business logic or state information in client agents when creating client agents.

Client Agent Structure

Before you build a client agent, it is helpful to have an overview of its structure.

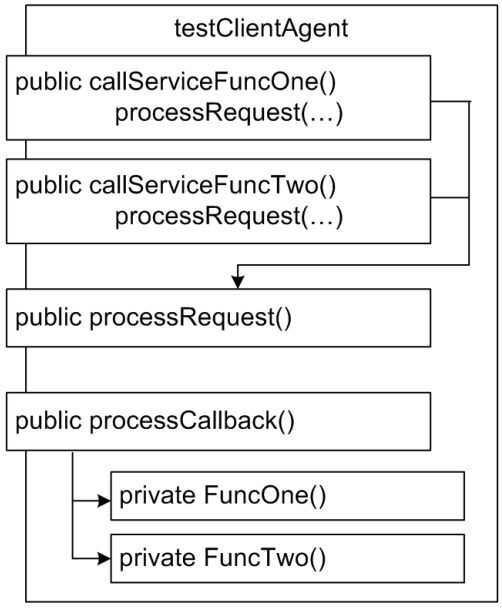


Figure 8-11: Structure of a Client Agent

Client agents consist of these main parts:

- **processRequest method** — Used to dispatch requests to the appropriate service and method on the server. With a typed interface, processRequest may be included in one or more public methods, as shown above.
- **processCallback method** — When a client agent is called from the service, processCallback is used to call methods to be implemented remotely on the *client* side.

Creating a Client Agent

To create a client agent:

1. Create a public class that extends the ClientAgentBaseClass, as shown in Code Sample 8-7.

```
public class TestClientAgent extends ClientAgentBaseClass {
    . . .
}
```

Code 8-7: TestClientAgent Extends the ClientAgentBaseClass

Notes: The example client agents created in this chapter extend the ClientAgentBaseClass. All Chordiant-provided business services extend the BusinessDataClientAgentBaseClass.

The BusinessDataClientAgentBaseClass has extra features used for proper business client agents. You can use the extra features of this base class or use the ClientAgentBaseClass to create your own business or system services.

As shown in Figure 8-12, the base class is fully functional and, as provided, is capable of calling any business service. You can simply instantiate it for typeless interfaces. However, you will likely want to subclass the base class to use a typed interface and make your client agent more targeted.

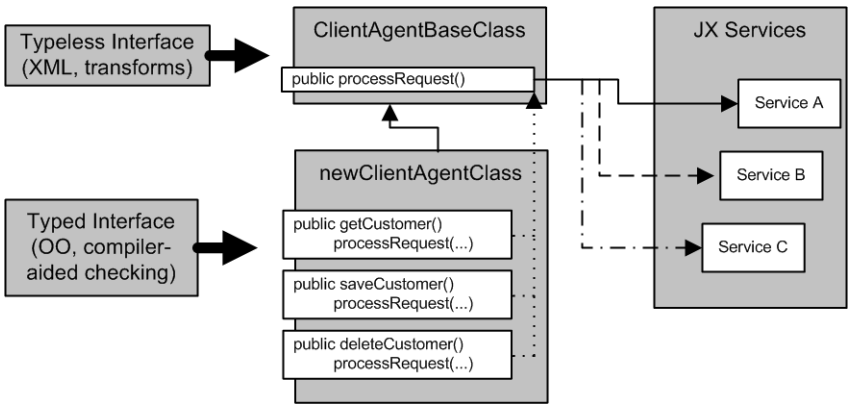


Figure 8-12: ClientAgentBaseClass

2. Define the CLASS_NAME and the PACKAGE_NAME as constants.

   The CLASS_NAME constant is used to hold the value of the client agent name specified in the XML configuration file. Applications use this constant to specify the desired client agent class name when calling the getClientAgent method in the ClientAgentHelper. The package name is commonly used for logging purposes.

   Code Sample 8-8 shows sample constants you might add within the client agent class definition.

```
public final static String CLASS_NAME = "TestClientAgent";
public final static String PACKAGE_NAME = "john.simple.jx.test";
```

Code 8-8: Sample Constants for the Client Agent Class Definition

3. Define the method name constants.

   Applications use these method name constants when calling the processRequest method in the client agent. The processRequest method then uses these method names to route the request to the appropriate service and method.

   If you already defined the method name constants in a separate class when creating the service (refer to Step 3 on page 115), you can use that same class here. If necessary, you can add method name constants to this same class.

   Alternatively, you might add this line within the client agent definition.

```
public final static String FUNCTION_DOWORK = "dowork";
```

4. Implement the methods of the interface to the service.

   The purpose of each of these methods is to provide a typed interface to the client application. The method itself simply passes the request through to the typeless interface of the service.

   In the case of the service defined in "Building a Service" on page 112, you would implement methods for these functions within the service interface:

   — docallback

   — dohelloecho

   — doservice2servicecall

   Here is the implementation of dohelloecho. For the other methods mentioned here, refer to the example code in the `Documentation/Samples/Services` directory on the Installation CD. You can also access this directory through the Chordiant Tools Platform under **Help | Help Contents**.

Note: Refer to "Passing Payload with PayloadData" on page 140 for information on payload.

```
public String dohelloecho(String userName, String authenticationToken, String inputData)
{
    final String METHOD_NAME = "dohelloecho";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    String retval = null;
    PayloadData requestPayload = null;
    PayloadData responsePayload = null;
    Object tmpResponsePayload = null;

    try
    {
        // Set the payload.
        requestPayload = new PayloadData();
        requestPayload.putDataWithName("theParameterName",inputData);

        // Call the remote JX service
        tmpResponsePayload =
            processRequest(
                userName,
                authenticationToken,
                TestService.CLASS_NAME,
                TestService.FUNCTION_DOHELLOECHO,
                requestPayload);

        // Cast the returned payload.
        responsePayload = (PayloadData)
            (tmpResponsePayload);

        // Pull out any needed return values from the payload as appropriate for the
        // return value of this method.
        retval = (String)(responsePayload.getDataWithName("theParameterName"));
    }
```

Code 8-9: Implementing the Service Methods

```
    catch (Throwable e)
    {
        LogHelper.error(
            PACKAGE_NAME,
            CLASS_NAME,
            METHOD_NAME,
            "Exception occurred",
            e);
    }

    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retval;
}
```

Code 8-9: Implementing the Service Methods (Continued)

5. Implement the dowork method, including using the processRequest method.

   The processRequest method serves as the single entry point for the service. Use the processRequest method within your client agent methods (such as dowork, getCustomer, or any method) to call the corresponding service.

   The processRequest method takes these five arguments:

   - **username**: The name of the user.
   - **authenticationToken**: From the Security service.
   - **serviceName**: The name of the service class. Should be the same as that specified within the XML configuration files.
   - **functionName**: The function within the specified service.
   - **requestPayload**: For requestPayload, you can use the PayloadData container class. Refer to "Passing Payload with PayloadData" on page 140 for details.

   There are additional forms for processRequest. Refer to "Additional Forms of processRequest" on page 139.

   Code Sample 8-10 shows an example segment within the dowork method, which uses processRequest.

```
public String dowork(String userName, String authentication, String inputData)
{
    final String METHOD_NAME = "dowork";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    String retval = null;
    PayloadData requestPayload = null;
    PayloadData responsePayload = null;
    Object tmpResponsePayload = null;

    try
    {
        // Set the payload.
        requestPayload = new PayloadData();
        requestPayload.putDataWithName("theParameterName",inputData);

        // Call the remote clientagent.
        tmpResponsePayload = processRequest(
            userName,
```

Code 8-10: doWork Method including processRequest

```
            authentication,
            TestClientAgent.CLASS_NAME,
            TestClientAgent.FUNCTION_DOWORK,
            requestPayload);

        // Cast the returned payload.
        responsePayload = (PayloadData)
            (tmpResponsePayload);

        // Pull out any needed return values from
        // the payload as appropriate for the return
        // value of this method.
        retval = (String)(responsePayload.
            getDataWithName("theParameterName"));
    }
    catch (Throwable e)
    {
        LogHelper.error(
            PACKAGE_NAME,
            CLASS_NAME,
            METHOD_NAME,
            "Exception occurred",
            e);
    }

    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retval;
}
```

Code 8-10: doWork Method including processRequest (Continued)

6. Optionally, specify any shutdown tasks you might want to add to the client agent's functionality. Make modifications to the shutdown method.

7. If you are using a callback, implement the processCallback method to enable the client agent to receive a callback from a service. The public processCallback method dispatches requests to private functions within the client agent. (See Figure 8-11 on page 134.)

   The private methods are meant to be implemented on the client side when called from the server side.

   This step is only needed if the client agent will be receiving callbacks.

8. Register the client agent class name into an XML configuration file for instantiation. Refer to "Adding Components through Configuration" on page 101 for instructions.

Additional Forms of processRequest

The ClientAgentBaseClass includes additional signatures for processRequest. The content is the same as in the original processRequest format (see Step 5 on page 138), however the form is more streamlined. We encourage you to use these more streamlined signatures in any new customizations.

Object processRequestObject (Object inputData)
or
String processRequestXMLString(String inputdata)

The Object or String must include the PayloadData along with the required username, authenticationToken, serviceName, and functionName parameters or tags. Public static constant definitions for the required parameter names (or XML tags) can be found in com.chordiant.service.constants.ServiceConstants, they are:

- PAYLOAD_DATA_USERNAME_TAG
- PAYLOAD_DATA_AUTHEN_TAG
- PAYLOAD_DATA_SERVICENAME_TAG
- PAYLOAD_DATA_FUNCTIONNAME_TAG

Note: Check that the tags you are using will not clash with these tag values.

For the XML version of processRequest, the XML that you use for the String input and output in this interface is the same format as that described in "XML Client Agent Interface" on page 143.

Passing Payload with PayloadData

The payload is how data is passed between a client agent and a business service. Use PayloadData to pass data instead of creating your own classes or vectors. PayloadData is in the package com.chordiant.service.

The class PayloadData is an intermediary container for data requests between client agents and business services. The PayloadData container must be the top-level object passed for all implemented processRequest methods.

Note: processRequest requires that payload data be at the top of the object graph or, for XML, payload data must be at the root level.

Here are the key methods of PayloadData:

- public object **getDataWithName** (String key)—gets the data using the associated String
- public void **putDataWithName** (String key, Object value)—sets the data property value for the named parameter

Note: In general, keys should be unique for any set of data. However, it is possible to use a key more than once. For example, if you use putDataWithName twice using the same key, like "LastName", the last object you enter is the one that will be used.

Supported Data Types

These data types are supported for communication between client agents and services.

- Java primitive types
- Java simple types
- Java arrays
- java.util.Vector
- java.util.Hashtable
- java.util.HashMap
- java.util.HashSet
- java.util.LinkedList
- java.util.ArrayList
- java.util.TreeMap
- org.jdom.Document
- org.jdom.element
- org.w3c.dom.Document

JAVA TYPE	XSD TYPE	JAVA TYPE	XSD TYPE
boolean	boolean	float	float
byte	byte	double	double
char	unsignedShort	java.lang.String	String
short	short	java.util.Date	dateTime
int	int	electric.util.Hex	hexBinary
long	long	java.math.BigDecimal	decimal

Table 8-1: Supported Java and XSD Types

All data types listed above, except electric.util.Hex, are serializable.

All data containers including these data types, for example business objects, should implement serializable and clonable.

These are communication data types, not persistence data type mappings. For information on persistence data type mapping, see "Mapping Java Data Types to Database Types" on page 225.

Additional Types of Client Agents

The Chordiant 5 Foundation Server client agents described so far, are Java-based and object-oriented. If your application is also Java-based and object-oriented, then these "standard" Chordiant 5 client agents are probably the best choice to use with your application.

The JX architecture is flexible, enabling you to use alternative client agent styles for different types of applications. For example, if you are using an XML-based application, you will probably want to use the XML client agent instead of the standard JX client agents used for Java-based, object-oriented applications. For other types of applications, you can use Chordiant's web services, as described in "Using Web Services" on page 133.

Note: The JX SOAP servlet is deprecated in this release and will be removed from the product in future releases. If your application uses SOAP over HTTP, use web services instead. For more information, refer to "Using Web Services" on page 133.

The client agent you choose depends on the "client" application architecture you are using. Regardless of client agent you use, the JX infrastructure will provide a seamless interface to the appropriate JX service to get the necessary work done.

You can even choose not to use client agents at all, but to rely on direct J2EE to access JX services through the JX EJB. Refer to "Accessing Services without Client Agents" on page 149 for details.

Requirements for Interaction with JX Services through XML

There are several formatting requirements for the requests and responses for interacting with a JX service through XML messages.

1. The XML formatting rules for all request and response XML data conform to the W3C XML Schema and the W3C SOAP Encoding Style specifications.

   The related specifications are:

   http://www.w3.org/TR

   - XML Schema Part 0: Primer
   - XML Schema Part 1: Structures
   - XML Schema Part 2: Datatypes
   - SOAP Version 1.2 Part 0: Primer
   - SOAP Version 1.2 Part 1: Messaging Framework
   - SOAP Version 1.2 Part 2: Adjuncts

2. The top level http://www.w3.org/TR for all requests and responses is of type http://www.w3.org/TR.
3. The http://www.w3.org/TR type is a top level container http://www.w3.org/TR which can contain one or more http://www.w3.org/TR nodes.

4. The ParameterPair type is a container node which contains name and value nodes for a single parameter.
5. All JX service requests are required to contain the following ParameterPairs:
   - userName
   - authenticationToken
   - serviceName
   - functionName
   - zero or more arbitrary parameters, based on the specific service or function that is called.
6. All JX service responses are required to contain the following ParameterPairs:
   - zero or more arbitrary parameters, based on the specific service or function that was called.

XML Client Agent

If you are working with an XML-oriented architecture, you can choose to use the XML Client Agent, included with Chordiant 5 Foundation Server. If you use this client agent, you can interface with any JX service without using Java objects at all. (See note below.) Services behave the same way regardless of whether they are accessed by the XML Client Agent or by an object-oriented client agent (described in the rest of this Client Agent section).

Note: The XML Client Agent works with all provided Foundation Server application components (JX services). If you want to build a custom JX service which participates with the XML Client Agent, the service must receive and return PayloadData. (See "Passing Payload with PayloadData" on page 140.)

XML Client Agent Interface

The XML Client Agent is basically another implementation of the processRequest method. Rather than taking the standard five parameters for processRequest (user name, authentication token, service name, method name, and payload), this client agent takes an XML document, which then must contain all five pieces of information just listed for processRequest.

The XML Client Agent is in the package com.chordiant.service.clientagent.xml.XMLClientAgent.

There are three interfaces for the XML Client Agent:

- org.w3c.dom.Document processRequest(org.w3c.dom.Document)
- org.jdom.Document processRequest(org.jdom.Document)
- String processRequest(String)

To implement the XML Client Agent, you must use the SOAP-compliant form for both request and response, as shown in Code Sample 8-11 and Code Sample 8-12 on page 145. These examples are for a specific use—the getparty method of the party service. In your implementations, you will use the same form, but will specify different methods, services, and parameter data.

Example Input XML

```
<?xml version='1.0' encoding='UTF-8'?>
<root xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xmlns:xsd='http://www.w3.org/2001/XMLSchema'
xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'>
  <payload id='id0' xmlns:ns1=
    'http://www.themindelectric.com/package/com.chordiant.service/'
    xsi:type='ns1:PayloadData'>
    <fieldData id='id1' xmlns:ns1='http://www.themindelectric.com/collections/'
         xsi:type='ns1:vector'>
      <item id='id2'
    xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
    xsi:type='ns1:ParameterPair'>
        <fieldName xsi:type='xsd:string'>userName</fieldName>
        <fieldData xsi:type='xsd:string'>hmonroe</fieldData>
      </item>
      <item id='id3'
     xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
    xsi:type='ns1:ParameterPair'>
        <fieldName xsi:type='xsd:string'>authenticationToken</fieldName>
        <fieldData xsi:type='xsd:string'>***</fieldData>
      </item>
      <item id='id4'
    xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
    xsi:type='ns1:ParameterPair'>
        <fieldName xsi:type='xsd:string'>serviceName</fieldName>
        <fieldData xsi:type='xsd:string'>PartyService</fieldData>
      </item>
      <item id='id5'
    xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
    xsi:type='ns1:ParameterPair'>
        <fieldName xsi:type='xsd:string'>functionName</fieldName>
        <fieldData xsi:type='xsd:string'>getparty</fieldData>
      </item>
      <item id='id6'
    xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
    xsi:type='ns1:ParameterPair'>
        <fieldName xsi:type='xsd:string'>aParty</fieldName>
        <fieldData id='id7'
        xmlns:ns1='http://www.themindelectric.com/package/
        com.chordiant.businessServices.partyBusinessClasses/'
        xsi:type='ns1:Party'>
          <PartyNumber xsi:type='xsd:string'>536477</PartyNumber>
        </fieldData>
      </item>
    </fieldData>
  </payload>
</root>
```

Code 8-11: Example Input XML for XML Client Agent

Example Output XML

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <payload id="id0" xmlns:ns1="http://www.themindelectric.com/
    package/com.chordiant.service/" xsi:type="ns1:PayloadData">
    <fieldData id="id1" xmlns:ns1="http://www.themindelectric.com/collections/"
    xsi:type="ns1:vector">
      <item id="id2"
    xmlns:ns1="http://www.themindelectric.com/package/com.chordiant.service/"
    xsi:type="ns1:ParameterPair">
        <fieldName xsi:type="xsd:string">aParty</fieldName>
        <fieldData id="id3"
        xmlns:ns1="http://www.themindelectric.com/package
        /com.chordiant.businessServices.partyBusinessClasses/"
        xsi:type="ns1:Party">
          <PartyRoleTypeCode
        xsi:type="xsd:string">CUSTOMER</PartyRoleTypeCode>
          <PartyTypeCode xsi:type="xsd:string">PERSON</PartyTypeCode>
          <GovernmentIdNumber
        xsi:type="xsd:string">432-45-9834</GovernmentIdNumber>
          <RelatedPartyId
        xsi:type="xsd:string">0000000001</RelatedPartyId>
          <CreateDate id="id4"
        xsi:type="xsd:dateTime">1995-12-06T08:00:00Z</CreateDate>
          <PartyNumber xsi:type="xsd:string">536477</PartyNumber>
          <ChallengeData xsi:type="xsd:string">Fleming</ChallengeData>
          <PassCode xsi:type="xsd:string">840391</PassCode>
          <Id xsi:type="xsd:string">-2147482299</Id>
        </fieldData>
      </item>
    </fieldData>
  </payload>
</root>
```

Code 8-12: Example Output XML for the XML Client Agent

Using the XML Client Agent

If you want to call two different services, call the XML Client Agent twice, each time with a different XML document which provides the parameters needed to communicate with the desired service and method. The XML Client Agent is another implementation of processRequest which just takes different parameters.

Calling the XML Client Agent is similar to calling any other client agent, as shown in Code Sample 8-13.

```
...
//Import the XML Client Agent
    import com.chordiant.service.clientagent.xml.XMLClientAgent;
...

    private static String userName = "hmonroe";
    private static String userPassword = "***";
    private static String authenticationToken = null;
    XMLClientAgent xmlClientAgent = null;
    org.w3c.dom.Document theInputXMLDocument = null;
```

Code 8-13: Calling the XML Client Agent

```
org.w3c.dom.Document theOutputXMLDocument = null;

FatClientStaticHelper.serviceControl(
    StaticHelperBaseClass.SERVICE_CONTROL_COMMAND_SETUP);

authenticationToken = SecurityManager.authenticate(
    userName, userPassword);

// Get the XMLClientAgent
xmlClientAgent =
    (XMLClientAgent) ClientAgentHelper.getClientAgent(
    XMLClientAgent.CLASS_NAME);

// Fill theInputXMLDocument with appropriate XML.

// Call the XMLClientAgent and receive an XML Document response.
theOutputXMLDocument = xmlClientAgent.processRequest(
    theInputXMLDocument);


if (theOutputXMLDocument != null)
{
    // XMLClientAgent call successful, return XML data
    // in theOutputXMLDocument.
}
else
{
    System.out.println("**** FAILURE on xml client agent request");
}
```

Code 8-13: Calling the XML Client Agent (Continued)

Additional examples of XML request and response instances that can be used with the XML client agent are available in the online `Documentation/Samples/JX_XML` directory on the Installation CD. You can also access this directory through the Chordiant Tools Platform under **Help** | **Help Contents**.

Accessing Services through Messaging

The XML Client Agent is also used to access JX services through Java Messaging Service (JMS). For information on this topic, refer to Chapter 10, "Chordiant Event Server", beginning on page 239.

Generic SOAP Servlet

Note: This servlet is deprecated in this release. It will be removed from future releases. We suggest you use web services and SOAP instead of using this servlet. For detailed information on Web Services, refer to the *Chordiant 5 Foundation Server Application Components Developer's Guide*.

You can use SOAP encoding over HTTP to access JX services. Any application that can communicate with SOAP can interact with the generic JX SOAP servlet included in the JX infrastructure. The generic JX SOAP servlet then communicates with the target JX service through an XML Client Agent on the caller's behalf.

The SOAP servlet[1] is installed with Chordiant Foundation Server as a web application and is automatically initialized. The name of the servlet is com.chordiant.application.SOAPServlet.

To use the generic SOAP servlet that is part of the Chordiant Foundation Server, the HTTP client makes an HTTP post request, with an embedded valid SOAP request, to this URL:

http://localhost/soap/SOAPServlet

For testing purposes, you can type or paste a valid SOAP request into this test page URL:

http://localhost/soap/presentations/html/SOAPForm.htm

Note: Replace localhost with your URL host information.

In the example on page 148, notice that the main information of the request is enclosed in the standard <SOAP-ENV:Body> tags. This information is the same as the XML Client Agent request example above. The generic JX SOAP servlet simply strips away the SOAP wrapping and essentially delivers the remaining XML through an XML Client Agent to the target service.

The examples starting on page 149 show the SOAP wrappers, without specific code.

1. The SOAP servlet is deprecated in this release. It will be removed from future releases. We suggest you use web services and SOAP instead of using this servlet.

Example SOAP Request to a JX Service[1]

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
<root xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
     xmlns:xsd='http://www.w3.org/2001/XMLSchema'
     xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'>
  <payload id='id0'
     xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
     xsi:type='ns1:PayloadData'>
    <fieldData id='id1' xmlns:ns1='http://www.themindelectric.com/collections/'
         xsi:type='ns1:vector'>
      <item id='id2'
     xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
     xsi:type='ns1:ParameterPair'>
        <fieldName xsi:type='xsd:string'>userName</fieldName>
        <fieldData xsi:type='xsd:string'>john</fieldData>
      </item>
      <item id='id3'
     xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
     xsi:type='ns1:ParameterPair'>
        <fieldName xsi:type='xsd:string'>authenticationToken</fieldName>
        <fieldData xsi:type='xsd:string'>ccs</fieldData>
      </item>
      <item id='id4'
     xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
     xsi:type='ns1:ParameterPair'>
        <fieldName xsi:type='xsd:string'>serviceName</fieldName>
        <fieldData xsi:type='xsd:string'>PartyService</fieldData>
      </item>
      <item id='id5'
     xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
     xsi:type='ns1:ParameterPair'>
        <fieldName xsi:type='xsd:string'>functionName</fieldName>
        <fieldData xsi:type='xsd:string'>getParty</fieldData>
      </item>
      <item id='id6'
     xmlns:ns1='http://www.themindelectric.com/package/com.chordiant. service/'
     xsi:type='ns1:ParameterPair'>
        <fieldName xsi:type='xsd:string'>aParty</fieldName>
        <fieldData id='id7'
          xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.
          businessServices.partyBusinessClasses/' xsi:type='ns1:Party'>
          <PartyNumber xsi:type='xsd:string'>536477</PartyNumber>
        </fieldData>
      </item>
    </fieldData>
  </payload>
</root>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

1. The SOAP servlet is deprecated in this release. It will be removed from future releases. We suggest you use web services and SOAP instead of using this servlet.

SOAP Request Template[1]

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        *** Chordiant "payload" request subtree goes here ***
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Response Template

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    <SOAP-ENV:Body>
        *** Chordiant "payload" response subtree goes here ***
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Response Fault Template

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
        <SOAP-ENV:Fault>
            <faultcode>SOAP-ENV:Server</faultcode>
            <faultstring>Server Error</faultstring>
            *** Chordiant "payload" response subtree goes here ***
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Accessing Services without Client Agents

You are not required to use a standard client agent to communicate with a Chordiant service. This section describes the various alternative methods of communication.

Using J2EE to Call the Foundation Server EJB

You can also communicate with any JX service through the Foundation Server EJB without using a client agent. Although client agents provide many benefits, depending on your system, you might not want or be able to use them. You can use J2EE, separate from the JX architecture, to reach JX services through the Foundation Server EJB.

1. The SOAP servlet is deprecated in this release. It will be removed from future releases. We suggest you use web services and SOAP instead of using this servlet.

Code Sample 8-14 is a sample of this J2EE call to the Foundation Server EJB.

```
String ejbJNDIName = "EJBGatewayServiceHome";
com.chordiant.service.ejb.EJBGatewayServiceHome homeInterface = null;
com.chordiant.service.ejb.EJBGatewayService theBean = null;
String someXMLString = null;

// Get the J2EE initial context.
javax.naming.InitialContext initialContext =
    new javax.naming.InitialContext();

// Get the J2EE lookup object for the JX EJB.
java.lang.Object lookupObject = initialContext.lookup(
    ejbJNDIName);

// Do a J2EE narrow on the J2EE lookup object.
homeInterface =
    (com.chordiant.service.ejb.EJBGatewayServiceHome)
    javax.rmi.PortableRemoteObject.narrow(
        lookupObject,
        com.chordiant.service.ejb.EJBGatewayServiceHome.class);

// Do a J2EE create on the J2EE home interface to get an actual
// reference to the JX EJB.
theBean = homeInterface.create();

// Call the JX EJB directly.
String retval = theBean.processRequestXMLString(
    someXMLString);
```

Code 8-14: J2EE Call to the Foundation Server EJB

To access any JX service without a client agent, you can use either of the following JX EJB single APIs:

String processRequestXMLString(String inputdata)
or
Object processRequestObject (Object inputData)

The Object or String must include the PayloadData along with the required username, authenticationToken, serviceName, and functionName parameters or tags. Public static constant definitions for the required parameter names (or XML tags) can be found in com.chordiant.service.constants.ServiceConstants:

- PAYLOAD_DATA_USERNAME_TAG
- PAYLOAD_DATA_AUTHEN_TAG
- PAYLOAD_DATA_SERVICENAME_TAG
- PAYLOAD_DATA_FUNCTIONNAME_TAG

For the XML version of processRequest, the XML that you use for the String input and output in this interface is the same format as that described in "XML Client Agent Interface" on page 143.

Note: processRequest requires that payload data be at the top of the object graph or, for XML, payload data must be at the root level.

Using the Foundation Server SocketGatewayService

Note: Only use the SocketGatewayService if your application cannot participate in IIOP or in the JX Architecture.

The Foundation Server SocketGatewayService is a custom object that allows access to all JX services through simple socket communications, with a user name and encrypted authentication token. (For information on Custom Objects, refer to "CustomObjects and the CustomObjectHelper" on page 176.) It is a general purpose socket server that enables non-Java applications, or those that cannot use IIOP, to interact with JX services.

In addition to interfacing with the JX services, SocketGatewayService provides a simple socket interface to:

- the application server's JNDI component, including lookup, bind, rebind, and unbind functions
- the ConfigurationHelper's getConfiguration functionality
- the ServiceControl functionality through the Administrative Console (setup, refresh, status, shutdown)

The SocketGatewayService runs as a singleton custom object in each JVM replicate of the application server. Note that these are the JVM replicates that run the JX EJBs and have connectivity to data stores and backend systems.

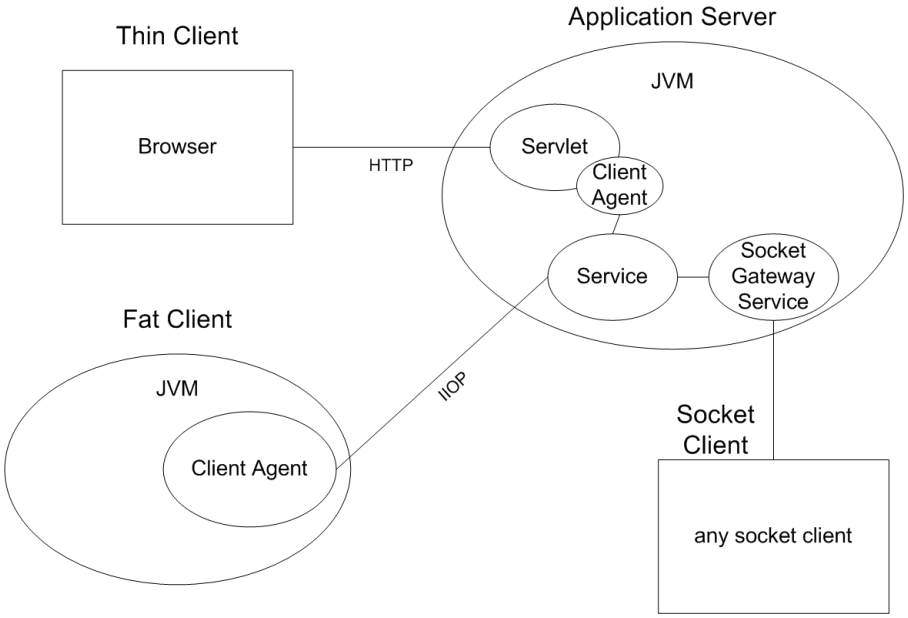


Figure 8-13: Interaction of Socket Client with Chordiant Foundation Server through the SocketGatewayService

Control of the listen IP address and port number for the SocketGatewayService is through the following Java system properties, which must be set on the application server JVM.

The following server-side settings are standard Java system properties that can be set through the Java command line using "-D" parameters.

SocketGatewayService Listen Port Number

Property Name: chordiant.service.socketGatewayServicePort

Command Line Property: -Dchordiant.service.socketGatewayServicePort=ppp

Default Value: There is no default value for this property.

SocketGatewayService Listen IP Address

Property Name: chordiant.service.socketGatewayServiceIPAddress

Command Line Property: -Dchordiant.service.socketGatewayServiceIPAddress=iii

Default Value: If this property is not set, then a default value of localhost is used.

For more information on the SocketGatewayService and the Foundation Server Administrative Console, refer to "Multiple Application Server JVMs and SocketGatewayService" on page 85.

Note: As a custom object, the SocketGatewayService can be disabled through configuration. However, if you disable it, you will not be able to use the Administrative Console (which, itself is a socket client application). Even if you do not plan to use the SocketGatewayService, we suggest that you leave it enabled so the Administrative Console is still functional.

To use the SocketGatewayService:

1. Open a socket on the port number of the SocketGatewayService.
2. Write a request using this format:

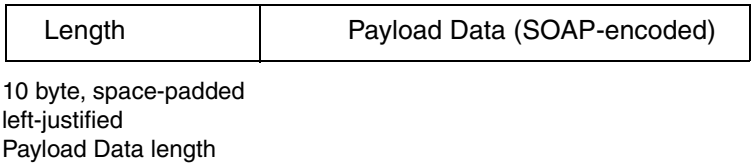


Figure 8-14: Length-Encoded SOAP Protocol for SocketGatewayService

   For information on the SOAP-encoded Payload Data, refer to "Requirements for Interaction with JX Services through XML" on page 142.

3. You will receive a response in the same format as your request.

Exceptions with Socket Protocol

For information on exceptions with socket protocol, refer to "Socket Protocol Exceptions" on page 93.

Security and the SocketGatewayService

Your company's Information Technology (IT) or Management Information Services (MIS) department protects your internal network so your back-end data stores and legacy systems are secure and only accessible to trusted clients on your network.

The JVMs which contain your EJBs are on a physical network or subnetwork that enables network connectivity to your enterprise data stores and legacy systems. The SocketGatewayService resides on the same JVMs where the EJBs reside. By definition, this is within your company's secure network. The only nodes or applications which would be able to access the SocketGatewayService through TCP/IP are those with network connectivity to the EJB JVMs within this secure environment. Anyone outside the network, even if they determine the host name and port number published by the SocketGatewayService, they would not be able to access the SocketGatewayService and its functionality because it is on the secure network.

As shown in Figure 8-15, only clients behind the firewall (point B) can access the SocketGatewayService. Individual HTTP clients (point A) do not have access to backend data, based on requirements imposed by your IT or MIS department.

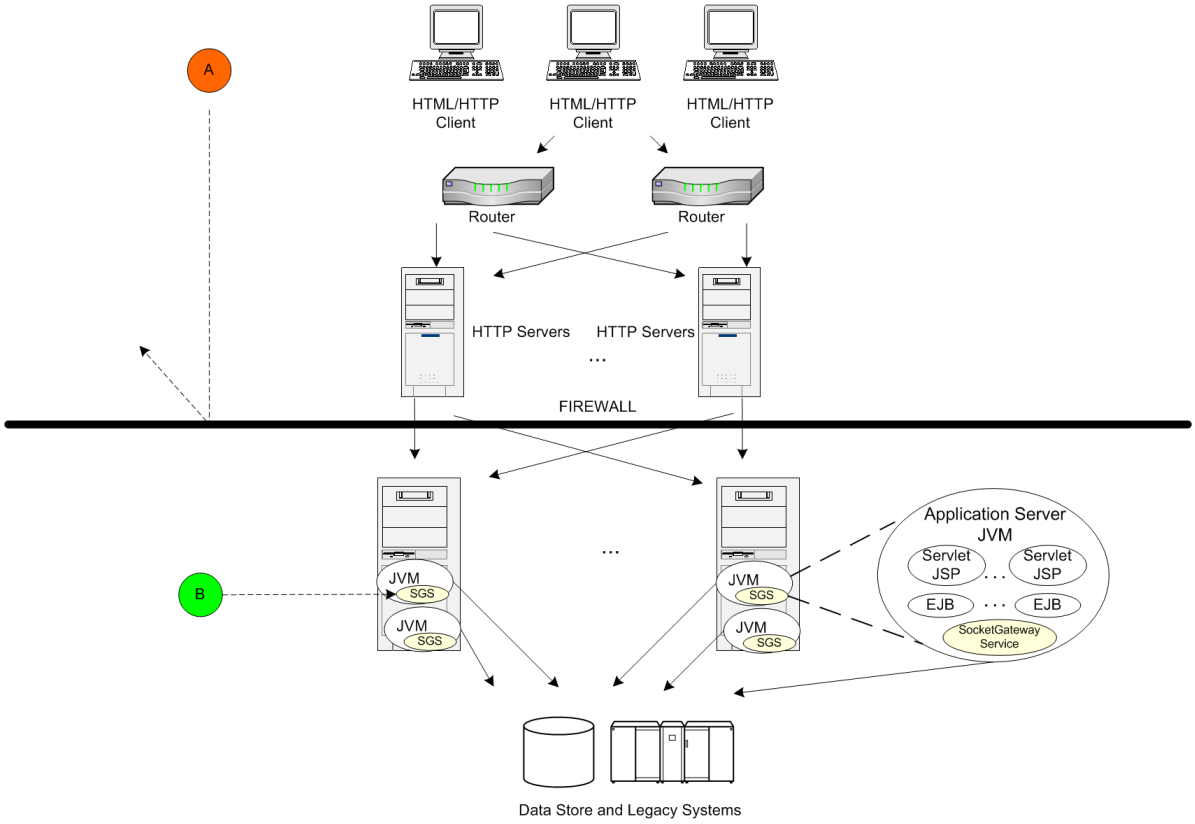


Figure 8-15: Security in a Typical Production Deployment

Configuring the Gateway Service

Currently, the system supports one gateway service mode at a time—either sockets or RMI:

- SocketGatewayService—Required for use with thin clients. May also be used for exclusively Java clients.
- RMIGatewayService—Can be used for exclusively Java clients. Cannot be used with thin clients.

You configure the gateway service within the `GatewayServices.xml` configuration file. The GatewayServices.name tag corresponds to a section which specifies the mode you are using.

Code Sample 8-15 shows a sample `GatewayServices.xml` file with the RMIGatewayService value commented out. If you want to use the RMIGatewayService, uncomment that value and comment out the SocketGatewayService value instead. Once you have checked to make sure the other configuration values work with your system, this is the only change you need to make to switch the type of gateway service you will use.

```
<Root>
   <Section>gatewayservices
      <Tag>GatewayServices.name
         <Value>SocketGatewayService</Value>
         <!-- <Value>RMIGatewayService</Value> -->
      </Tag>
   </Section>
   <Section>SocketGatewayService
      <Tag>classname
         <Value>com.chordiant.service.socket.gateway.SocketGatewayService</Value>
      </Tag>
      <Tag>protocol
         <Value>na</Value>
      </Tag>
   </Section>
   <Section>RMIGatewayService
      <Tag>classname
         <Value>com.chordiant.service.RMIGatewayService</Value>
      </Tag>
      <Tag>protocol
         <Value>RMI</Value>
      </Tag>
   </Section>
   <Section>RMI
      <Tag>classname
         <Value>com.chordiant.service.RMISmartSkeleton</Value>
      </Tag>
      <Tag>connectioninformation
         <Value>none</Value>
      </Tag>
   </Section>
</Root>
```

Code 8-15: Sample GatewayServices.XML File

processRequest Method: Client Agent vs. Service

Figure 8-16 illustrates the processRequest method on the server side and being used in the client agent.

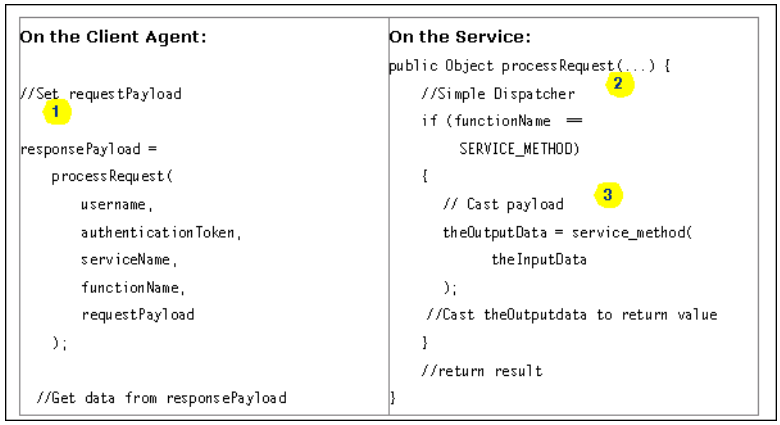


Figure 8-16: processRequest Method

1. On the client agent, a typed interface might surround the processRequest method. The request, consisting of five parameters, is passed to the processRequest method on the service.
2. The processRequest method on the service acts as a dispatch method to route the request to the appropriate service and function.
3. The PayloadData Object is passed as a parameter to processRequest as a Java Object. For the service to use it as PayloadData in the processRequest dispatch method, it must be cast to PayloadData.

Note: The processRequest method is overloaded. For more details on processRequest, see "Additional Forms of processRequest" on page 139.

ClientAgentHelper

The ClientAgentHelper vends client agents to applications when the application requests a client agent. For details on the ClientAgentHelper, refer to "ClientAgentHelper" on page 61.

BUILDING THE CLIENT APPLICATION

This section describes how to build a client application using Chordiant 5 Foundation Server. The client application uses client agents to interact with services running on the application server.

Note: This is an example of a test application. It is not a typical example of a Chordiant client application, but is a simplistic example to highlight the major points of this process.

To build a client application:

1. Create a public class for the test application, as shown in Code Sample 8-16.

```
public class TestApplication {
    . . .
}
```

Code 8-16: Test Application

2. Define the CLASS_NAME and the PACKAGE_NAME as constants.

   Code Sample 8-17 shows sample constants you might add within the test application class definition.

```
public final static String CLASS_NAME = "TestApplication";
public final static String PACKAGE_NAME = "john.simple.jx.test";
```

Code 8-17: Sample Constants for the Test Application Class Definition

Note: You can choose to define constants within a separate class.

3. Define the main method.

```
public static void main(String[] args) {
    . . .
}
```

4. Add code to set up the static helpers.

   You can use the FatClientStaticHelper.serviceControl method.

```
FatClientStaticHelper.serviceControl ( StaticHelperBaseClass. SERVICE_CONTROL_COMMAND_SETUP );
```

5. Add code to log in to the Chordiant 5 Foundation Server system.

   Use the SecurityMgrBeanClientAgent.authenticate method, as shown in Code Sample 8-18.

```
SecurityMgrBeanClientAgent client = (SecurityMgrClientAgent)ClientAgentHelper.getClientAgent(
    SecurityMgrClientAgent.CLASS_NAME);

authenticationToken = client.authenticate( userName, userPassword);
```

Code 8-18: SecurityMgrBeanClientAgent's authenticate Method

6. Add code to enable Network Presence, if applicable.

   Use the GatewayHelper.enableNetworkPresence method.

   ```
   networkPresenceKey = GatewayHelper.enableNetworkPresence(userName, authenticationToken);
   ```

7. Add code to get a client agent.

   Use the ClientAgentHelper.getClientAgent method.

   ```
   testClientAgent = (TestClientAgent) ClientAgentHelper.getClientAgent(TestClientAgent.CLASS_NAME);
   ```

8. Call the service methods defined in the client agent.

   ```
   response = testClientAgent.dohelloecho( userName, authenticationToken, payload );
   ```

9. Add code to disable Network Presence for a clean shutdown.

   Use the GatewayHelper.disableNetworkPresence method.

   ```
   GatewayHelper.disableNetworkPresence();
   ```

Tip: Only disable Network Presence if you enabled it in Step 6 on page 158.

10. Add code to shutdown static helpers for a clean shutdown.

    Use the FatClientStaticHelper.servicecontrol method.

    ```
    FatClientStaticHelper.serviceControl(StaticHelperBaseClass.SERVICE_CONTROL_COMMAND_SHUTDOWN);
    ```

IMPLEMENTING A CALLBACK

This section describes how to implement a callback using Chordiant 5 Foundation Server, in the context of a sample service and application. You implement callbacks to enable the service to request the client application to perform useful work. A callback is similar to a processRequest method in reverse.

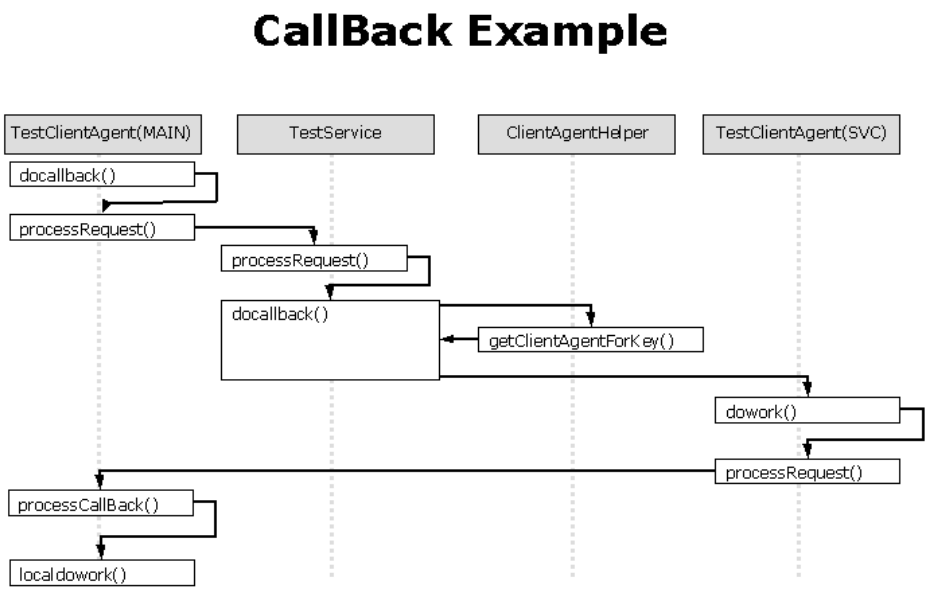


Figure 8-17: Flow of CallBack

You can choose to use callbacks in these scenarios:

- A user is awaiting the availability of a resource. When that resource becomes available, the service performs a callback to the client agent, which, in turn, will notify the user.
- A supervisor might want to know when VIP customer is created or calls in. Callbacks in the Customer service can notify the supervisor.
- Push-based process flow — starting a process from the server side.

To implement a callback method:

1. Implement a method within the service to perform the callback.

   The method in the service issuing the callback uses the ClientAgentHelper with the getClientAgentForKey, specifying the client agent's network presence key, to get the specific client agent for the callback. Once determined, the service can issue a call to the method within the client agent, along with the callback data.

   The network presence key can be passed as a parameter, as shown in this example. You could also create your own service to manage clients' network presence keys for use in callbacks.

   Code Sample 8-19 shows a sample method you can create with a service to perform a callback.

```
private String docallback(String inputData)
{
    final String METHOD_NAME = "docallback";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    String retval = null;
    String callbackMessage = null;
    String callbackResponse = null;
    TestClientAgent testClientAgent = null;

    System.out.println("TestService.docallback() received: ["+inputData+"]");
    try
    {
        // This service function does a callback to
        // a specific client agent using the value
        // of the inputData as the networkPresenceKey.
        testClientAgent = (TestClientAgent)
            ClientAgentHelper.getClientAgentForKey(
            TestClientAgent.CLASS_NAME,inputData);
        callbackMessage = "A callback message from TestService.docallback()";
        System.out.println("TestService.docallback() sending: ["+callbackMessage+"]");
        callbackResponse = testClientAgent.dowork(
            getServiceLoginName(),
            getServiceAuthentication(),
            callbackMessage);
        System.out.println("TestService.docallback() received: ["+callbackResponse+"]");
    }
    catch (Throwable e)
    {
        LogHelper.error(
            PACKAGE_NAME,
            CLASS_NAME,
            METHOD_NAME,
            "Exception occurred",
            e);
    }
```

Code 8-19: Sample Callback Method

```
    retval = "I called you back from TestService.docallback()";
    System.out.println("TestService.docallback() returning: ["+retval+"]");

    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retval;
}
```

Code 8-19: Sample Callback Method (Continued)

Note: This example does not illustrate a typical callback—it was created to show a callback while keeping the code example short.

Usually callbacks are initiated from an outside stimulus, such as an email, which is not related to the client.

2. Within the client agent, implement the callback method that you called from the service in the previous step.

   The method calls the client agent on the client side. Typically, you can have this method within the client agent do nothing more than provide a typed interface that simply passes the request through to the typeless interface of the client agent using the processRequest call.

   Code Sample 8-20 shows a sample method within a client agent to handle the callback from the service.

```
public String dowork(String userName, String authentication, String inputData)
{
    final String METHOD_NAME = "dowork";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    String retval = null;
    PayloadData requestPayload = null;
    PayloadData responsePayload = null;
    Object tmpResponsePayload = null;

    try
    {
        // Set the payload.
        requestPayload = new PayloadData();
        requestPayload.putDataWithName("theParameterName",inputData);

        // Call the remote clientagent.
        tmpResponsePayload = processRequest(
            userName,
            authentication,
            TestClientAgent.CLASS_NAME,
            TestClientAgent.FUNCTION_DOWORK,
            requestPayload);

        // Cast the returned payload.
        responsePayload = (PayloadData)
            (tmpResponsePayload);

        // Pull out any needed return values from the payload as appropriate for the
        // return value of this method.
        retval = (String)(responsePayload.getDataWithName("theParameterName"));
```

Code 8-20: Sample Client Agent Method to Handle Callback

```
}
catch (Throwable e)
{
    LogHelper.error(
        PACKAGE_NAME,
        CLASS_NAME,
        METHOD_NAME,
        "Exception occurred",
        e);
}
```

Code 8-20: Sample Client Agent Method to Handle Callback (Continued)

3. Implement the processCallback method within the client agent.

   The processCallback method serves as the single entry point for the client agent and dispatches the incoming request locally based on the incoming method name.

   Code Sample 8-21 shows a sample processCallback method within a client agent.

```
public Object processCallback(
    String username,
    String authentication,
    String serviceName,
    String functionName,
    Object payload)
    throws ServiceException
{
    final String METHOD_NAME = "processCallback";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    Object retval = null;
    PayloadData requestPayload = null;
    PayloadData responsePayload = null;

    if ((functionName != null) && (functionName.length() > 0))
    {
        try
        {
            // Do a simple "if statement" dispatcher
            if (functionName.compareToIgnoreCase(FUNCTION_DOWORK) == 0)
            {
                // Cast the payload as needed.
                requestPayload = (PayloadData)(payload);

                // Pull any required parameters out of the payload
                // as needed for the typed local function.
                String theInputData = (String)(requestPayload.
                    getDataWithName("theParameterName"));

                // Call the specific local function
                String theOutputData = localdowork(theInputData);

                // Reuse the input payload for the return payload be clearing it out.
                requestPayload.removeAllData();
                responsePayload = requestPayload;

                // Fill the return payload with the appropriate parameters.
                responsePayload.putDataWithName(
                    "theParameterName",theOutputData);

                // Assign the response payload to the return
                // value of this method.
                retval = responsePayload;
            }
            else
            {
                LogHelper.error(
                    PACKAGE_NAME,
                    CLASS_NAME,
                    METHOD_NAME,
                    "Unknown function name ["+functionName+"]");
            }
```

Code 8-21: ProcessCallback Method on the Client Agent

```
        }
        catch (Throwable e)
        {
            LogHelper.error(
                PACKAGE_NAME,
                CLASS_NAME,
                METHOD_NAME,
                "Exception occurred”,
                e);
        }
    }
    else
    {
        LogHelper.error(
            PACKAGE_NAME,
            CLASS_NAME,
            METHOD_NAME,
            "Null or zero length function name");
    }

    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retval;
}
```

Code 8-21: ProcessCallback Method on the Client Agent (Continued)

4. Implement a local method within the client agent to perform the useful work of the callback on the *client* side, as shown in Code Sample 8-22.

```
private String localdowork(String inputData)
{
    final String METHOD_NAME = "localdowork";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    String retval = null;

    // This clientagent function does nothing except print out the inputData it
    // received and echo a hard-coded string.
    LogHelper.debug(
        PACKAGE_NAME,
        CLASS_NAME,
        METHOD_NAME,
        “Received: ["+inputData+"]");
    System.out.println("!!!!!!!!!!!!!!!!!!!!["+inputData+"]!!!!!!!!!!!!!!!!!!!!");
    retval = "Return data from TestClientAgent.localdowork()";
    LogHelper.debug(
        PACKAGE_NAME,
        CLASS_NAME,
        METHOD_NAME,
        "Returning: ["+retval+"]");

    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retval;
}
```

Code 8-22: Method on the Client Agent to Perform Callback Work

5. If time will have elapsed since the last server-client agent communication, you might want to check on the "health" of the client agent's connection to the remote service by using the ping method. The ping method is implemented for you on the ClientAgentBaseClass. This works for callbacks as well as callins.

Note: Services wanting to make callbacks to clients should NOT routinely use the ping method, as it makes a full round trip and unbounded use will cause performance problems. Under normal circumstances, a service should simply make the desired callback to the client (without calling ping first) and handle any exceptions that can result.

6. Use the callbackShutdown method to clean up (disconnect) any connections to the remote client after the service has finished using the callback client agent. Services that do callbacks must use this method when they finish with a callback client agent. Otherwise, connections from the application server to the remote clients will build up and out of file descriptors errors will eventually result.

```
void ping( String userName, String authenticationToken )
```

IMPLEMENTING A SERVICE TO SERVICE CALL

This section describes how to implement a service to service call using Chordiant 5 Foundation Server. You can use service to service calls to have a service issue a call to a peer service to request some work to be performed.

Figure 8-18 shows the general flow of a service to service call.

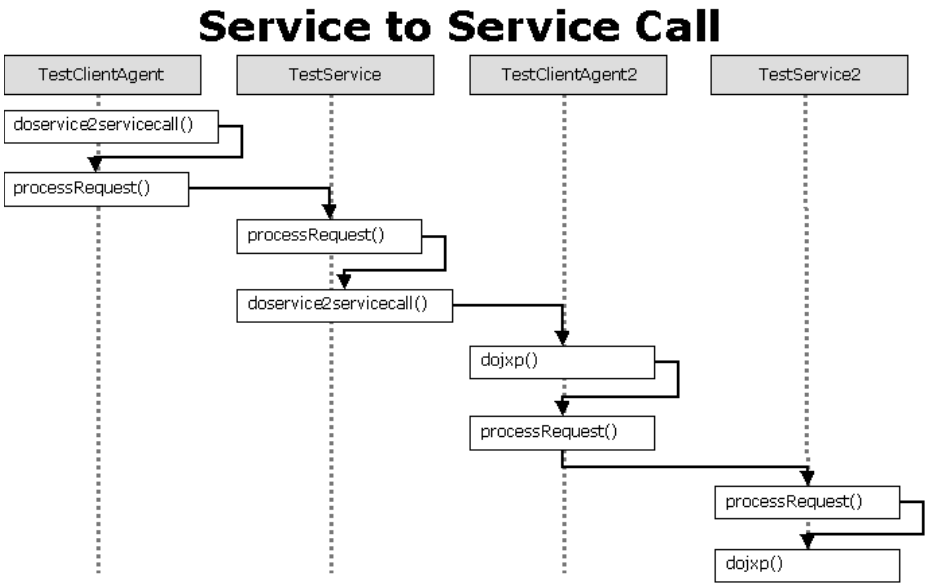


Figure 8-18: Flow of Service to Service Call

A service to service call is the same as a client agent's calling a service. Here, the service calls a client agent, which then works as any other client agent as it calls a service.

Note: This section provides the implementation details of a service to service call. If you are using client agents and services, you do not need to know anything more than how to use a client agent. These details are provided for your information.

To implement a service to service call:

1. Implement a method within the service to perform the call to the peer service.

   The method in the service issuing the call uses the ClientAgentHelper. getClientAgent method to get the specific client agent for the peer service. Once determined, the service can issue a call to the method within the client agent, along with the associated data.

   Code Sample 8-23 shows a sample method within a service to perform a call to a peer service.

```
private String doservice2servicecall(String inputData)
{
    final String METHOD_NAME = "doservice2servicecall";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    String retval = null;
    TestClientAgent2 testClientAgent2 = null;

    // This service function, simply calls a peer
    // service and returns the given return value.
    LogHelper.debug(
        PACKAGE_NAME,
        CLASS_NAME,
        METHOD_NAME,
        “Received: ["+inputData+"]");
    try
    {
        // Get a clientagent to the peer service.
        testClientAgent2 = (TestClientAgent2)
            ClientAgentHelper.getClientAgent(
                TestClientAgent2.CLASS_NAME);

        // Call the peer service
        retval = testClientAgent2.dojxp(
            getServiceLoginName(),
            getServiceAuthentication(),
            inputData);
    }
    catch (Throwable e)
    {
        LogHelper.error(
            PACKAGE_NAME,
            CLASS_NAME,
            METHOD_NAME,
            "Exception occurred”,
            e);
    }
    LogHelper.debug(
        PACKAGE_NAME,
        CLASS_NAME,
        METHOD_NAME,
        "Returning: ["+retval+"]");

    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retval;
}
```

Code 8-23: Service to Service Method

2. Within the *client agent for the peer service*, implement the method that you called from the originating service in the previous step. Include in that method a call to the processRequest method.

   This is the same as using processRequest in any client agent. The processRequest method serves as the single entry point for the service. A typical implementation for this method is to dispatch the incoming request to local functions, defined within the class, based on the function name parameter.

   Code Sample 8-24 shows a sample method within a client agent to handle the call from the originating service.

```
public String dojxp(String userName, String authenticationToken, String inputData)
{
    final String METHOD_NAME = "dojxp";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    String retval = null;
    PayloadData requestPayload = null;
    PayloadData responsePayload = null;
    Object tmpResponsePayload = null;

    try
    {
        // Set the payload.
        requestPayload = new PayloadData();
        requestPayload.putDataWithName(
            "theParameterName",inputData);

        // Call the remote JX service
        tmpResponsePayload =
            processRequest(
                userName,
                authenticationToken,
                TestService2.CLASS_NAME,
                TestService2.FUNCTION_DOJXP,
                requestPayload);

        // Cast the returned payload.
        responsePayload = (PayloadData)
            (tmpResponsePayload);

        // Pull out any needed return values from
        // the payload as appropriate for the return
        // value of this method.
        retval = (String)(responsePayload.
            getDataWithName("theParameterName"));
    }
    catch (Throwable e)
    {
```

Code 8-24: Client Agent Method for the Receiving Peer Service

```
        LogHelper.error(
            PACKAGE_NAME,
            CLASS_NAME,
            METHOD_NAME,
            "Exception occurred”,
            e);
    }

    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retval;
}
```

Code 8-24: Client Agent Method for the Receiving Peer Service (Continued)

These four service control methods are called automatically by the infrastructure. You do not write code to call these methods, but you can control what they do. You must implement them, but it is up to you how you want to use them.

- **setup**—initializations, including start caching (the business service base class includes caching for static data) and getting a Resource Manager. This is called once when the application starts up.
- **reinitialize**—to reset the service to its original starting state without shutting it down, for example refreshing the cache.(The business service base class includes caching for static data.) This method can be called several times.
- **status**—to assess status while the service is running. This method can be called several times.
- **shutdown**—to provide a clean shut down when the service has finished its function. This is called once just before exit.

Refer to Chapter 6, “Chordiant 5 Foundation Server Administration” for additional details.

3. Implement the local method for the peer service.

   Implement the method on the second service as you would any other method.

   The local method called by the peer service in the previous step is dojxp. It is a lengthy method and its implementation can be found in the example code, accessible through the `Documentation/Samples/Services` directory on the Installation CD. You can also access this directory through the Chordiant Tools Platform under **Help | Help Contents**.

   To give you an idea of a basic implementation you need, Code Sample 8-25 shows a basic method named donothing.

```
private String donothing(String inputData)
{
    final String METHOD_NAME = "donothing";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    String retval = null;

    // This service function, does nothing except
    // print out the inputData it recieved and echo
    // a hard coded string.
    System.out.println("TestService2.donothing() recieved:["+inputData+"]");
    retval = "Hello from TestService2.donothing()";
    System.out.println("TestService2.donothing() returning: ["+retval+"]");

    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);
    return retval;
}
```

Code 8-25: Simplified Method for Doing Work on Peer Service

Note: A service communicating to another service through a client agent will always communicate from one instance of the JX EJB to a different instance of the JX EJB. This follows J2EE standards for EJB communication. The application server automatically ensures that a service to service call will take place within the same JVM, rather than hopping between JVMs, so performance is not impacted.

CHORDIANT RESOURCE MANAGER

Chordiant 5 Foundation Server uses a Resource Manager to perform these functions:

- Read, load, and cache configuration and metadata information, stored in XML-formatted configuration files
- Act as an object factory for business objects, business object criteria objects, data accessor objects, and business object behavior objects

You should always use the object factory methods of the Resource Manager to return instances of a business object, data accessor, business object criteria object, or business object behavior object. This is because, among other reasons, the object factory is aware of overrides of business object behavior, and always returns the customized object, if available.

Figure 8-19 illustrates the object model for the Resource Manager.

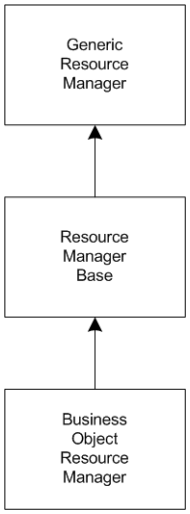


Figure 8-19: Resource Manager Object Model

Always use the BusinessObjectResourceManager. This resource should have all of the functionality you will need. If you feel that you need additional functionality, extend this class and override or add your own methods. Do not remove functionality from the BusinessObjectResourceManager.

The Resource Manager uses these three types of information:

- **Class** — Will return an instance of a class and keep it locally for your use.
- **String** — A generic type for anything you might want to keep locally, such as cache size or the default location for storing files. As long as the String is not null, it will be cached locally.
- **JNDI** — Will find the data source in the JNDI space and cache it locally for easy access.

Resource Manager Configuration

You configure a Resource Manager in your *{component}*.xml configuration file, for example, in the configuration file for your service.

You can use the Resource Manager to read and cache configuration information, stored in XML-formatted configuration files. A configuration section within the file enumerates the required resources, and includes one or more tag elements, specifying the resource key, and a corresponding value tag.

By convention, you should form the Section labels in the configuration using the Java package name of the class. For example, a configuration file could contain the following Section label, which you can use to enumerate resource names.

```
com.chordiant.bc.services.GenericService
```

To avoid name conflicts, the Section label is typically the service class name.

Table 8-2 describes the parameters you can configure for the Resource Manager. A sample Resource Manager file is shown in Code Sample 8-26 on page 173.

Note: You must include these tags in your configuration file if your service uses a resource manager, that is, if it will be hitting a database. By default, all services generated with the Business Component Generator include a Business Object Resource Manager, so your service can interact with a database. You will need to add the Resource Manager configuration parameters described in Table 8-2 to the configuration file.

If your service does not interact with a database, override the setup method in your customized class so the Resource Manager is not initialized.

PARAMETER TAG	DESCRIPTION
CMI_FILE	The path of the CMI file for the application that describes the JXP objects. You must specify the directory and file name of a well-formed CMI file. If you want to reference more than one CMI file, list the path to each file, separated by semicolons (;). There is no default value for this tag.
RESOURCE_TAG_ FOR_SQL_DSN	The string used to request a connection pool connection from the application server. This name must be registered with the application service, and the connection pool must be properly set up. The valid values for this parameter include any DataSourceName registered with the application server.
OBJECT_DIRECTORY	The root directory to which objects that have been transformed into XML strings are written. The files are named using the following syntax: `object directory.rdbPhysicalName. attributeName.rdbPrimaryKeyValue.` You can specify any valid, accessible directory for this parameter. There is no default value.
ENVIRONMENT_NAME	The value used during Global Unique Identifier Generation. This string ensures that the GUIDs are unique across multiple environments. You can specify string values less than or equal to seven characters in length. The system does not supply a value if you do not specify a value for this parameter.

Table 8-2: Resource Manager Configuration Parameters

Code Sample 8-26 shows a segment of a sample service configuration file containing the Resource Manager configuration. Notice that some values are using substitutions specified in the `master.dtd` file (see "master.dtd" on page 99 for details).

```
<Section>com.chordiant.bc.services.GenericService
    <Tag>ResourceName
        <Value>CMI_PATH</Value>
    </Tag>
    <Tag>ResourceName
        <Value>chordiantXAds</Value>
    </Tag>
    <Tag>ResourceName
        <Value>ENVIRONMENT_NAME</Value>
    </Tag>
    <Tag>ResourceName
        <Value>OBJECT_DIRECTORY</Value>
    </Tag>
</Section>
<Section>com.chordiant.bc.services.GenericService.CMI_PATH
    <Tag>ResourceType
        <Value>STRING</Value>
    </Tag>
    <Tag>ResourceValue
        <Value>&JXB_META_DATA_ROOT_DIRECTORY;/jxb/cmi.xml</Value>
    </Tag>
</Section>
<Section>com.chordiant.bc.services.GenericService.chordiantXAds
    <Tag>ResourceType
        <Value>JNDI</Value>
    </Tag>
    <Tag>ResourceValue
        <Value>&JXB_XA_DATASOURCE_REF;</Value>
    </Tag>
</Section>
<Section>com.chordiant.bc.services.GenericService.ENVIRONMENT_NAME
    <Tag>ResourceType
        <Value>STRING</Value>
    </Tag>
    <Tag>ResourceValue
        <Value>JXB</Value>
    </Tag>
</Section>
<Section>com.chordiant.bc.services.GenericService.OBJECT_DIRECTORY
    <Tag>ResourceType
        <Value>STRING</Value>
    </Tag>
    <Tag>ResourceValue
        <Value>&JXB_BINARY_DATA_STORAGE_ROOT_DIRECTORY;/bindata</Value>
    </Tag>
</Section>
```

Code 8-26: Sample Resource Manager Configuration

Note: The ResourceValues for both the CMI_PATH and OBJECT_DIRECTORY sections can have multiple paths listed. Separate each path with a semicolon (;).

Using this configuration, you could then instantiate an instance of the BusinessObjectResourceManager using the constructor format shown in Code Sample 8-27.

```
new BusinessObjectResourceManager("package name",
    "service class name (section name in configuration file)")
```

Code 8-27: BusinessObjectResourceManager Constructor Format

Code Sample 8-28 shows a sample constructor.

```
new BusinessObjectResourceManager("com.chordiant.bc.services",
    "com.chordiant.bc.services.GenericService")
```

Code 8-28: Sample BusinessObjectResourceManager Constructor

You can use each resource name, prefixed with the package name, as a section name with which to read a configuration for each resource (resource type and resource value).

Configuring for Multiple Data Sources

You can configure the Resource Manager for multiple data sources.

To configure the Resource Manager for more than one data source:

1. Configure the data sources in your development environment. In this example, there is one Oracle and one DB2 data source.
2. Configure your service to have multiple data sources, as shown in Code Sample 8-29.

```
<Section>com.chordiant.bd.services.AccountService
       <Tag>ResourceName
          <Value>chordiantNoXAds</Value>
       </Tag>
       <Tag>ResourceName
          <Value>chordiantDb2NoXAds</Value>
       </Tag>
...
<Section>com.chordiant.bd.services.AccountService.chordiantNoXAds
       <Tag>ResourceType
          <Value>JNDI</Value>
       </Tag>
       <Tag>ResourceValue
          <Value>chordiantNoXAds</Value>
       </Tag>
</Section>
<Section>com.chordiant.bd.services.AccountService.chordiantDb2NoXAds
       <Tag>ResourceType
          <Value>JNDI</Value>
       </Tag>
       <Tag>ResourceValue
          <Value>chordiantDb2NoXAds</Value>
       </Tag>
</Section>
```

Code 8-29: Service Configuration File Configured for Multiple Data Sources

3. Make sure the JNDI for your data source above matches the JNDI value you specified in Step 1.

4. The Resource Manager for account service configured here will have access to both the Oracle and DB2 data sources through calls like:

```
currentPool = (javax.sql.DataSource)myResourceManager.getResourceForName(DSN);
```

   where DSN is the data source resource name specified in the XML configuration file.

Using the Factory Methods

The Business Object Resource Manager includes a set of factory methods that you can use to create objects. The Business Object Resource Manager loads metadata information which drives the object factory, and returns specific objects such as the Business Object, the Business Object Criteria object, the Data Access object, and the Business Object Behavior object.

If you are making calls from the server side, call the Business Object Resource Manager (or Business Object Factory) methods directly. If you are making client-side calls, use the Business Object Factory Client Agent to reach the Business Object Factory.

The Business Object Resource Manager contains the following factory methods:

- **getBusinessObjectForName**—Returns an instance of a business object for the specified BO name.

```
public final Object getBusinessObjectForName(String name) throws Exception
```

Code 8-30: getBusinessObjectForName Method Signature

```
NullPartyRelationship nullPartyRelationship = (NullPartyRelationship)
    myResourceManager.getBusinessObjectForName(PartyRelationshipBehavior.CLASS_NAME);
```

Code 8-31: Using the getBusinessObjectForName Method

- **getBusinessObjectBehaviorForName**—Returns an instance of a Business Object Behavior object for the specified name.

```
public final BusinessObjectBehavior
    getBusinessObjectBehaviorForName(String name) throws Exception
```

Code 8-32: getBusinessObjectBehaviorForName Method Signature

```
PartyRelationshipBehavior partyRelBOB = (PartyRelationshipBehavior)
    myResourceManager.getBusinessObjectBehaviorForName(person.CLASS_NAME);
```

Code 8-33: Using the getBusinessObjectBehaviorForName Method

- **getBusinessObjectBehaviorForObject**—Returns an instance of a Business Object Behavior object for the specified name.

```
public final BusinessObjectBehavior
    getBusinessObjectBehaviorForObject(Object businessObject) throws Exception
```

Code 8-34: getBusinessObjectBehaviorForObject Method Signature

```
PartyRelationshipBehavior partyRelBOB = (PartyRelationshipBehavior)
    myResourceManager.getBusinessObjectBehaviorForObject(PartyRelationshipBO);
String relationshipCode = partyRelBOB.getPartyRelationshipTypeCode(
    roleTypeCode, relatedRoleTypeCode);
```

Code 8-35: Using the getBusinessObjectBehaviorForObject Method

- **getBusinessObjectCriteriaForName**—Returns an instance of a Business Object Criteria object for the specified name.

```
public final BusinessObjectCriteria
    getBusinessObjectCriteriaForName(String name) throws Exception
```

Code 8-36: getBusinessObjectCriteriaForName Method Signature

```
(PartyRelationshipViewTableCriteria)
    myResourceManager.getBusinessObjectCriteriaForObject(
    PartyRelationshipViewTableCriteria.CLASS_NAME);
```

Code 8-37: Using the getBusinessObjectCriteriaForObject Method

When specifying the name parameter, use the business object name with "Criteria" appended.

- **getDataAccessForName**—Returns an instance of a Data Access object for the specified name.

```
public final DataAccess getDataAccessForName(String name) throws Exception
```

Code 8-38: getDataAccessForName Method Signature

```
Vector allCommonObjectRoles = myResourceManager.getDataAccessForObject(
    PartyRoleTableDataAccess.CLASS_NAME).retrieveRay(prtBOC);
```

Code 8-39: Using the getDataAccessForObject Method

When specifying the name parameter, use the business object name with "DataAccess" appended.

For an example of using the Resource Manager for persistence, see "The Resource Manager and Persistence" on page 219 and "Example of Using Persistence Server" on page 231.

CUSTOMOBJECTS AND THE CUSTOMOBJECTHELPER

CustomObjects are a feature of the JX architecture. Their main purpose is to provide the ability to run Java code outside the constraints of the J2EE EJB framework yet still inside the container of the J2EE application server. Examples of CustomObjects include cache helpers, socket servers, and Remote Method Invocation (RMI) objects.

The JX CustomObject design follows a singleton pattern in that there is one instance of each configured JX CustomObject class per application server replicate (or JVM). In a development situation, this will usually mean that there is only one instance of each configured CustomObject, since there is typically only one replicate of the J2EE application server. In a production environment, however, you might have JX running on multiple physical servers with multiple J2EE application server replicates per server. In this situation, there will be many replicates of each configured CustomObject (one in each application server replicate). In fact, you will have more than one replicate running on the same server box.

For information on the SocketGatewayService, a specific CustomObject provided with Chordiant Foundation Server, see "Using the Foundation Server SocketGatewayService" on page 151.

CustomObject Requirements and Features

A JX CustomObject:

- Must be declared as a public class.
- Must implement a public default (no-parameter) constructor.
- May implement the JX ServiceControl interface.
- May implement a *non-blocking* main().
- Can be a subclass of any class hierarchy.
- Can perform valid Java logic

Note: Be careful with this last bullet. Remember that the CustomObject runs inside the same J2EE application server JVM that servlets, JSPs, and EJBs are running in. So unbounded processing scenarios could interfere with the health of the surrounding JVM.

Caution: In the near future, the J2EE specification will prohibit spawning threads from code within an EJB. For this reason, we suggest you avoid spawning threads from within CustomObjects.

CustomObjectHelper

Configuring CustomObjects

CustomObjects are instantiated and managed by the JX CustomObjectHelper when JX starts up. The JX CustomObjectHelper instantiates CustomObjects according to a CustomObject configuration section in the JX XML configuration files. Consequently, the classpath of the associated J2EE application server replicate must contain any needed classes to support the CustomObject.

The name of the CustomObject configuration section that the JX CustomObjectHelper uses is specified on the surrounding JVM's system properties through the -Dchordiant.customobject.configuration system property. In fact, it is possible to define different CustomObject configurations for each J2EE application server replicate in a distributed space.

Code Sample 8-40 shows a CustomObject configuration section defining three CustomObjects.

```
<Section>CustomObjectConfiguration
   <Tag> mycfgfile.name
      <Value>examples.hello.HelloImpl</Value>
   </Tag>
   <Tag> mycfgfile.name
      <Value>examples.hello.JXServerSocketTester</Value>
   </Tag>
   <Tag> mycfgfile.name
      <Value>com.chordiant.custom.mytesters.JXServerSocketTester</Value>
   </Tag>
</Section>
```

Code 8-40: CustomObject Configuration Section

The CustomObjectsHelper is in the package com.chordiant.core.customobjects.CustomObjectsHelper.

Managing CustomObjects

CustomObjects can implement an optional interface that enables them to be managed externally. The JX ServiceControl interface enables CustomObjects to implement four standard commands:

- setup
- shutdown
- refresh (typically a combination of shutdown and setup commands)
- status

The CustomObjectHelper will invoke the setup command on a CustomObject after instantiating it. The CustomObjectHelper will also call the shutdown command at the time that the JX application is being brought down. CustomObjects that implement the JX ServiceControl interface can be monitored and managed through the JX Administration Tool. See "Chordiant 5 Foundation Server Administration" on page 63 for more information.

In addition to the communicating through the ServiceControl interface, the CustomObjectHelper, also invokes the CustomObject's main method, if it exists.

Here is the order in which the CustomObjectHelper interacts with configured CustomObjects when starting up:

1. The CustomObjectHelper instantiates all configured CustomObjects. This invokes the CustomObject constructors.
2. For each CustomObject, the CustomObjectHelper invokes the setup command via the ServiceControl interface (if implemented).
3. For each CustomObject, the CustomObjectHelper invokes the main method (if implemented).

Minimally, the CustomObjectHelper will instantiate configured CustomObjects. If the CustomObjectHelper is for any reason unable to create a configured CustomObject, it will log an error and move on to the next CustomObject.

The ServiceControl Interface

The ServiceControl interface (com.chordiant.service.ServiceControl) consists of a single method:

ServiceControlResponse serviceControl(ServiceControlRequest request)

Inside this method, the CustomObject can obtain the request command via the ServiceControlRequest interface and determine its response.

The following constants represent the standard request commands:

BaseServiceControlRequest.SETUP_COMMAND

BaseServiceControlRequest.SHUTDOWN_COMMAND

BaseServiceControlRequest.REINITIALIZE_COMMAND

BaseServiceControlRequest.STATUS_COMMAND

The ServiceControlResponse interface enables you to provide the caller with a textual feedback on the processing of the request and a success indicator. A common class that implements this interface is BaseServiceControlResponse.

Code Sample 8-41 illustrates the Java imports and implementation required for participating in the JX ServiceControl framework.

```
import com.chordiant.service.ServiceControl;
import com.chordiant.service.CommandException;
import com.chordiant.service.ServiceControlRequest;
import com.chordiant.service.ServiceControlResponse;
import com.chordiant.service.BaseServiceControlRequest;
import com.chordiant.service.BaseServiceControlResponse;
import com.chordiant.core.ThinClientStaticHelper;
…
public ServiceControlResponse serviceControl(ServiceControlRequest request) throws CommandException
{
    ServiceControlResponse response = null;
    String command = null;

    if (request == null)
    {
        String errorString = "Request is null";
        throw new CommandException(errorString);
    }

    command = request.getCommand();
    if ((command == null) || (command.length() == 0))
    {
        String errorString = "Command is null or zero length";
        throw new CommandException(errorString);
    }

    if (command.compareToIgnoreCase(BaseServiceControlRequest.SETUP_COMMAND) == 0)
    {
       response = setup(request);
    }
```

Code 8-41: Required Code for JX ServiceControl Framework

```
    else if (command.compareToIgnoreCase(BaseServiceControlRequest.SHUTDOWN_COMMAND) == 0)
    {
        response = shutdown(request);
    }
    else if (command.compareToIgnoreCase(BaseServiceControlRequest.REINITIALIZE_COMMAND) == 0)
    {
        response = reinitialize(request);
    }
    else if (command.compareToIgnoreCase(BaseServiceControlRequest.STATUS_COMMAND) == 0)
    {
        response = status(request);
    }
    else
    {
        String errorString = "Unknown command: ["+command+"]";
        throw new CommandException(errorString);
    }

    return response;
}

private ServiceControlResponse setup(ServiceControlRequest theRequest) throws CommandException
{
    ServiceControlResponse retval = null;
    // Process the request …
    retval = new BaseServiceControlResponse();
    retval.setResponse("["+METHOD_NAME+"] was Ok");
    return retval;
}

private ServiceControlResponse shutdown(ServiceControlRequest theRequest) throws CommandException
{
    ServiceControlResponse retval = null;
    // Process the request …
    retval = new BaseServiceControlResponse();
    retval.setResponse("["+METHOD_NAME+"] was Ok");
    return retval;
}

private ServiceControlResponse reinitialize(ServiceControlRequest theRequest) throws CommandException
{
    ServiceControlResponse retval = null;
    // Process the request …
    retval = new BaseServiceControlResponse();
    retval.setResponse("["+METHOD_NAME+"] was Ok");
    return retval;
}

private ServiceControlResponse status(ServiceControlRequest theRequest) throws CommandException
{
    ServiceControlResponse retval = null;
    // Process the request …
    retval = new BaseServiceControlResponse();
    retval.setResponse("["+METHOD_NAME+"] was Ok");
    return retval;
}
```

Code 8-41: Required Code for JX ServiceControl Framework (Continued)

Chapter 9

Chordiant Persistence Server

Chordiant Persistence Server enables a service to access information in an Enterprise Information System (EIS), such as an SQL-based RDBMS, WebSphere MQ, or CICS system, and perform persistence operations to store and retrieve data from these data stores. Persistence Server does this by providing a standard CRUD (Create, Retrieve, Update, and Delete) interface to business services.

Figure 9-1 illustrates the *logical* representation of the Persistence Server layer. Note that in a deployed model, the Persistence Server, EJBs, and servlets all typically reside in the same JVM.

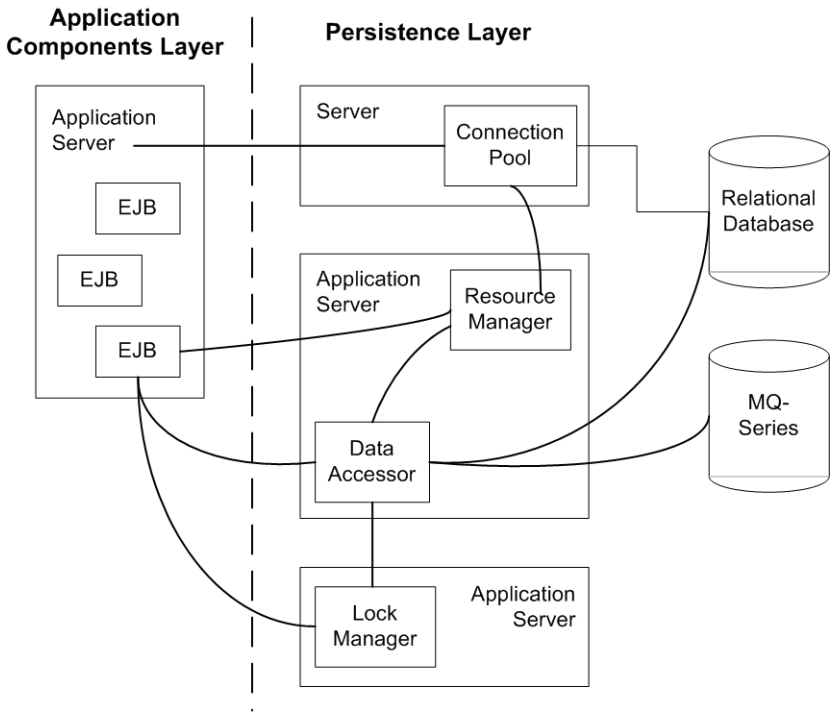


Figure 9-1: Logical Representation of Persistence Server Architecture

You can use the Chordiant Persistence Server component to add persistence functionality to the business services you write. Persistence Server offers these capabilities:

- **An XML-based meta model** — The meta model is fully extensible, enabling you to add new information about business objects within your application.
- **Object-oriented design tool UML Extenders** — Using the supplied UML Extenders for Rational Rose, along with ones you can develop, you can have Persistence Server automatically create the metadata files used by other parts of the system.
- **An extensible code generator** — You can use standard object-oriented design tools to generate the Java classes for the Data Accessor interface, defining standard CRUD operations.
- **Advanced plug-in connectors** — Using an extensible architecture, you can create new data connectors, in addition to using the SQL, and WebSphere MQ connectors supplied with Chordiant Persistence Server.

Note: Chordiant Persistence Server offers a superset of Java Connector Architecture (JCA) functionality. In particular, Persistence Server supports any JCA connector, but is not bound to any JCA meta model. And while JCA has a specific connector interface, Persistence Server supports that interface and others.

It is important to note that Chordiant Persistence Server uses existing object-oriented technology at its base. This technology includes XML Metadata Interchange (XMI), used by object-oriented design tools to output models, and Extensible Stylesheet Language (XSL) used to format XML. With this technology, Chordiant Persistence Server can leverage object-oriented design tools and can be customized through XSL stylesheets.

THE DEVELOPMENT CYCLE

The development cycle for Chordiant Persistence Server includes the participation of several people occupying the following roles when incorporating persistence into the application design:

- Business Analyst
- User Interface Designer
- Database Specialist
- Application Developer

Figure 9-2 illustrates the development cycle when working with Chordiant Persistence Server.

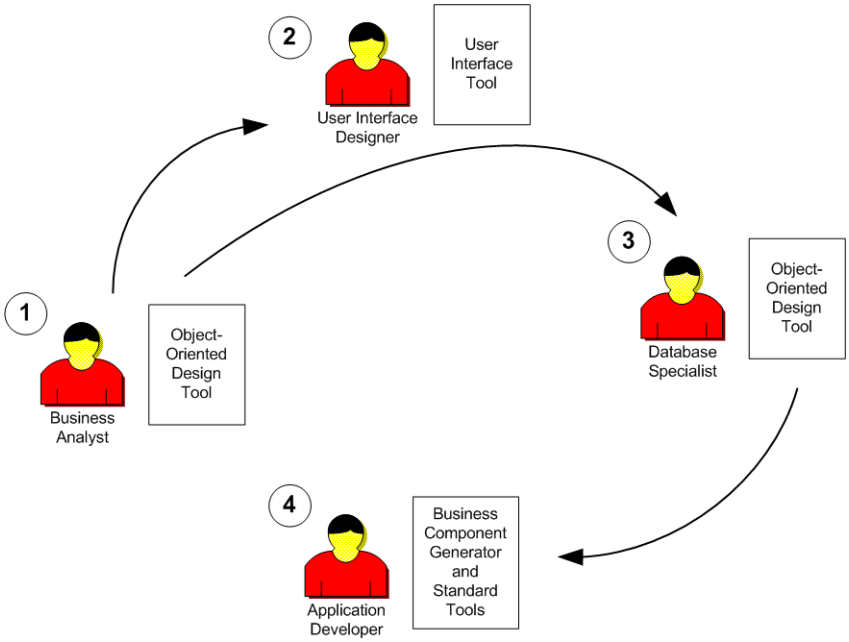


Figure 9-2: Chordiant Persistence Server Development Cycle

1. The Business Analyst creates the conceptual model of the business data and business behavior using an object-oriented design tool such as Rational Rose.

   The Business Analyst works in the context of business objects, such as customers and accounts, as well as behaviors related to these objects, such as creating a customer and listing an account. In creating the conceptual model, the Business Analyst does not need to consider Foundation Server-related issues, focusing instead on capturing the business objects associated with the organization.

   Once the Business Analyst has created the conceptual model and output the XMI definition, both the User Interface Designer and the Application Developer can begin using the model to create the interface for the web application and business services respectively.

   This enables parallel development to occur as the developers code to the interface (but without being able to actually run the application or service at this stage).

2. The User Interface Designer builds the screens and creates the front end for the web applications.

   The User Interface Designer uses Chordiant Interaction Designer to create the web application interface in relation to the business objects and behavior defined by the Business Analyst.

3. The Database Specialist maps the conceptual model created by the Business Analyst to the database environment within the organization.

   The Database Specialist maps objects defined by the Business Analyst to tables within the database, or creates new tables as required. The Database Specialist works with the underlying data store, such as an SQL-based RDBMS or WebSphere MQ data store, and uses the UML Extender for Rational Rose with the model to define the persistence needs of the application.

4. The Application Developer takes the output of the object model and runs the Business Component Generator to create the Java classes required for persistence. The Application Developer then modifies the Java code.

   Typically, the Application Developer creates business services that capture the business logic of the application and perform create, read, update, and delete (CRUD) operations on the objects defined by the Business Analyst.

Persistence Server Process Flow

Chordiant Persistence Server offers significant flexibility in how Business Analysts, Users Interface Designers, Database Specialists, and Application Developers work together to produce metadata and programatic interfaces for supporting persistence within applications.

This section describes the process flow for developing systems using Persistence Server, along with the metadata and interfaces produced at each step of the process, as illustrated in Figure 9-3.

Numbers in the diagram correspond to step numbers in the text section beginning on page 186.

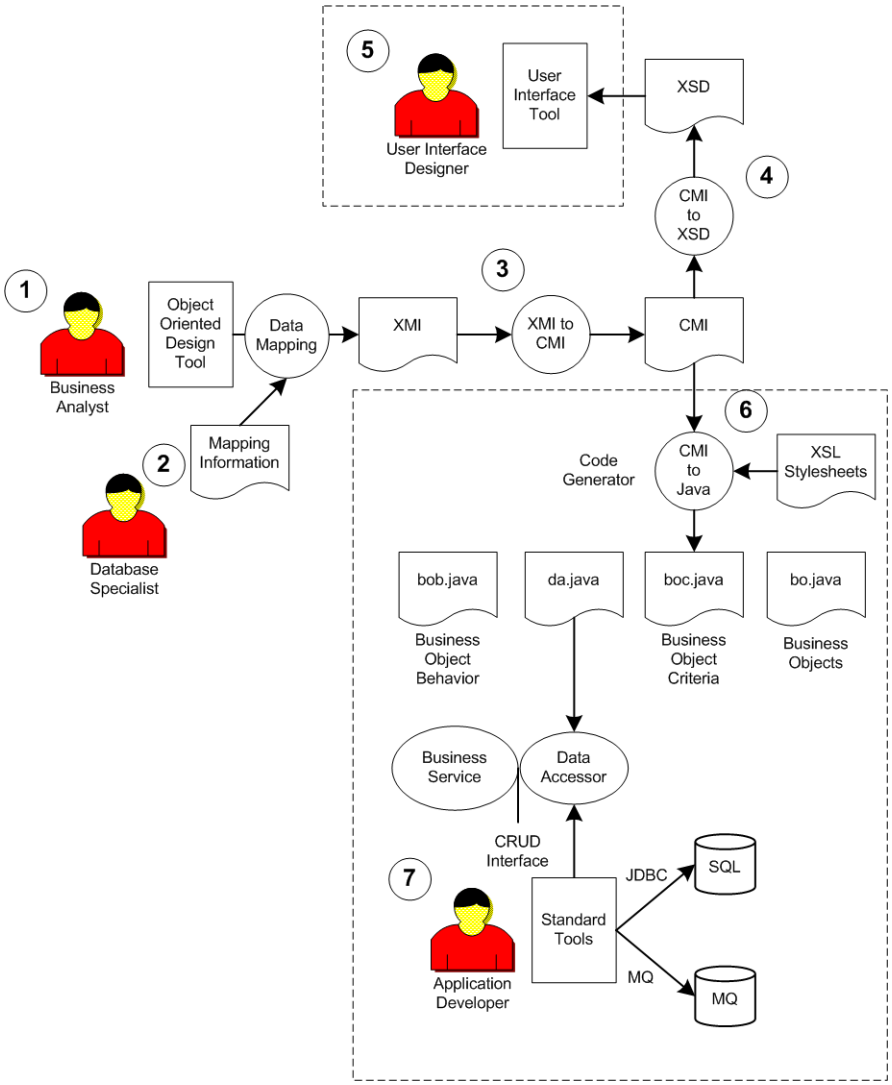


Figure 9-3: Using Chordiant Persistence Server

The process flow for developing applications using Chordiant Persistence Server is:

1. The Business Analyst creates the conceptual model of the business data and business behavior using an object-oriented design tool.

   The direct output from the object-oriented design tool is an XML Metadata Interchange (XMI) file. While XMI offers a portable format between object-oriented design tools, the format contains extra information not required by Persistence Server, such as visualization details, among other information.

2. The Business Analyst, Database Specialist, or both use mapping information to relate the business data to specifics within the database environment.

   Using the Chordiant UML Extender for Rational Rose, the Database Specialist maps the business data to the underlying data store, adding Persistence Server metadata to the CMI file, and thus defining the persistence information for the application. In mapping the data, the Database Specialist must take into account the data types supported in the underlying data store, as described in "Data Type Support" on page 225.

3. The Business Analyst generates a Chordiant Metadata Information (CMI) file from the XMI output.

   The generated CMI file contains the core information concerning the conceptual model, as illustrated in Code Sample 9-1.

```
<?xml version ="1.0" ?>
<root>
    <package>
        <name>test.jx.simple</name>
            <class>
                <name>MyObject</name>
                <parentClass>Object</parentClass>
                    <attribute>
                        <name>Type</name>
                        <javaType>java.lang.String</javaType>
                        <multiplicity>1..1</multiplicity>
                    </attribute>
            </class>
    </package>
</root>
```

Code 9-1: Sample CMI File

   For more information about CMI and metadata, see the *Chordiant 5 Foundation Server Application Components Developer's Guide.*

4. The Business Analyst optionally generates an XML Schema Definition (XSD) file for use by the User Interface Designer.

5. The User Interface Designer uses the Chordiant Interaction Designer to build the screens and create the front end for the web applications.

6. The Application Developer generates the Java classes for the Persistence Server interface, using the Business Component Generator.

   The Business Component Generator generates these Java classes:

   — ***{yourobject}*.java**: Contains the business objects (BOs) for the Persistence Server interface. The objects define only basic getter and setter methods.

   — ***{yourobject}*Behavior.java**: Contains a skeleton for the server-side business object behavior methods.

   — ***{yourobject}*Criteria.java:** Contains methods defining the business object criteria. Business object criteria enable developers to identify groups of business data, such as equivalences, ranges, and sets. For example, you could use business object criteria to find all customers within a certain age, or with a certain last name.

   — ***{yourobject}*DataAccess_*{databasename}*.java:** Defines the Data Accessor, which offers methods for interacting with the specific data store, such as Oracle or DB2, using SQL or WebSphere MQ. The Data Accessor runs on the application server, and can only be called by a service—the Data Accessor cannot be invoked by a client. The parent class, ***{yourobject}***`DataAccess.java`, has minimal functionality. Use the data accessor with the database name specified.

   The Business Component Generator uses two inputs to generate the Java classes: the CMI file, and an Extensible Style Language (XSL) stylesheet. Based on definitions in the XSL stylesheet, the tool generates the appropriate Java code for the particular underlying data store.

   By customizing the XSL stylesheet, you can change the Java classes you generate. See "Chordiant Persistence Server and XSL Stylesheets" on page 233 for more information.

7. The Application Developer creates services and client agents using the CRUD interface defined within the Data Accessor.

DATA ACCESSOR OVERVIEW

The Data Accessor is a server-side component that enables you to create business services that are decoupled from the data source, such as a database or file. You access the functionality of the Data Accessor through an Application Programmer Interface (API) consisting of 31 methods.

Business services that require a Data Accessor object issue a request to the Resource Manager, along with the name of the requested Data Accessor. The Resource Manager returns an instantiated object containing the interface. For information on the Resource Manager, refer to "Chordiant Resource Manager" on page 170 and to the *Chordiant 5 Foundation Server Application Components Developer's Guide*.

Chordiant 5 Foundation Server offers three Data Accessors for the following data stores as part of the standard implementation:

- Oracle
- DB2UDB
- WebSphere MQ

You can optionally implement a Data Accessor for other data stores, as required.

Note: Class attribute names are limited to 27 characters or fewer. They are used as unique identifiers for joins and are limited in length by the database. This example shows how the attribute name is used.

select *{table name1}.{column name}* as rs_*{attribute name}* from *{table name1},{table name2}* where *{table name1}.{primary key column}*=*{table name2}.{primary key column}*

This section describes the following topics related to the Data Accessor:

Interface Notation

The Data Accessor uses these four concepts to represent the number and range of records returned by data access operations:

- Point
- Set
- Ray
- Segment

This section describes each of these concepts, and illustrates an example use of the concept within the Data Accessor API.

Points

A point represents a single record within a table.

```
x={unique_value}
```

Note: You can also define a point with more than one primary key. The combination of the primary keys must be unique to specify a point. For example, you can define a point like this:

```
PrimaryKey1 + PrimaryKey2 = {unique_value}
```

You must specify all primary keys in the metadata. For more information, refer to the "Metadata" chapter of the *Chordiant 5 Foundation Server Application Components Developer's Guide*.

Figure 9-4 illustrates the selection of points within two tables.

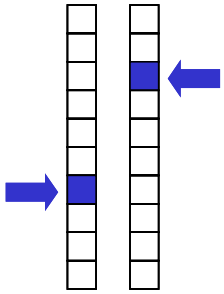


Figure 9-4: Points

The createPoint method works with a point within a table.

```
createPoint(Object);
```

Methods dealing with points throw an UnexpectedMultipleRecordsException exception if more than one record meets your criteria.

Sets

A set represents a grouping of non-sequenced records, based on uniquely-specified primary keys, within a table. Use this format to create a set.

```
x={value1}, {value2}, ...
```

Figure 9-5 illustrates the selection of sets within two tables.

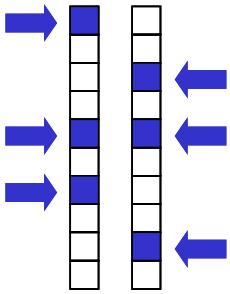


Figure 9-5: Sets

Here is an example of a method that works with a set within a table.

```
updateSet(BusinessObjectVector);
```

Rays

A ray represents all records above or below a value within a table, and is defined by a single BusinessObjectCriteria. A ray can be defined in any of the following manners, depending on whether the ray is inclusive or exclusive:

```
x > {value1}
x >= {value1}
x < {value1}
x <= {value1}
```

Figure 9-6 illustrates the selection of rays within two tables.

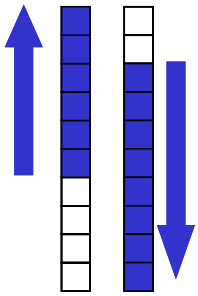


Figure 9-6: Rays

Here is an example of a method that works with a ray within a table.

```
deleteRay(BusinessObjectCriteria);
```

You specify the BusinessObjectCriteria to define the beginning and direction of the SELECT statement.

Segments

A segment represents all records within a range in a table, and is defined by two business object criteria. A segment can be defined as shown in the examples below, depending on whether the segment is inclusive or exclusive:

```
x >= {value1} AND X <= {value2}
x > {value1} AND X < {value2}
x <= {value1} AND X < {value2}
x < {value1} AND X <= {value2}
```

Note that all operations defining segments perform a logical AND operation. The Data Accessor API does not provide multi-segment operations. When you need to implement a logical OR operation, you can do so using multiple calls.

Figure 9-7 illustrates the selection of segments within two tables.

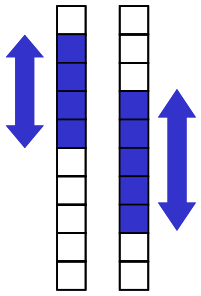


Figure 9-7: Segments

Here is an example of a method that works with a segment within a table.

```
retrieveSegment(BusinessObjectCriteria1, BusinessObjectCriteria2);
```

You specify the BusinessObjectCriteria to define the beginning and end of the SELECT statement.

Data Access Methods

This section outlines the Data Access methods supported by Relational Database Management Systems using SQL and WebSphere MQ data stores.

Table 9-1 outlines the available methods supported by the SQL Data Accessor.

SQL Data Access Methods	
• countPoint	• updatePoint
• countRay	• updatePointOptimistic
• countSegment	• updatePointPessimistic
• createPoint	• updateSegment
• createSet	• updateSet
• retrievePoint	• updateSetOptimistic
• retrieveSet	• updateSetPessimistic
• retrieveRay	• updateRay
• retrieveSegment	• deletePoint
• retrieveRayOrdered	• deleteRay
• retrieveSegmentOrdered	• deleteSegment
• retrievePointPessimistic	• deleteSet
• retrieveSetPessimistic	• deletePointOptimistic
• retrieveRayPessimistic	• deletePointPessimistic
• retrieveSegmentPessimistic	• deleteSetOptimistic
	• deleteSetPessimistic

Table 9-1: SQL Data Access Methods

Table 9-2 outlines the available methods supported by the WebSphere MQ Data Accessor.

WEBSPHERE MQ DATA ACCESS METHODS	
• countPoint	• updatePointPessimistic
• createPoint	• updateSet
• createSet	• updateSetPessimistic
• retrievePoint	• deletePoint
• retrieveSet	• deleteSet
• retrievePointPessimistic	• deletePointPessimistic
• retrieveSegmentPessimistic	• deleteSetPessimistic
• updatePoint	

Table 9-2: WebSphere MQ Data Access Methods

The methods you select depend on the number and range of records you need returned. For example, in the case of the retrieve operations, you can select from among the following methods to retrieve one or more records, depending on your requirement:

- **retrievePoint** — Retrieves a single person by a unique value
- **retrieveSet** — Retrieves a group of people, each with unique values
- **retrieveRay** — Retrieves a group of people of age 25 and above
- **retrieveSegment** — Retrieves a group of people of age 29 to 30

Performance Tip for Updating Data

When you make a small change to an object, you must update all of the attributes in the database. Some objects can be quite large and can contain information that is extraneous to your objective. In these cases, persisting the large objects might be overly expensive. For most of these large objects, only certain attributes are read and written to frequently. To enhance performance, you can consider creating a *summary view* of a large object, which contains only the subset of attributes which are frequently read or written to. The summary view is a common object-oriented design concept. By using a shallow object, you avoid moving data that is not relevant to your transaction, thus saving time and resources. To improve your performance, model summary views alongside the larger object they are related to and use them when appropriate to avoid handling excess data.

By default, null values are updated to the database.

Global Unique Identifier (GUID) Generation

The Chordiant Global Unique Identifier (GUID) is a system-wide identifier that is guaranteed to be unique. Chordiant 5 Foundation Server uses the GUID to:

- Uniquely identify a record within the system

  When defining the primary key for a record, you can instruct the system to create a GUID as part of the primary key auto generation method, either within your object model or within the CMI file.

- Issue a unique lock token for a record

  Optimistic locking uses a lock token that is updated each time the record is updated.

The GUID has the following format:

```
SEED_TIMS _ENVIRONMENTCOUNTER
```

The interpretation of each element is shown here:

- **SEED**—An operational space-unique value for each Resource Manager, represented by a 20-character string.
- **TIMS**—Time in milliseconds, represented by a 20-character string.
- **ENVIRONMENTCOUNTER**—A counter managed by the Resource Manager, represented by a five-character string.

Note: Underscores separate the three fields in the GUID. All fields within the GUID are fixed length, with numeric fields right-justified and zero-padded.

Specifying the GUID

To specify automatic GUID generation within your object model using Chordiant's UML Extender for Rational Rose:

1. In your object model, open the class specifications.
2. In the Class Specifications window, select the JXP SQL tab.

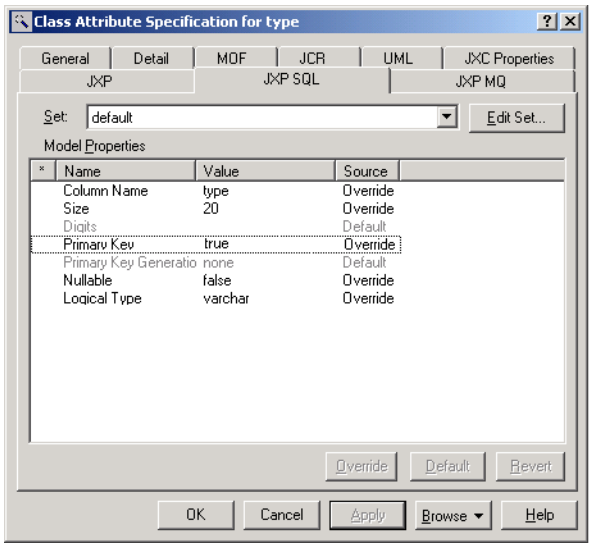


Figure 9-8: Specifying Primary Key Generation within Rational Rose

3. Click in the **Value** column next to Primary Key Generation and select Auto.
4. Click **OK**.

To specify automatic GUID generation within the CMI file, use this tag.

```
<rdbPrimaryKeyGenerationType> true </rdbPrimaryKeyGenerationType>
```

Business Object Criteria

Use business object criteria to identify groups of business data, such as ranges and equalities, when working with the Persistence Server API. The business object criteria methods are created automatically when you generate the Java classes using the object-oriented design tools, as described in "Persistence Server Process Flow" on page 185.

When using the business object criteria, you can specify your equivalency criteria using these constants:

- **CRITERIA_EQUAL** — equality criteria
- **CRITERIA_NOT_EQUAL** — inequality criteria
- **CRITERIA_GREATER** — greater than criteria
- **CRITERIA_LESSER** — less than criteria
- **CRITERIA_EQUAL_GREATER** — greater than or equal to criteria
- **CRITERIA_EQUAL_LESSER** — less than or equal criteria

The business object criteria for a particular object define methods that enable you to define rays and segments. You can also use business object criteria to identify additional information for an attribute. This is because Chordiant Persistence Server associates a criteria with every attribute within an object.

For example, you could use business object criteria to find all customers matching a particular income range, or living in a specific county. You could also use business object criteria to order your returned results.

Code Sample 9-2 illustrates how to retrieve all persons living in "York" county with last names greater than "Smith".

```
PersonCriteria myPerson = new PersonCriteria();
myPerson.setCounty("York");
myPerson.setCountyCriteria(BusinessObjectCriteria.CRITERIA_EQUAL);
myPerson.setLastName("Smith");
myPerson.setLastNameCriteria(BusinessObjectCriteria.CRITERIA_GREATER);
DataAccess myDA =
    resourceManager.getDataAccessForName(myPerson.CLASS_NAME + "DataAccess");
Vector returnPersons = myDA.retrieveRay(myPerson);
```

Code 9-2: Code Sample Demonstrating Criteria

Optimistic and Pessimistic Locking

The Data Accessor offers these two data record locking mechanisms for the Foundation Server, both of which are data source independent:

- **Optimistic locking** — A column locking strategy where each record has a lock column that is checked before update operations are performed. Optimistic locking establishes a shared (non-exclusive) lock on the record.
- **Pessimistic locking** — Data is retrieved and locked. Pessimistic locking establishes an exclusive lock on the record.

Notes: The locking strategy you use for any particular record should be consistent within a data source. See "Caution: Two Locking Strategies on Same Data" on page 202 for details.

Locking strategies are defined at the class level. A class can specify either optimistic or pessimistic locking, but not both. Classes used to access the same data should use the same locking strategy. See page 202 for more information.

Do not mix locking strategies within an object graph. All objects within a graph should have the same strategy as the head object.

Both optimistic and pessimistic locking adhere to these rules:

- All locks have a configurable life span, after which your application can release the lock.

  This prevents deadlocks (stalemates between users) and orphan locks (locks left over at the end of a process) from accumulating in the system.

- The business service layer maintains conformance to locking strategies.

  The business service must implement the business process choices and business actions with regard to lock management. The business service, or a delegated service, is also responsible for all exception handling that might occur during lock conflicts.

This section describes these topics related to optimistic and pessimistic locking:

Optimistic Locking

Optimistic locking uses a token to coordinate the update of records in a table. In a distributed environment, this token must be unique and not repeated throughout the entire environment. Chordiant Persistence Server uses the Chordiant GUID as the token for optimistic locking.

In optimistic locking, the system updates the lock token every time a record is written. When another process attempts to write to that record, the system compares the lock token received when the record was retrieved against the current lock token. If the two are different, a LockUnavailableException is thrown. When this happens, the second process trying to write to the record must reconcile its changes with the new record in the table.

Optimistic locking methods have the following operational behavior with respect to the basic data store operations:

- **Create** — With the createPoint and createSet methods, the system generates a new lock token, and stores it in the database.
- **Retrieve** — The lock token stored in the database is returned (on every call).
- **Update** — If the lock token previously retrieved matches the one in the database, the update operation is performed. Otherwise, an exception is raised. The lock token is updated whenever the row is updated. Note that the update operation is not supported for rays and segments.
- **Delete** — The record is deleted using the lock token. Note that the delete operation is not supported for rays and segments.

Optimistic locking has a *first write, next write* failure pattern. This means that in the case when two entities read a particular record, if the first entity then performs a subsequent write, the second entity will receive a optimistic lock exception if it also tries to write. At this point, the second entity must re-read the new record and reconcile the differences between their changes and the new record.

Note: Optimistic locking does not place an exclusive lock on the record, and therefore, there is a possibility of operational data loss since reconciling of the differences between records is dependent upon the defined business and application processes.

To use optimistic locking on a class, the lock strategy on the class should be set to optimistic. One of the class attributes must be a String and marked as the LockField (LockField as true). In fact, the LockField should be in addition to other attributes and should not be modified. The LockField is controlled by the system and the unique lock value will be generated and placed within this attribute and also within the corresponding column.

Figure 9-9 illustrates an example of portions of a CMI file used to define optimistic locking for the Customers table. Note that Col_1 has the LockField specified as true, and the javaType specified as java.lang.String, as required.

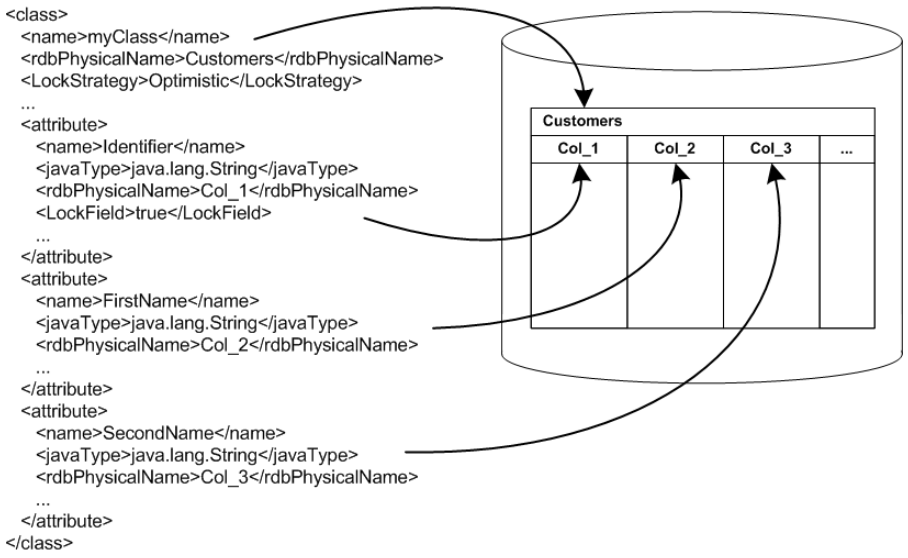


Figure 9-9: Specifying a Lock Field

Optimistic locking is useful when many people might want to access the same data concurrently. It allows for maximum throughput of data that is not possible under pessimistic locking strategy, where one person's accessing a record would prevent all others from even viewing it.

Pessimistic Locking

Pessimistic locking enables you to establish exclusive locks on a record. However, when using pessimistic locks, your business services must account for the possibility of record deadlocks (stalemates between users) and orphan locks (locks left over at the end of a process).

With pessimistic locking, only one process can update a record at any given time. When a process attempts to retrieve a record that is locked (using the retrievePessimistic method), the LockUnavailableException is thrown.

For this reason, Chordiant recommends that you use pessimistic locking only on areas with low throughput. For maximum throughput of data, use optimistic locking if possible.

In general, consider these guidelines for preventing record deadlocks when using pessimistic locks:

- Always lock multiple records in a formal pattern.
- Try to acquire a lock, read, update, and release a lock all within one transaction block.
- If you can not secure multiple locks on records, narrow down the possible records so the window for locking is smaller.

In the case of orphan locks, each lock has a configurable life span to prevent locks from remaining in existence indefinitely. Specifically, each lock has a time stamp, with a resolution in milliseconds.

You can optionally have your application check each lock, at a defined time, to determine whether it has exceeded its life span. In cases when the life span is exceeded, your application can remove the lock. Configuring the life span to shorter values reduces the likelihood of record deadlocks and orphan locks from arising.

Note: Using the ID of a record, you can have any application unlock a record. This is useful in cases when the process that originally locked the record is no longer available to perform the unlock operation.

Optimistic and Pessimistic Locking in One Model

While you should not use both optimistic and pessimistic locking strategies in the same data (see "Caution: Two Locking Strategies on Same Data" on page 202), you can use both locking strategies in the same model.

Pessimistic locking uses an object's ID field as a lock field. Since all of the objects in a model are usually derived from the same base class, you cannot use the inherited ID as the lock field, since it would change the ID for the base class and all classes inherited from that base class. In this case, you should create a new GUID field to specify as the lock field for the pessimistic locking sections of your model. This way, it does not interfere with the sections of the model using optimistic locking.

Figure 9-10 illustrates an example using two locking strategies: an optimistic strategy is used on the car data, since it is fairly unlikely that two people will try to update engine or tire information simultaneously; a pessimistic strategy is used on the driver information, since this information can change more often and we want to avoid data collisions.

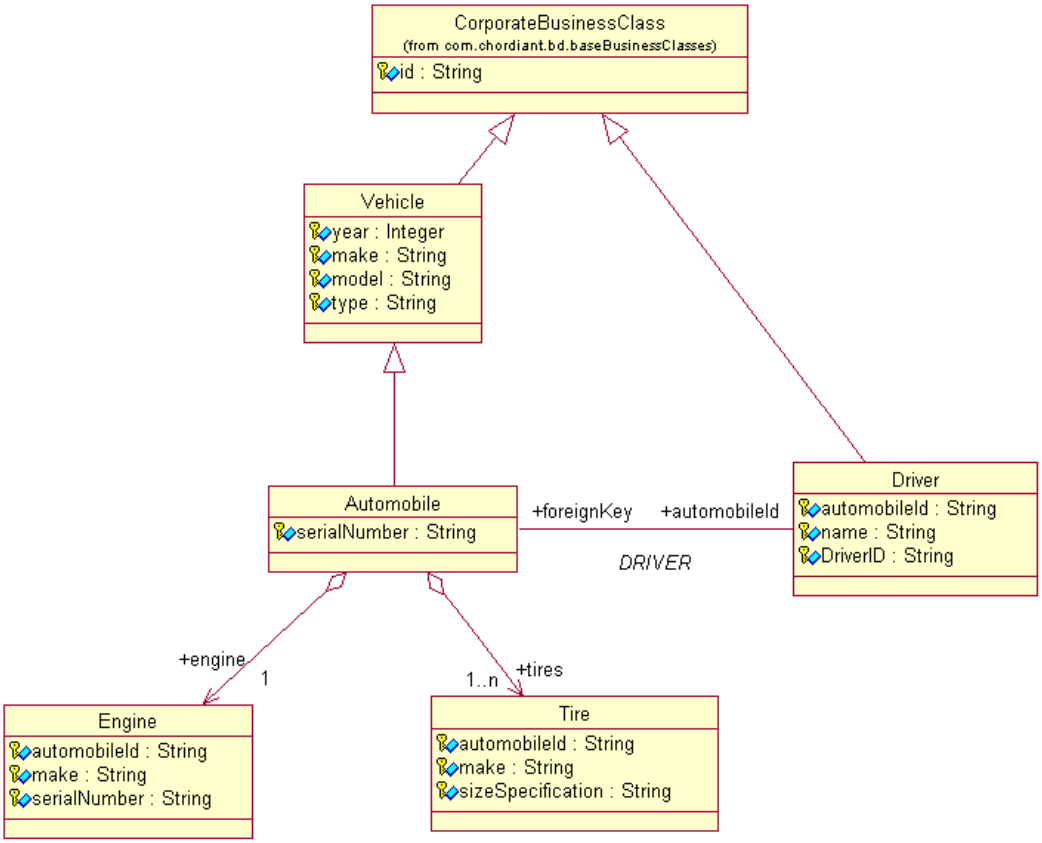


Figure 9-10: Optimistic and Pessimistic Locking in Same Model

Notice that each class inherits an ID attribute from the base class. If you were to change the ID for the driver by making it a lock field, this would change the ID for the base class as well as for the vehicle, the car and the car's engine and tires, making this the lock field for all. This is not desirable with a mixed-strategy model. The optimistic strategy requires a different lock field, so this would provide a second lock field. By creating a separate GUID attribute for the Driver class (called DriverID), setting this ID to a lock field does not interfere with the IDs of the other parts of the model.

Caution: Two Locking Strategies on Same Data

You must exercise caution when using both optimistic and pessimistic locks within your applications since Chordiant Persistence Server does not impose restrictions on the interaction between optimistic and pessimistic locks. For example, consider the example in Figure 9-11.

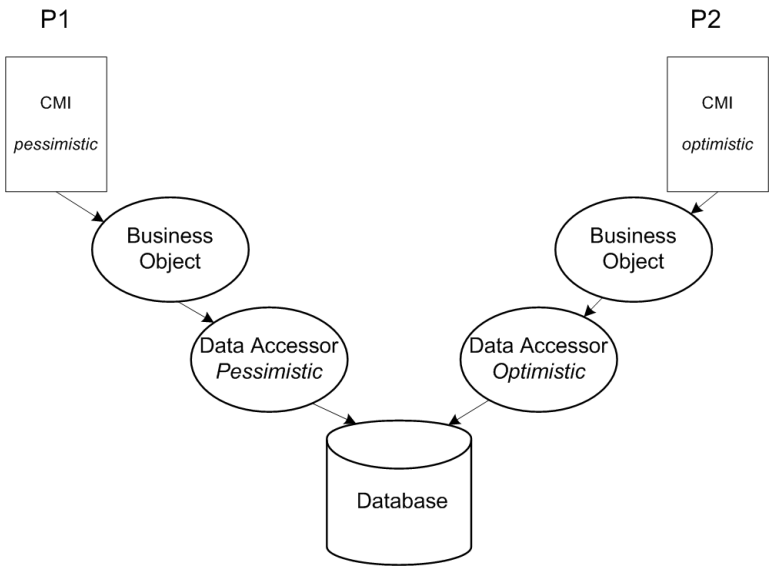


Figure 9-11: Caution—Using Both Optimistic and Pessimistic Locking Strategies for Same Data Can Cause Problems

This scenario could occur at four time points (T1 to T4):

T1: Process P1 uses retrievePointPessimistic to lock and retrieve record R1
T2: Process P2 uses retrievePoint to retrieve the same record, R1, using an optimistic lock
T3: Process P2 updates record R1 using updatePoint
T4: Process P1 uses updatePointPessimistic to update record R1 and unlock the record.

In this case, Process P1 will overwrite the changes made by Process P2, which is typically an unintended consequence.

Optimistic and Pessimistic Locking API

Chordiant Persistence Server offers a distinct Data Access Application Programming Interface (API) for both optimistic and pessimistic locking. Table 9-3 outlines the API available to both types of locking.

OPTIMISTIC INTERFACE	PESSIMISTIC INTERFACE
• updatePointOptimistic	• retrievePointPessimistic
• updateSetOptimistic	• retrieveSetPessimistic
• deletePointOptimistic	• retrieveRayPessimistic
• deleteSetOptimistic	• retrieveSegmentPessimistic
	• updatePointPessimistic
	• updateSetPessimistic
	• deletePointPessimistic
	• deleteSetPessimistic

Table 9-3: Optimistic and Pessimistic Locking API

Note: The UnexpectedMultipleRecordsException exception is thrown when more than one record is matched for an operation involving a point.

Optimistic Locking Interface

The optimistic locking interface includes:

- **updatePointOptimistic**—Updates a record based upon a unique value, derived from a combination of one or more primary keys. The record must have at least one primary key to be updated. The method uses a lock token to update and change the token while updating the record.

```
public abstract void updatePointOptimistic(java.lang.Object data)
throws java.lang.Exception
```

Code 9-3: updatePointOptimistic Method Signature

- **updateSetOptimistic**—Updates each element based upon a unique value, derived from a combination of one or more primary keys. The record must have at least one primary key to be updated.

```
public abstract void updateSetOptimistic(java.util.Vector data) throws
java.lang.Exception
```

Code 9-4: updateSetOptimistic Method Signature

- **deletePointOptimistic**—Deletes a locked record. Each record must have the unique value (based on one or more primary keys) and the lock token. When the lock token does not match the current lock token, the record is not deleted.

```
public abstract void deletePointOptimistic(java.lang.Object data)
throws java.lang.Exception
```

Code 9-5: deletePointOptimistic Method Signature

- **deleteSetOptimistic**—Deletes one or more locked records. Each record must have the primary key value and the lock token. When the lock token does not match the current lock token, the record is not deleted.

```
public abstract void deleteSetOptimistic(java.util.Vector data) throws
java.lang.Exception
```

Code 9-6: deleteSetOptimistic Method Signature

Pessimistic Locking Interface

The pessimistic locking interface includes:

- **retrievePointPessimistic**—Retrieves a single record matching the supplied data points, and locks the record. In cases when more than one record is retrieved, an exception is thrown. The method returns null when no matches are found.

```
public abstract java.lang.Object
retrievePointPessimistic(java.lang.Object data) throws
java.lang.Exception
```

Code 9-7: retrievePointPessimistic Method Signature

- **retrieveSetPessimistic**—Retrieves a single record for each supplied business object within the Vector, and locks each record. In cases when more than one record is retrieved for a business object, an exception is thrown. The method returns null if no matches are found.

```
public abstract java.util.Vector
retrieveSetPessimistic(java.util.Vector data) throws
java.lang.Exception
```

Code 9-8: retrieveSetPessimistic Method Signature

- **retrieveRayPessimistic**—Retrieves all records matching the supplied data points, and locks each record. In cases when no data points are supplied, all records are retrieved. The method returns null if no matches are found.

```
public abstract java.util.Vector
retrieveRayPessimistic(BusinessObjectCriteria data) throws
java.lang.Exception
```

Code 9-9: retrieveRayPessimistic Method Signature

- **retrieveSegmentPessimistic**—Retrieves all records matching all supplied data points. In a case when no data points are supplied, all records are retrieved. The method returns null if no matches are found.

```
public abstract java.util.Vector
retrieveSegmentPessimistic(BusinessObjectCriteria firstCriteria,
BusinessObjectCriteria secondCriteria) throws java.lang.Exception
```

Code 9-10: retrieveSegmentPessimistic Method Signature

- **updatePointPessimistic**—Updates a record based on a unique value, derived from a combination of one or more primary keys. The record must have at least one primary key in order to be updated. The method unlocks the record after the operation is complete. In cases when the update or lock fails, and the transactional state is set to true, none of the changes are committed.

```
public abstract void updatePointPessimistic(java.lang.Object data)
throws java.lang.Exception
```

Code 9-11: updatePointPessimistic Method Signature

- **updateSetPessimistic**—Updates each element based on a unique value, derived from a combination of one or more primary keys. The record must have at least one primary key in order to be updated.

```
public abstract void updateSetPessimistic(java.util.Vector data) throws
java.lang.Exception
```

Code 9-12: updateSetPessimistic Method Signature

- **deletePointPessimistic**—Deletes the matching locked record. The method checks for a lock and, if it finds one, deletes and unlocks the record. When using the method, you must pass the primary key (or a unique value based on a combination of multiple primary keys), and the record must have been previously locked.

```
public abstract void deletePointPessimistic(java.lang.Object data)
throws java.lang.Exception
```

Code 9-13: deletePointPessimistic Method Signature

- **deleteSetPessimistic**—Deletes all locked records within a set. The method checks for a lock and, if it finds one, deletes and unlocks the records. When using the method, you must pass the primary key (or a unique value based on a combination of multiple primary keys), and the record must have been previously locked.

  If one record within the set fails due to locking issues, none of the records are deleted. Alternatively, within a localized transaction at the database level, the records before the failure are deleted, but the others are left untouched.

```
public abstract void deleteSetPessimistic(java.util.Vector data) throws
java.lang.Exception
```

Code 9-14: deleteSetPessimistic Method Signature

Examples of Optimistic and Pessimistic Locking

This section provides a series of examples of how to use the optimistic and pessimistic locking API, and includes the following samples:

- "Using the deletePointOptimistic Method" on page 206
- "Using the updateSetOptimistic Method" on page 206
- "Using the retrieveRayPessimistic Method" on page 207
- "Using the updatePointPessimistic Method" on page 207

Using the deletePointOptimistic Method

You must use a lock token when using optimistic locking to delete or update records in a table. If the lock token matches the one in the database, the operation is completed; otherwise an exception is raised.

Code Sample 9-15 illustrates how to use the deletePointOptimistic method to delete a record from a table.

```
Person myPerson = new Person();
myPerson.setId("1234567890");
myPerson.setLockToken("theCurrentLockToken");
DataAccess myDA =
    resourceManager.getDataAccessForName(myPerson.CLASS_NAME +"DataAccess")
myDA.deletePointOptimistic(myPerson);
```

Code 9-15: Using the deletePointOptimistic Method

Using the updateSetOptimistic Method

This example updates the "rank" of a set of people to gold using the updateSetOptimistic method, while ensuring that the changes do not conflict with any other updates. Note that the updateSetOptimistic method throws an UnexpectedMultipleRecordsException if an element was not updated, or if multiple records were about to be updated but were not.

Code Sample 9-16 illustrates how to use the updateSetOptimistic method to update a set of records in a table.

```
DataAccess myDA =
    resourceManager.getDataAccessForName(myPerson.CLASS_NAME + "DataAccess");
Vector people = new Vector();

Person my1Person = new Person();
my1Person.setId("1234567890");
Person my2Person = new Person();
my2Person.setId("0987654321");
Person my3Person = new Person();
my3Person.setId("6543210987");
people.add(my1Person);
people.add(my2Person);
people.add(my3Person);
//retrieve the current records.
people = myDA.retrieveSet(people)

for (int index = 0; index < people.size(); index++) {
    Person aPerson = (Person)people.get(index);
    if (aPerson.equalsIgnoreCase("regular") {
        aPerson.setRank("gold");
    }
}

myDA.updateSetOptimistic(theParameters);
```

Code 9-16: Using the updateSetOptimistic Method

Using the retrieveRayPessimistic Method

Code Sample 9-17 illustrates retrieving all people from the table that have an alphanumeric last name greater than “Smith” using the retrieveRayPessimistic method.

```
PersonCriteria myPerson = new PersonCriteria();
myPerson.setLastName("Smith");
myPerson.setLastNameCriteria(BusinessObjectCriteria.CRITERIA_GREATER);
DataAccess myDA =
    resourceManager.getDataAccessForName(myPerson.CLASS_NAME + “DataAccess");

try {
    Vector returnPersons = myDA.retrieveRayPessimistic(myPerson);
}
catch(Exception ex) {
    // Contend with lock issues
}
```

Code 9-17: Using the retrieveRayPessimistic Method

Using the updatePointPessimistic Method

Code Sample 9-18 illustrates updating a single person by ID exclusively using the updatePointPessimistic method.

```
Person myPerson = new Person();
myPerson.setId("1234567890");
DataAccess myDA =
    resourceManager.getDataAccessForName(myPerson.CLASS_NAME +"DataAccess");

try {
    myPerson = myDA.retrievePointPessimistic(myPerson);
    myPerson.setLastName("Smith");
    myDA.updatePointPessimistic(myPerson);
}
catch(Exception ex) {
    // Contend with lock issues
}
```

Code 9-18: Using the updatePointPessimistic Method

Order By Interface

Using the Order By interface, you can instruct the Data Accessor to return data sorted by a specific column in either ascending or descending order. You use the Order By interface with retrieve operations that return results as a vector, such as retrieveRay or retrieveSegment.

When creating the request to the Data Access object within your business service, you specify the following information as parameters:

- The business object
- One or two criteria objects, if you need a range returned

Within the criteria object, you specify a criteria field for each attribute within the business object. The Data Accessor uses this information to generate the where clause to perform the data retrieval. For more information about Business Object Criteria, see "Business Object Criteria" on page 196.

You can also specify the following additional information when performing the retrieve operation:

- Whether the sort is to be returned ascending (asc) or descending (desc)
- The order for sorting when multiple columns are requested

The order by interface consists of the following methods:

- **retrieveRayOrdered**—Performs an ordered ray retrieval, which returns all records matching a given criteria, ordering the results as specified by the order by criteria.

```
public retrieveRayOrdered(BusinessObjectCriteria data, java.util.Vector order);
```

Code 9-19: retrieveRayOrdered Method Signature

- **retrieveSegmentOrdered**—Performs an ordered segment retrieval. The method returns null if no row is found.

```
public retrieveSegmentOrdered(BusinessObjectCriteria firstCriteria,
BusinessObjectCriteria secondCriteria, java.util.Vector order)
```

Code 9-20: retrieveSegmentOrdered Method Signature

Note that the final argument to the retrieveRayOrdered and retrieveSegmentOrdered methods is a Vector that enables you to specify multiple fields on which to order. For example, you could instruct the Data Accessor to return results ordered by last name, first name, and city by adding these three Business Object Criteria to the Vector passed to the retrieveRayOrdered and retrieveSegmentOrdered methods.

Code Sample 9-21 illustrates how to retrieve all people with the last name of "Smith," and ordered by first name, and then by middle initial.

```
java.util.Vector theReturn = new Vector();

PersonCriteria myPerson = new PersonCriteria();
myPerson.setLastName("Smith");
myPerson.setLastNameCriteria(BusinessObjectCriteria.CRITERIA_EQUAL);

PersonCriteria orderOne = new PersonCriteria();
orderOne.setFirstNameCriteria(BusinessObjectCriteria.ORDER_DESCENDING);
PersonCriteria orderTwo = new PersonCriteria();
 orderTwo.setMiddleInitialCriteria(BusinessObjectCriteria.ORDER_DESCENDING);
java.util.Vector orderBy = new java.util.Vector();
orderBy.add(orderOne);
orderBy.add(orderTwo);

try {
    theReturn = (java.util.Vector) dataAccess.retrieveRayOrdered(aBOC, orderBy);
}
catch (Exception ex) {
    System.out.println(ex);
}
return theReturn;
```

Code 9-21: Using the orderBy Interface

For more information about the order by interface, including further examples, see the Javadoc for Chordiant 5 Foundation Server.

Count Interface

You can use the count interface to determine the number of records that exist within a table matching a specific criteria. The count interface is optimized to count the number of records without having to retrieve the records themselves.

The count interface includes:

- **countPoint**—Counts all matching records, returning the number of records matching the values provided in the business object passed into the method. In cases when no data values are supplied to the call, the method returns a count of all records.

```
public abstract java.lang.Object countPoint(java.lang.Object data)
throws java.lang.Exception
```

Code 9-22: countPoint Method Signature

- **countRay**—Counts all rows matching the supplied criteria. In cases when no criteria data point is supplied to the call, the method returns a count of all records.

```
public abstract java.lang.Object countRay(BusinessObjectCriteria data)
throws java.lang.Exception
```

Code 9-23: countRay Method Signature

- **countSegment**—Counts all records that match all provided data points. In case when no data points are supplied to the call, the method returns a count of all records. The countSegment method does not perform checks on the criteria bounds.

```
public abstract java.lang.Object countSegment(BusinessObjectCriteria
firstCriteria, BusinessObjectCriteria secondCriteria) throws
java.lang.Exception
```

Code 9-24: countSegment Method Signature

Code Sample 9-25 illustrates how to count all persons with a last name "Smith."

```
Integer numberOf = null;
Person myPerson = new Person();
myPerson.setLastName("Smith");

DataAccess myDA = resourceManager.getDataAccessForName(
    myPerson.CLASS_NAME + "DataAccess");
numberOf = (Integer) myDA.countPoint(myPerson);
```

Code 9-25: Using the Count Interface

For more information about the count interface, including enhanced examples, refer to the Javadoc for the data access class.

Performing Transactions

Transactions enable you to commit multiple operations across multiple processes. This section describes how to use the J2EE transaction interface to perform distributed transactions.

There are two types of transactions that you can perform with the JX architecture:

- Bean Managed Transactions (BMT), which are handled manually within the Java code of the EJB using the J2EE UserTransaction interface.

  When using Bean Managed Transactions with Chordiant 5 Foundation Server, you must not use nested transactions. Instead, you must join an existing transaction, if available.

- Container Managed Transactions (CMT), which are not handled in the code, but are handled through the J2EE container.

  Container Managed Transactions can be nested within an application server.

For a complete description of the two types of transactions, refer to "Transactions with the JX EJB" on page 20.

Creating Bean Managed Transactions

To perform container managed transactions (CMTs), refer to "Performing Container Managed Transactions" on page 215.

The J2EE UserTransaction interface used by Foundation Server comprises these methods:

- **getStatus** — Enables you to determine the status of the transaction. You can use this status to determine whether there is a currently active transaction to join, or whether your process must begin a new transaction.
- **begin** — Enables you to begin a new transaction.
- **commit** — Enables you to commit an existing transaction. Only the process that began the transaction should perform the commit operation.
- **rollback** — Enables you to rollback a transaction in case of an error. As with the commit method, only the process that began the transaction should perform a rollback operation.

To use bean managed transactions (BMTs) within your application:

1. Define a user transaction object, and get the current instance of the transaction.

   Code Sample 9-26 shows how you can retrieve the current instance of a transaction using the Java EJB session context.

```
javax.transaction.UserTransaction myTransaction = null;
myTransaction = getSessionContext().getUserTransaction();
```

Code 9-26: Retrieving a Transaction Instance

2. Verify that the current instance of the transaction is valid.

   You can do this by checking whether the current transaction is null. If so, you can throw an exception. Otherwise, you can proceed with the transaction processing.

   Code Sample 9-27 shows how you can check whether a transaction is valid.

```
if (myTransaction != null) {
    // Process the transaction
    . . .
    }
else {
    Exception ex = new BusinessServiceException(
        "Unable to obtain javax.transaction.UserTransaction");
    LogHelper.error(PACKAGE_NAME, CLASS_NAME,
        METHOD_NAME, "Unable to establish a user transaction", ex);
        throw ex;
}
```

Code 9-27: Checking if Transaction is Valid

3. Check if the transaction is already active.

   Foundation Server requires applications to join an existing transaction, if available, instead of beginning a new transaction. If a transaction is already active, you can stop processing the transaction within your code since the process that began the transaction is always responsible for completing the transaction.

   Code Sample 9-28 shows how you can check whether a transaction is already active.

```
if (myTransaction.getStatus() == javax.transaction.Status.STATUS_ACTIVE) {
    myTransaction = null;
}
else {
    myTransaction.begin();
}
```

Code 9-28: Checking if Transaction is Active

   Note that if a transaction is already active, you can discard the reference to the transaction. Otherwise, you can begin a new transaction.

4. Perform your database operations, as appropriate.

5. Commit the transaction.

   You must only commit transactions that your code began. You can determine whether you are responsible for committing the transaction by checking your reference to the transaction, as illustrated in Code Sample 9-29.

```
if (myTransaction != null) {
    myTransaction.commit();
}
```

Code 9-29: Committing Transactions

6. If there is an error, catch the exception and perform a rollback of the transaction.

   Your application should only attempt a rollback if it was the process to begin the transaction. Also, be aware that the rollback itself might fail. Finally, be sure to throw the same exception before exiting to enable it to propagate through the system properly.

   Code Sample 9-30 shows how you can catch an exception and perform a rollback, if appropriate.

```
catch (Throwable e1) {
    if (myTransaction != null) {
        try {
            myTransaction.rollback();
        }
        catch (Throwable e2) {
            // Don't throw e2; throw e1 below.
        }
    }
    throw (Exception) e1;
}
```

Code 9-30: Catching Exceptions and Performing Transaction Rollbacks

7. Set the configuration files to show that this is a bean managed transaction, as shown in Code Sample 9-31.

```
<Section>MyService
    <Tag>classname
        <Value>com.chordiant.service.MyService</Value>
    </Tag>
    <Tag>ConnectionName
        <Value>EJBBMT</Value>
    </Tag>
</Section>
```

Code 9-31: Configuring as Bean Managed Transaction

   See "Transaction Control Mechanism" on page 119 or "Adding Components through Configuration" on page 101 for more information.

Code Sample 9-32 illustrates a method employing a transaction to perform persistent operations.

```
protected CEICMCase startNewCase(String username, String authentication, String description,
    String ctiConnectionIdentifier) throws Exception {
    final String METHOD_NAME = "startNewCase";
    LogHelper.methodEntry(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    CEICMCase newCase = null;
    javax.transaction.UserTransaction myTransaction = null;

    try {
        myTransaction = getSessionContext().getUserTransaction();
        if (myTransaction != null) {
            if (myTransaction.getStatus() == javax.transaction.Status.STATUS_ACTIVE) {
                myTransaction = null;
            }
            else {
                myTransaction.begin();
            }

            // Create new case to write to database.
            newCase = new CEICMCase();
            newCase.setCaseNumber(caseNumberGen.getItemNumber());
            newCase.setDescription(description);
            newCase.setCtiConnectionIdentifier(ctiConnectionIdentifier);
            newCase.setLockFlag("1");

            // Write new case to DB; call create on DAO.
            newCase = (CEICMCase) ((CEICMCaseDataAccess)
            myDAResourceManager.getDataAccessForName(
                "CEICMCaseDataAccess")).createPoint(newCase);

            if (myTransaction != null) {
                myTransaction.commit();
            }
        }
        else {
            Exception ex = new BusinessServiceException(
                "Unable to obtain javax.transaction.UserTransaction");
            LogHelper.error(PACKAGE_NAME, CLASS_NAME, METHOD_NAME,
                "Unable to establish a user transaction", ex);
            throw ex;
        }
    }
    catch (Throwable e1) {
        LogHelper.error(PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "Exception occurred", e1);
        if (myTransaction != null) {
            try {
                myTransaction.rollback();
            }
            catch (Throwable e2) {
                LogHelper.error( PACKAGE_NAME, CLASS_NAME, METHOD_NAME,
                    "Exception occurred trying to rollback a transaction",
                    e2);
```

Code 9-32: Performing Persistent Operations in a Transaction

```
                // Don't throw e2; throw e1 below.
            }
        }
        throw (Exception) e1;
    }

    LogHelper.methodExit(PACKAGE_NAME, CLASS_NAME, METHOD_NAME);

    return newCase;
}
```

Code 9-32: Performing Persistent Operations in a Transaction (Continued)

Performing Container Managed Transactions

To perform a container managed transaction (CMT), specify that the service is to run as a Required CMT within the configuration file where the service is specified. Code Sample 9-33 provides an example of the configuration file code.

```
<Section>MyService
    <Tag>classname
        <Value>com.chordiant.service.MyService</Value>
    </Tag>
    <Tag>ConnectionName
        <Value>EJBCMTRequired</Value>
    </Tag>
</Section>
```

Code 9-33: Specifying CMT within the Configuration File

From that point on, everything is handled by the container.

See "Transaction Control Mechanism" on page 119 or "Adding Components through Configuration" on page 101 for more information.

Performing Joins

Using joins enables you to retrieve information from multiple tables by relating unique values that appear in each of the tables. For example, Code Sample 9-34 shows how you can use SQL against a relational database to execute a statement to retrieve customer profile information using a unique, shared identifier.

```
Select * from customer, profile where customer.cust_id = profile.profile_id;
```

Code 9-34: Using SQL to Retrieve Customer Profile Information

You can define similar views within Persistence Server by including the parameters for the join within your object model or within the CMI file.

To define a join within your object model using Chordiant's UML Extender for Rational Rose:

1. In your object model, open the class specifications.
2. In the Class Specifications window, select the JXP SQL tab.

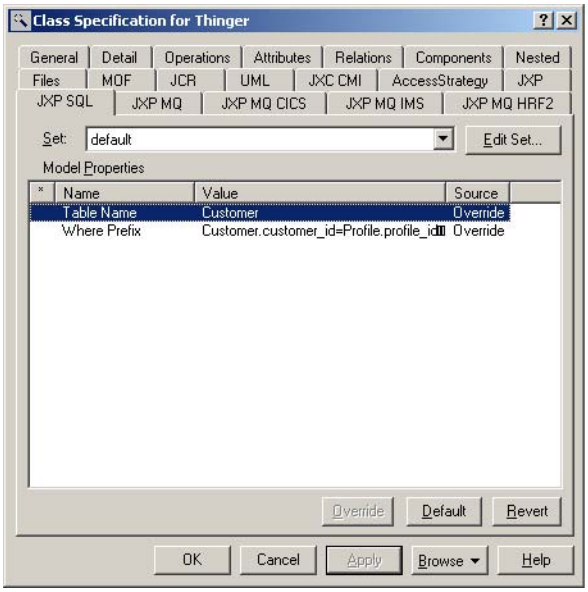


Figure 9-12: Specifying the Where Prefix within Rational Rose

3. Click in the Value column next to Where Prefix and type your specification for the join. You must also specify all table names, separated by commas.
4. Click **OK**.

To define a join within the CMI file:

1. Specify all the table names, separated by commas, as the rdbPhysical name for the containing class.
2. Define the view by including each of the attributes using table.column as the rdbPhysical name for each attribute.
3. Include the WherePrefix tag at the class level of the containing class.

   Use a full conditional statement, using the table.column notation, to specify the WherePrefix. For example, you could use this conditional to relate identifiers for the join:

```
<WherePrefix>Customer.customer_id=Profile.profile_id</WherePrefix>
```

Note: You can use AND elements to create extended where clauses when defining the join.

Code Sample 9-35 illustrates how to define a join within the CMI file to retrieve customer profile information.

```
<class>
    <name>CustProf</name>
    <parentClass>Object</parentClass>
    <DSN>testdb1ds</DSN>
    <rdbPhysicalName>Customer, Profile</rdbPhysicalName>
    <WherePrefix>Customer.customer_id=Profile.profile_id</WherePrefix>
    <persistentType>Oracle</persistentType>
    <LockStrategy>pessimistic</LockStrategy>
        <attribute>
            <name>cust_id</name>
            <javaType>java.lang.String</javaType>
            <multiplicity>1..1</multiplicity>
            <rdbPhysicalName>Customer.customer_id</rdbPhysicalName>
            <rdbLogicalType>VARCHAR</rdbLogicalType>
            <rdbSize>80</rdbSize>
            <rdbDigits>0</rdbDigits>
            <rdbNotNull>true</rdbNotNull>
            <rdbPrimaryKey>true</rdbPrimaryKey>
        </attribute>
        <attribute>
            <name>prof_id</name>
            <javaType>java.lang.String</javaType>
            <multiplicity>1..1</multiplicity>
            <rdbPhysicalName>Profile.profile_id</rdbPhysicalName>
            <rdbLogicalType>VARCHAR</rdbLogicalType>
            <rdbSize>80</rdbSize>
            <rdbDigits>0</rdbDigits>
            <rdbNotNull>true</rdbNotNull>
            <rdbPrimaryKey>true</rdbPrimaryKey>
        </attribute>
</class>
```

Code 9-35: Defining a Join within the CMI File

You can supply a clause within the WherePrefix tag since the system makes no assumptions concerning the relationship between tables, attribute names, or pattern of association. You can also use outer join syntax within the where prefix value.

Note: Traditionally, join operations can result in performance issues. This carries forward to join operations performed in the context of Chordiant Persistence Server, since the system does not implement any optimizations or syntax checking during class generation.

Not every method within the Data Access interface supports join operations. Specifically, only these methods support joins:

- countPoint
- countRay
- countSegment
- retrievePoint
- retrieveSet
- retrieveRay
- retrieveSegment
- retrieveRayOrdered
- retrieveSegmentOrdered

Methods that do not support joins throw an OperationNotSupported exception if you try to perform a join operation.

Note: If you do not relate tables with a unique value, multiple rows (a Cartesian product) will always be returned. Consequently, you cannot use any point methods in these cases.

CLOB Support

You can use CLOBs, or Character Large Objects, to store large strings in a record. Chordiant Persistence Server limits VARCHAR fields within a database to a maximum of 256 characters. CLOBs, on the other hand, enable you to store strings of indefinite size, as supported by the computing platform and associated drivers.

Note: CLOBs are only supported for Oracle and DB2 databases.

To enable CLOB support:

1. Define the attribute that holds the CLOB data to have a Java type of java.lang.String in the CMI file.

   ```
   <javaType>java.lang.String</javaType>
   ```

2. Specify the rdbLogicalType of the attribute to be CLOB.

   This activates logic within the XSL stylesheet logic to include the methods required to handle CLOB fields.

   ```
   <rdbLogicalType>CLOB</rdbLogicalType>
   ```

Code Sample 9-36 illustrates a sample attribute definition to support a CLOB.

```
<attribute>
   <name>Type</name>
   <javaType>java.lang.String</javaType>
   <multiplicity>1..1</multiplicity>
   <rdbPhysicalName>TYPE</rdbPhysicalName>
   <rdbLogicalType>CLOB</rdbLogicalType>
   <rdbNotNull>true</rdbNotNull>
   <rdbPrimaryKey>false</rdbPrimaryKey>
</attribute>
```

Code 9-36: Example CLOB Attribute Definition

When storing large character strings, CLOB attributes conform to the following rules:

- The Data Accessor methods updateRay and updateSegment do not update CLOB attribute values
- The retrieve methods do not use CLOB attributes to determine which records to retrieve
- The delete methods do not use CLOB attributes to determine which records to delete
- You must have a primary key field defined to insert CLOB data

THE RESOURCE MANAGER AND PERSISTENCE

The Resource Manager serves the following roles within the Data Accessor:

- Factory for Data Accessors, Business Objects, Business Object Behavior classes, and Business Object Criteria classes for services.
- Run-time context for the Data Access objects

Note: The Resource Manager uses the CMI file to build the factory.

The application server typically maintains a connection pool to the underlying data store to enable applications to gain more efficient access to information. The specific connection pool used is generally dependent on the application server employed within the environment.

The Resource Manager provides a layer of abstraction enabling Chordiant 5 Foundation Server applications to gain access to the connection pool without having to know the specifics of the application server architecture. During system initialization, the Resource Manager acquires a handle to the connection pool from the application server, thereby enabling it to service requests for connections from objects within the Chordiant 5 Foundation Server.

An application service can then use the Resource Manager to get an instantiated Data Access object, as illustrated in Figure 9-13. The numbered steps are described after the figure.

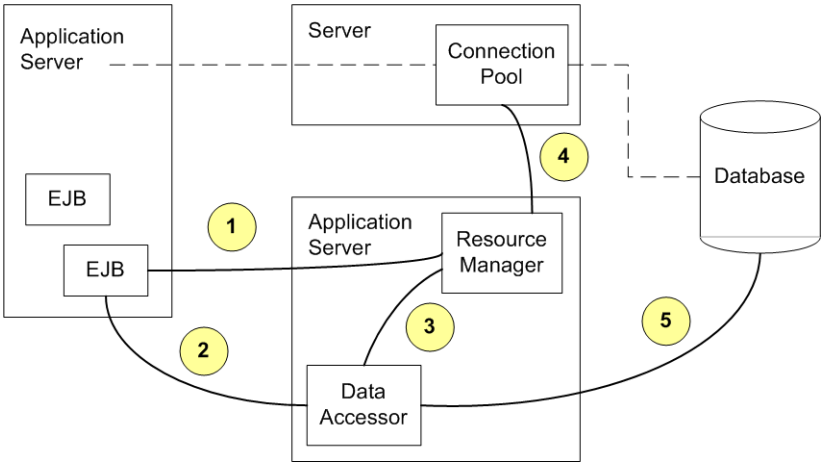


Figure 9-13: Resource Manager Overview

1. The application service requests a Data Accessor from the Resource Manager to perform a database operation.

   The Resource Manager instantiates a Data Access object. Note that the application service executes within an EJB.

2. The service uses the Data Accessor to perform the database operation.
3. The Data Accessor requests a connection from the Resource Manager.
4. The Resource Manager uses its handle to the connection pool, and requests a connection from the application server.
5. The Data Accessor performs the requested operation.

   After completing the database operation, the Data Accessor releases the connection to the database.

Figure 9-14 illustrates the involvement of the Resource Manager in the execution flow of an application performing a database operation.

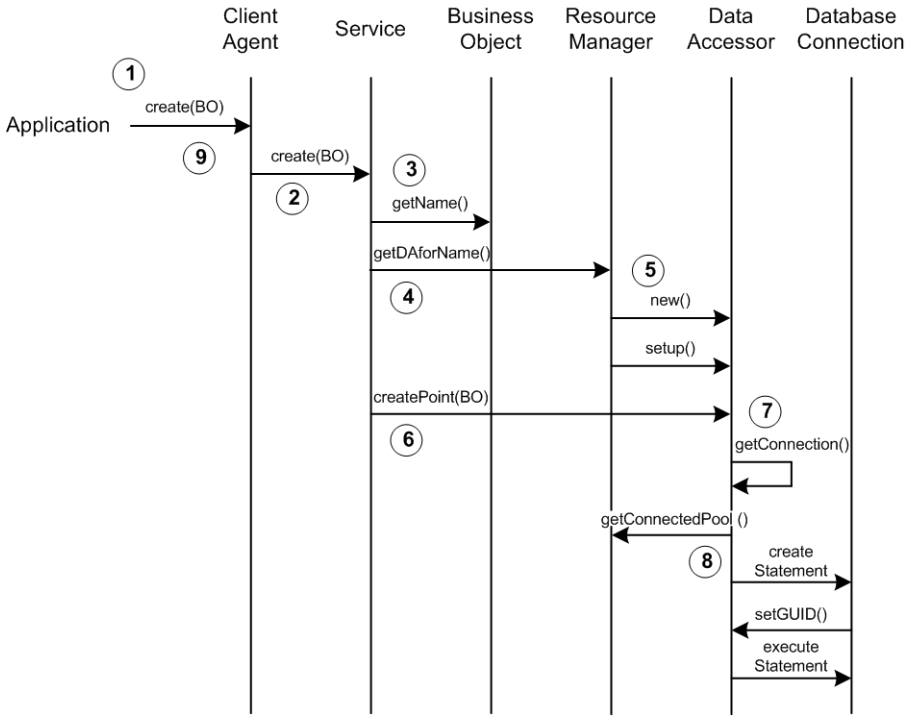


Figure 9-14: Resource Manager Execution Flow

These numbered steps describe the execution flow of an application performing a database operation, such as create, using the Resource Manager:

1. The application issues a call to the client agent, passing the Business Object.
2. The client agent calls the service.
3. The service accepts the Business Object, and attempts to locate the Data Accessor associated with the Business Object.
4. The service gets a Data Accessor for the Business Object from the Resource Manager.
5. The Resource Manager creates an instance of the Data Access object for the Business Object, and performs setup operations.
6. The service uses the Data Access object to perform the work it wants to complete.

   For example, the service might issue a createPoint call directly against the Data Accessor.
7. The Data Accessor gets a database connection from the connection pool.
8. The Data Accessor performs the requested operations.

   The Data Accessor returns a GUID, if appropriate.

9. The routines pass results back to the calling functions.

The Resource Manager uses the package and class names within the CMI file to establish the full path name for all data access objects, thereby enabling the Resource Manager to instantiate the Data Accessor for a calling service.

THE LOCK MANAGER

The Lock Manager is a service that provides object locking, unlocking, and lock status functionality for components within the Chordiant 5 Foundation Server. The Lock Manager locks objects using the object name and unique object identifier, such as the GUID. You can use the Lock Manager to gain an exclusive lock on an entity, such as a row in a database table, a screen, or other objects within the system.

You could use the Lock Manager to get an exclusive lock on a customer record, for example, thereby enforcing a business rule that enables only a single person to update information about a particular customer at any given time. Alternatively, you could use the Lock Manager to lock a maintenance screen, preventing multiple users from attempting to perform the same maintenance operation at the same time.

The Lock Manager is independent of any other component, enabling you to use the client agent of the Lock Manager from any component. The Data Accessor uses the Lock Manager, for example, to perform pessimistic locking operations. Optimistic locking, on the other hand, does not use the Lock Manager. Instead, optimistic locking uses a lock token and a lock column in the table.

Note: With pessimistic locking, all locking state information is stored in a database instead of being memory resident. This enables distributed locking without raising issues related to threading or resource contention. However, the application must properly handle orphaned lock records in a database caused by application or system exception.

Figure 9-15 provides an overview of the Lock Manager, and its relationship to other components with the system. The Lock Manager is typically used in the following ways:

A—Applications can lock and unlock objects directly, using the client interface to the Lock Manager, as described in "Client Interface to the Lock Manager" on page 224.

B—The Data Accessor can lock and unlock objects when applications perform pessimistic locking, using methods such as retrievePointPessimistic.

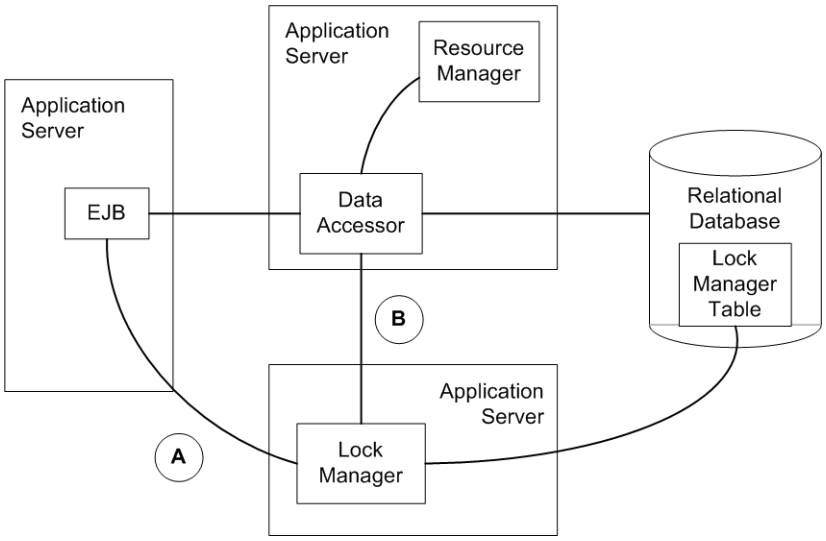


Figure 9-15: Overview of the Lock Manager

Note: The Lock Manager does not rely on the underlying database or data store capabilities to implement the locking implementation. This means that other processes, operating independent of the Lock Manager and accessing objects directly, can gain access to objects you have locked using the Lock Manager.

Configuring the Lock Manager

Like other components within Chordiant Foundation Server, the LockService is configured through an XML file, namely `lock.xml`. In addition to defining the location of the LockService, `lock.xml` enables you to define a lock timeout period, in milliseconds. By default, the timeout value is set to zero, which essentially disables the timeout mechanism. If you set the timeout value to be any integer greater than zero, locks will expire after that amount of time passes.

Client Interface to the Lock Manager

The system provides a client interface to the Lock Manager, enabling applications to perform basic functions such as the locking and unlocking objects, as well as checking the lock status of a specific object.

The client interface to the Lock Manager consists of the following methods:

- **lock**—Locks a single object using the object name and object GUID. An object can be locked if it has not already been locked or if its lock has already expired. The method returns without exception if the lock is successful, otherwise a LockUnavailableException is thrown in cases when the record is unable to be locked. There is no lock queueing for wait periods.

```
public void lock(java.lang.String userName, java.lang.String authenticationToken,
    com.chordiant.lock.classes.LockMultipleContainer data)
public void lock(java.lang.String userName, java.lang.String authenticationToken,
    java.lang.String BusinessObjectName, java.lang.String GUID)
```

Code 9-37: lock Method Signatures

- **unlock**—Unlocks a single object that was previously locked using the lock method. The method throws an exception in cases when the requested business object name and GUID are currently not locked.

```
public void unlock(java.lang.String userName, java.lang.String authenticationToken,
    com.chordiant.lock.classes.LockMultipleContainer data)
public void unlock(java.lang.String userName, java.lang.String authenticationToken,
    java.lang.String BusinessObjectName, java.lang.String GUID)
```

Code 9-38: unlock Method Signatures

The Lock Manager uses BMT when performing a lock operation. If the Lock Manager is used within a CMT operation, the application must do an unlock if the CMT operation fails. There is no automatic unlock, as it is in a separate transaction context from the CMT operation.

There is also a version of the unlock method that enables you to remove all locks older than a supplied age.

```
public void unlock(java.lang.String userName, java.lang.String authenticationToken,
    long age)
```

Code 9-39: unlock Method Signature for long age

- **check**—Checks whether an object is currently locked. Returns True if the object is locked and the lock has not expired.

```
public boolean check(java.lang.String userName, java.lang.String authenticationToken,
java.lang.String BusinessObjectName, java.lang.String GUID)
```

Code 9-40: check Method Signature

Note: The Lock Service runs inside of an EJB. The Data Accessor uses the Lock Service by issuing calls to the Lock Service client agent.

DATA TYPE SUPPORT

In mapping the business data to specifics within the database environment, the Database Specialist must consider the specific Java data types being mapped to columns that appear within tables in the database.

Table 9-4 outlines the mapping from the Java data types to the types used within the respective databases.

JAVA DATA TYPE	DATABASE TYPE
• java.lang.BigDecimal	• numeric(X,2)
• java.lang.Boolean	• char
• java.lang.Double	• numeric(X,2)
• java.lang.Integer	• numeric
• java.lang.Long	• numeric
• java.lang.Object	• File system file of object, converted to an XML string
• java.lang.String (maximum 256 characters)	• Varchar
• java.lang.String (more than 256 characters)	• CLOB (Oracle and DB2 only)
• java.util.Date	• Timestamp (DB2) • Datetime (Oracle)

Table 9-4: Mapping Java Data Types to Database Types

Note: You can only perform a mapping between the listed Java data types; no other data types are supported.

These are supported data types for persistence data type mapping. For supported communication data types, see "Supported Data Types" on page 141.

Understanding Object to File Support

Chordiant 5 Foundation Server serializes objects to an XML string, and stores the resulting string in a file using the primary key (typically the GUID) of the object as the file name, as illustrated in Figure 9-16.

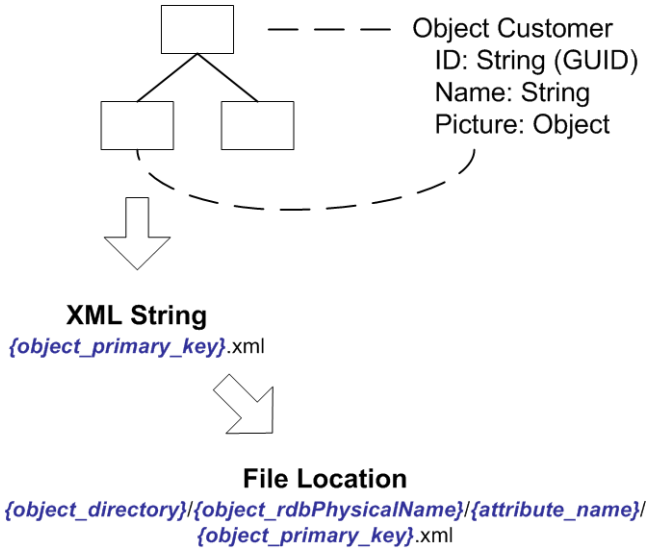


Figure 9-16: Mapping Objects to XML Strings

The system constructs the path leading to the file using these elements:

- **object_directory**—The root of the object directory, defined using the OBJECT_DIRECTORY parameter in component configuration file and usually in the `master.dtd` file. If you are working in a clustered environment, the object_directory should point to a shared disk. For example, Code Sample 9-41 shows the Generic Service's entry in its configuration file to define the root directory for the path.

```
    <Section>com.chordiant.bc.services.GenericService
...
    <Tag>ResourceName
        <Value>OBJECT_DIRECTORY</Value>
    </Tag>
</Section>
...
<Section>com.chordiant.bc.services.GenericService.OBJECT_DIRECTORY
    <Tag>ResourceType
        <Value>STRING</Value>
    </Tag>
    <Tag>ResourceValue
        <Value>&JXB_BINARY_DATA_STORAGE_ROOT_DIRECTORY;/bindata</Value>
    </Tag>
</Section>
```

Code 9-41: Generic Service Configuration File Showing Root Directory

  Documents are not actually stored in the database. This enables more performant queries.

- **object_rdbPhysicalName**—The rdbPhysicalName of the object, as specified in the CMI file as class-level metadata. For example, you could use Code Sample 9-42 in the CMI file to define the rdbPhysicalName for the object in Figure 9-16.

```
<class>
    <name>Customer</name>
    <parentClass>Object</parentClass>
    <DSN>customerdblds</DSN>
    <rdbPhysicalName>customer</rdbPhysicalName>
    <persistentType>Oracle</persistentType>
    <LockStrategy>pessimistic</LockStrategy>
    ...
</class>
```

Code 9-42: Defining the rdbPhysicalName for the Object in the CMI File

- **attribute_name**—The rdbPhysicalName of the attribute (data member) in the object, as specified in the CMI file as attribute-level metadata. For example, you could use Code Sample 9-43 in the CMI file to define the rdbPhysicalName for the attribute in Figure 9-16 on page 226.

```
<attribute>
    <name>Type</name>
    <javaType>java.lang.String</javaType>
    <multiplicity>1..1</multiplicity>
    <rdbPhysicalName>TYPE</rdbPhysicalName>
    ...
</attribute>
```

Code 9-43: Defining the rdbPhysicalName for the Attribute in the CMI File

Note: Application developers do not need to know about the mapping of Java data types to database entities. Instead, the application developer simply codes to the Java interface generated by the Database Specialist, as described in "Persistence Server Process Flow" on page 185.

CONFIGURING WEBSPHERE MQ PERSISTENCE

Before using data available on WebSphere MQ data stores, you must configure a series of parameters in the `jxpmq.xml` configuration file to define the interaction between Chordiant 5 Foundation Server and WebSphere MQ.

This section describes parameters related to the following aspects of configuring WebSphere MQ persistence:

- Connection Pool
- Connection
- Runtime

Table 9-5 outlines the WebSphere MQ Connection Pool parameters. You should include the Connection Pool parameters in the MQConnectionPoolConfiguration section of the configuration file.

PARAMETER	DESCRIPTION
RESOURCE_TAG_FOR_ DESTROY_CONNECTIONS_ TIMEOUT	Time in milliseconds until the Connection Pool destroys unused connections.
RESOURCE_TAG_FOR_MAX_ UNUSED_CONNECTIONS	Maximum number of unused connections.

Table 9-5: WebSphere MQ Connection Pool Parameters

Code Sample 9-44 illustrates a sample section of the configuration file containing the MQ Connection Pool parameters.

```
<Section>
   MQConnectionPoolConfiguration
   <Tag>
      RESOURCE_TAG_FOR_DESTROY_CONNECTIONS_TIMEOUT
      <Value>3600000</Value>
   </Tag>
   <Tag>
      RESOURCE_TAG_FOR_MAX_UNUSED_CONNECTIONS
      <Value>10</Value>
   </Tag>
</Section>
```

Code 9-44: Sample WebSphere MQ Connection Pool Configuration

Table 9-6 outlines the WebSphere MQ Connection parameters. You should include the Connection parameters in a user-defined section specified by the DSN parameter in the CMI file.

PARAMETER	DESCRIPTION
RESOURCE_TAG_FOR_MQ_ QMGR	Name of the Queue Manager to which to connect.
RESOURCE_TAG_FOR_MQ_ PUTQ	Name of the Put Queue to open and to which to send requests.
RESOURCE_TAG_FOR_MQ_ PUTQMGR	(Optional) Name of the remote Queue Manager.
RESOURCE_TAG_FOR_MQ_ GETQ	Name of the Model Queue to use to generate a Dynamic Queue.
RESOURCE_TAG_FOR_MQ_ CONNECTIONTYPE	CLIENT requires a host name and a channel. The valid values are CLIENT or SERVER.
RESOURCE_TAG_FOR_MQ_ HOSTNAME	Host name where the WebSphere MQ Queue Manager is running.
RESOURCE_TAG_FOR_MQ_ CHANNEL	Name of the Client/Server channel to connect as a client.
RESOURCE_TAG_FOR_MQ_ PORT	A positive integer port number to which WebSphere MQ is listening. Zero (0) is the default setting for this parameter. You should use this parameter if MQ is using a port number other than the default port number of 1414.
RESOURCE_TAG_FOR_MQ_ USERID	The userID used to connect to WebSphere MQ.
RESOURCE_TAG_FOR_MQ_ PASSWORD	The password used to connect to WebSphere MQ.

Table 9-6: WebSphere MQ Connection Parameters

Table 9-7 outlines the WebSphere MQ Runtime parameters. You should include the Runtime parameters in a user-defined section specified by the DSN parameter in the CMI file.

PARAMETER	DESCRIPTION
RESOURCE_TAG_FOR_MQ_ WAITINTERVAL	Number of milliseconds MQ waits for a response.
RESOURCE_TAG_FOR_MQ_ EXCEPTION_ALLOWED	Whether the Data Accessor handles data exceptions for each attribute. The valid values for this parameter are TRUE or FALSE.

Table 9-7: WebSphere MQ Runtime Parameters

Code Sample 9-45 illustrates a sample section of the configuration file containing the WebSphere MQ Connection and Runtime parameters.

```
<Section>
    mq_test_connection
    <Tag>
        RESOURCE_TAG_FOR_MQ_QMGR
        <Value>MQJH</Value>
    </Tag>
    <Tag>
        RESOURCE_TAG_FOR_MQ_PUTQ
        <Value>MQTHINGER</Value>
    </Tag>
    <Tag>
        RESOURCE_TAG_FOR_MQ_PUTQMGR
        <Value>MQJH</Value>
    </Tag>
    <Tag>
        RESOURCE_TAG_FOR_MQ_GETQ
        <Value>MQTHINGER.REPLY</Value>
    </Tag>
    <Tag>
        RESOURCE_TAG_FOR_MQ_WAITINTERVAL
        <Value>300000</Value>
    </Tag>
    <Tag>
        RESOURCE_TAG_FOR_MQ_CHANNEL
        <Value>JH1</Value>
    </Tag>
    <Tag>
        RESOURCE_TAG_FOR_MQ_HOSTNAME
        <Value>localhost</Value>
    </Tag>
    <Tag>
        RESOURCE_TAG_FOR_MQ_PORT
        <Value>0</Value>
    </Tag>
    <Tag>
        RESOURCE_TAG_FOR_MQ_USERID
        <Value>None</Value>
    </Tag>
    <Tag>
        RESOURCE_TAG_FOR_MQ_PASSWORD
        <Value>None</Value>
    </Tag>
    <Tag>
        RESOURCE_TAG_FOR_MQ_CONNECTIONTYPE
        <Value>SERVER</Value>
    </Tag>
    <Tag>
        RESOURCE_TAG_FOR_MQ_EXCEPTION_ALLOWED
        <Value>TRUE</Value>
    </Tag>
</Section>
```

Code 9-45: Sample WebSphere MQ Connection and Runtime Parameters in Configuration File

EXAMPLE OF USING PERSISTENCE SERVER

This section provides instructions for adding persistence to your service.

To add persistence to your application:

1. Define the service that will be performing the persistence operations. Follow the instructions in "Building a Service" on page 112 or create one using the UML Extender for Rational Rose. For details, refer to the *Chordiant 5 Foundation Server Application Components Developer's Guide*.

2. Be sure that you import any persistence business services that you want to use for your project. These will not necessarily be in the com.chordiant package.

```
import com.choriant.lock.client.LockClientAgent;
import com.choriant.persistence.businesscriteria.BusinessObjectCriteria;
import com.choriant.persistence.DataAccess;

import.java.sql.*;
import.java.util.Vector;
import java.sql.DataSource;

import com.choriant.service.*;
```

3. In your service, create a new attribute to reference the Resource Manager, implementing the BusinessObjectResourceInterface. Initialize the value to null.

```
protected BusinessObjectResourceInterface myDAResourceManager = null;
```

   For background information on the Resource Manager, refer to "Chordiant Resource Manager" on page 170.

4. Instantiate the Resource Manager when your service starts up. The commonSetup method can be used within a service to set up the Resource Manager. Here is the commonSetup method for the BusinessObjectResourceManager.

```
public BusinessObjectResourceInterface commonSetup(
    String userName, String authentication, String packageName,
    String sectionName, Service service)
        throws Exception
{
    final String METHOD_NAME = "commonSetup";
    try
    {
        // Give the login name and token of this service to the resource mgr.
        // so the DA can call the lock manager as a proxy to this service.

        // Login name and authentication are used when the resource manager or
        // the data access objects need to call a client agent.  (LockClientAgent,
        // SeedClientAgent, etc).
        // All operations done within the scope of a service use the service's
        // login name and authentication.

        putResource(TAG_USER_NAME, userName);

        putResource(TAG_SECURITY_TOKEN, authentication);

        // Get and place a lock client agent on the resource manager.
        // This is used during pessimistic locking.
        putResource("LockClientAgent", (LockClientAgent)
            ClientAgentHelper.getClientAgent(LockClientAgent.CLASS_NAME));

        if ( packageName != null )
        {
            if ( ! xmlConfigSectionDone )
            {
                getConfiguration( packageName, sectionName );
            }
            setup( service );

            setUpdateNull( true );
        }
    }
    catch (Throwable e1)
    {
        LogHelper.error(PACKAGE_NAME, CLASS_NAME, METHOD_NAME, "Exception
            occurred", e1);
        throw (Exception) e1;

    }
    return this;
}
```

The commonSetup method performs these tasks:

- Calls putResource to add this service's name and authentication to the Resource Manager so it can be used to call other client agents.
- Gets an instance of lockClientAgent, for use in obtaining client agents.
- If the package name is not null, gets the configuration information for this service from the relevant configuration XML files.

— Calls setup(service) to get a J2EE session context for this service and set up the object lookup table.

— Sets setUpdateNull to true, making the service accept null values for database update. This is required for Chordiant-provided business services. You can set this value to be false, which tells the Data Accessor to bypass null attributes during a database update operation.

5. When you need to access a data store, from your service, call the Resource Manager to get the Data Accessor, Business Object, and Business Object Criteria classes.

   You must get the business object by using the class name (but not the package name) of the object.

```
PartyRelationship nullPartyRelationship = (PartyRelationship)
    myResourceManager.getBusinessObjectForName(NullPartyRelationship.CLASS_NAME);
```

Code 9-46: Using the getBusinessObjectForName Method

   You can get the Data Accessor or Business Object Criteria classes forName.

```
Vector allCommonObjectRoles = (Vector) myResourceManager.getDataAccessForName(
    PartyRoleTableDataAccess.CLASS_NAME).retrieveRay(prtBOC);
```

Code 9-47: Using the getBusinessObjectForName Method

```
PartyRelationshipViewTableCriteria)
    myResourceManager.getBusinessObjectCriteriaForName(
    PartyRelationshipViewTableCriteria.CLASS_NAME);
```

Code 9-48: Using the getBusinessObjectCriteriaForName Method

6. Once you have the Data Accessor, Business Object, and Business Object Criteria classes, use the methods on the Data Accessor to access the data store. The methods you can use are described in "Data Access Methods" on page 192.

Chordiant Persistence Server and XSL Stylesheets

Chordiant Persistence Server offers support for these data accessors:

- SQL (for both Oracle and DB2)
- WebSphere MQ (for both text and XML messages)

You can customize the data accessors using XSL stylesheets. Chordiant supplies a set of stylesheets that you can copy and customize to suit your requirements. This section offers tips for using the XSL stylesheets. For more information on XSL, consult a resource dedicated to this topic.

XSL Header

All XSL files start with a header, as illustrated in Code Sample 9-49.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
```

Code 9-49: XSL File Header

XSL Options and Begin Statement

The start of every XSL file includes the information shown in Code Sample 9-50.

```
<xsl:output method="text" indent="yes" omit-xml-declaration="yes"
doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN" />
<xsl:template match="root">
</xsl:template>
</xsl:stylesheet>
```

Code 9-50: Standard XSL File Starting Information

The <xsl:template match=> tag can specify an arbitrary value. Chordiant uses the value root as the root tag within the XML document (CMI). Since this will generate Java code, omit-xml-declaration is set to yes to prevent the output from having an XML header.

Value-of Tag

The value-of tag retrieves the value of the DSN field.

```
<xsl:value-of select="DSN"/>
```

Sort

The sort tag orders the selected data set from table specified using rdbPhysicalName. The following illustrates how to specify sorting in descending order.

```
<xsl:sort select="rdbPhysicalName" order="descending"/>
```

If - Then

You can use the if-then tag to verify that rdbPhysicalName has a value.

```
<xsl:if test="rdbPhysicalName!=''">
</xsl:if>
```

Legal Filter Operators

Table 9-8 lists legal filter operators.

Filter Operator	Description
=	equal
=!	not equal
&LT&	less than
&GT&	greater than

Table 9-8: Legal Filter Operators

For

The for tag enables you to specifying looping operations. Code Sample 9-51 loops through all attribute nodes with multiplicity equal to 1..1.

```
<xsl:for-each select="attribute[multiplicity='1..1']">
</xsl:for-each>
```

Code 9-51: Using the For Tag

Choose (Case/Switch)

The choose tag enables you to encode a logical switch. In Code Sample 9-52, the String is normalized when the value is a String.

```
<xsl:choose>
   <xsl:when test="javaType='java.lang.String'">
      firstStatementBuffer.append(normalize(data.get<xsl:value-of
         select="name"/>()));
   </xsl:when>
   <xsl:otherwise>
      firstStatementBuffer.append(data.get<xsl:value-of
         select="name"/>());
   </xsl:otherwise>
</xsl:choose>
```

Code 9-52: Using the Choose Tag

Hold Value - “element_name”

Code Sample 9-53 illustrates how to declare the local attribute stylesheetversion with a value of Style Sheet Version: common 11152001:16:160. Each declaration creates a new local value within the given scope.

```
<xsl:variable name="stylesheetversion">* Style Sheet Version:  common
11152001:16:160</xsl:variable>
```

Code 9-53: Using the Hold Value Tag

XSL Limitation

Be aware that unlike many programming systems, XSL does not keep variables outside of the local scope. This means that when a named variable is defined at a high level within the code, even though the value of that variable is changed within a code block (such as an if block or a for block), once you exit the block, the variable again assumes the value defined at the higher level.

This eliminates the possibility of using variables as counters or sentinel flags.

Code Sample 9-54 illustrates this point.

```
class example {
    protected String myVariable = "blue";

    public void testFunction(){
        //Variable change occurs here
        String myVariable = "green";
        //For the rest of this function, myVariable equals "green",
        //when you get to any other function, even if it is called from
        //this function, the variable "myVariable" will equal "blue",
        //not "green";

        if (true)
        {
            myVariable = "purple";
        }

        //myVariable == "green" -- this is different than other
        //languages, where the "purple" would have persisted

    } //end function testFunction

}
```

Code 9-54: Example of XSL Limitation

The following illustrates how to place the value of the local attribute stylesheetversion.

```
<xsl:value-of select="$stylesheetversion"/>
```

Trim Value

You can trim a string to a fixed number of characters. The following statement trims the selected value to its first 27 characters.

```
<xsl:value-of select="substring(name,1,27)"/>
```

Call Procedure

You can perform procedure calls using XSL. Code Sample 9-55 illustrates how to call the CommonBlock template by name, passing in as parameters the local variables class_name and table_name.

```
<xsl:call-template name="CommonBlock">
   <xsl:with-param name="class_name" select="$class_name">
   </xsl:with-param>
   <xsl:with-param name="table_name" select="$table_name">
   </xsl:with-param>
</xsl:call-template>
```

Code 9-55: Using the CommonBlock Template

Template Header

Code Sample 9-56 illustrates how to declare a CommonBlock template with two parameters. The template uses the value of the class_name parameter, which is the same as the use of a local variable (see "Hold Value - "element_name"" on page 235).

```
<xsl:template name="CommonBlock">
   <xsl:param name="class_name"/>
   <xsl:param name="table_name"/>
   .
   .
   .
<xsl:value-of select="$class_name"/>
   .
   .
   .
</xsl:template>
```

Code 9-56: Declaring a CommonBlock Template

Import Statement

Code Sample 9-57 illustrates how to import the ResultSetToObject.xsl stylesheet:

```
<xsl:import
href="file:///E:/Chordiant/DevEnv/jxp/da/oracle/ResultSetToObject.xsl"/>
```

Code 9-57: Using the Import Statement

Escape from XSL Grammar

You can add information that is not specified in the XSL grammar. When you do, the information inside these tags are decoded as XSL. This enables you to output XSL code tags without the tags being interpreted.

This also enables you to add a semicolon to the end of a line of Java code without XSL introducing a carriage return, as illustrated in Code Sample 9-58.

```
<xsl:text/>;<xsl:text/>
```

Code 9-58: Adding Non-XSL Information

Additionally, you can include greater than and less than symbols in your intended Java code by using the following combinations:

- < <
- > >

Failing to do so will result in an XSL error.

Chapter 10

Chordiant Event Server

The Chordiant Event Server provides a rich set of asynchronous XML messaging capabilities leveraging the industry-standard Java Messaging Service (JMS), enabling configurable inbound access to all Chordiant services. Additionally, outbound messages generated by Chordiant services can be handled and passed to JMS.

The Foundation Server asynchronous messaging capabilities enable you to create loosely-coupled distributed applications and services.

If you will be using the Chordiant Event Server, you must configure it during installation of Chordiant 5 Foundation Server. Refer to the *Chordiant 5 Tools Platform Installation and Configuration Guide* for details. The Message Driven Beans (MDB) used for inbound message processing are configured through the Application Packaging Manager. Refer to the *Chordiant 5 Applications Deployment Guide* for details.

Event Server Components

The Chordiant Event Server includes the following components:

- OutboundMessageHelper — Enables messages to be directed to specific JMS queues and topics based on calls to specific Chordiant service calls. Depending upon calls to specific Chordiant services, the OutboundMessageHelper listens for specific Chordiant service requests or responses, and then maps the requests and responses to JMS topics or queues based on filters, implemented as MessageDispatcher objects. The OutboundMessageHelper maintains a list of the available MessageDispatcher objects.
- MessageDispatcher — Sends specific Chordiant service requests or responses to JMS topics or queues. Each MessageDispatcher object must implement the MessageDispatcher interface. Chordiant supplies a reference implementation, the XMLMessageDispatcher. The XMLMessageDispatcher can be extended to transform the data or direct it to other destinations based on your own logic and behavior.

  After you create a new MessageDispatcher class, you can configure the OutboundMessageHelper to load the customized MessageDispatcher class.
- Message Driven Bean — Enables inbound messages from JMS to be directed to specific Chordiant JX services. MDBs act as JMS message listeners. Based on its configuration, an MDB gets messages from a queue or topic and maps the messages to Chordiant services.

- MessageFilter — Provides MessageHandlers to the Message Driven Bean. The MessageFilter looks in the configuration file to see which MessageHandler is appropriate for the received message. Chordiant provides a DefaultMessageFilter, but you can also extend it to suit your needs. Customized MessageFilters must implement the IMessageFilter interface.
- MessageHandler — Sends inbound requests to specific Chordiant services, handling responses as appropriate. Each MessageHandler object must implement the IMessageHandler interface. Chordiant supplies two default implementations:
  - TextMessageHandler: Handles an XML string as a text message. This string includes user name, password, and service call information. This is the parent class for XMLMessageHandler.
  - XMLMessageHandler: Handles Chordiant's payload data. For details on payload data, refer to "Passing Payload with PayloadData" on page 140.

  You can use the XMLMessageHandler as is or copy and modify it to implement custom behavior. You can choose to have handlers for different types of messages, including ByteMessage, StreamMessage, ObjectMessage, and MapMessage.

Figure 10-1 illustrates the architecture and component interactions of the Foundation Server Asynchronous Messaging Architecture.

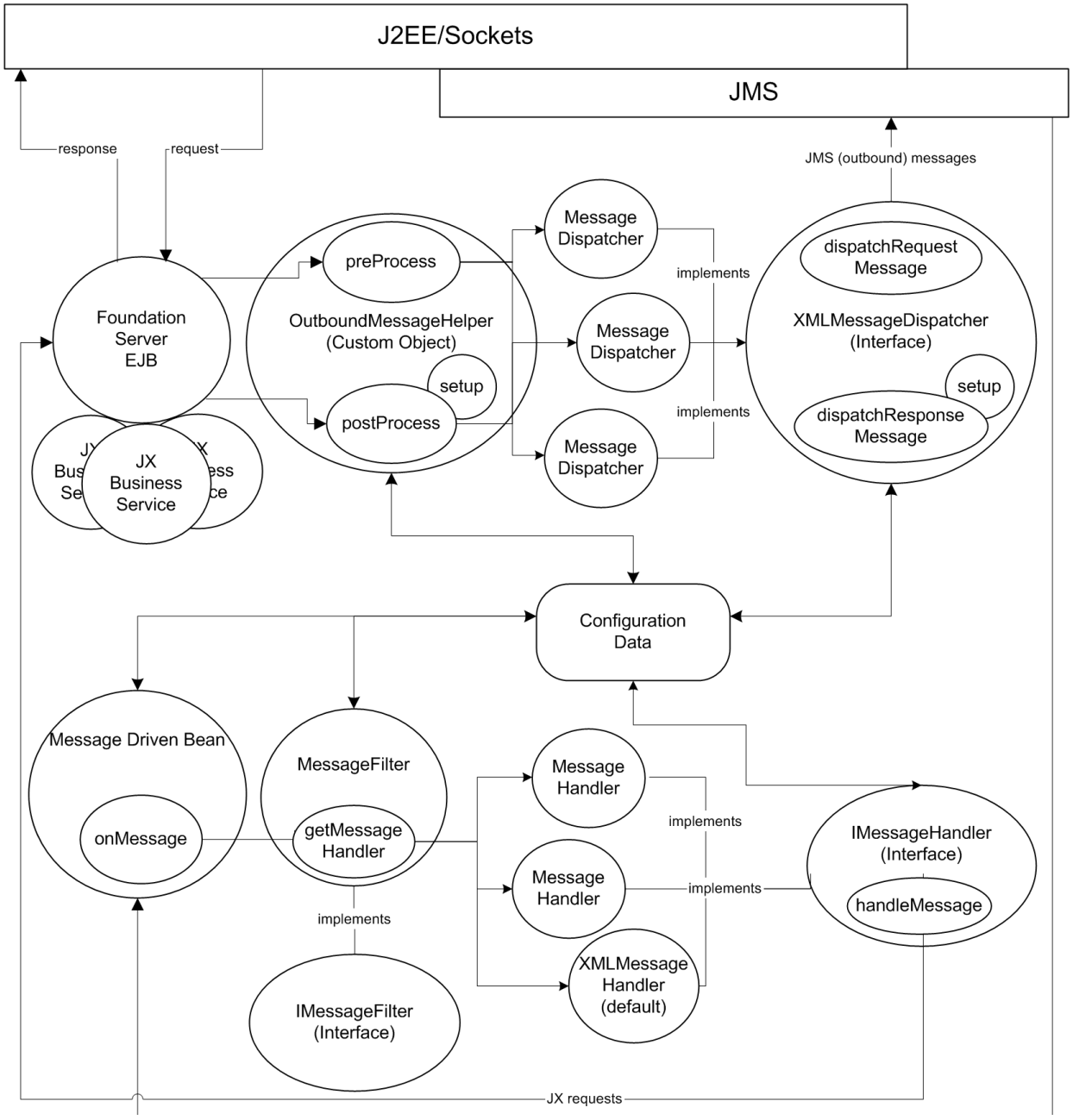


Figure 10-1: Foundation Server Asynchronous Messaging Architecture

The OutboundMessageHelper class is implemented as a CustomObject, and is located in the com.chordiant.service.message package. For more information about CustomObjects, refer to "CustomObjects and the CustomObjectHelper" on page 176. The XMLMessageDispatcher class is located in the com.chordiant.service.message.xml package.

For inbound messages, Chordiant uses a Message Driven Bean. The IMessageFilter and IMessageHandler classes are located in the com.chordiant.jxe package.

Understanding the Execution Flow

This section describes the following execution flows:

- "Outbound Messages" on page 242
- "Inbound Messages" on page 244

Outbound Messages

Figure 10-2 illustrates the outbound messaging reference implementation.

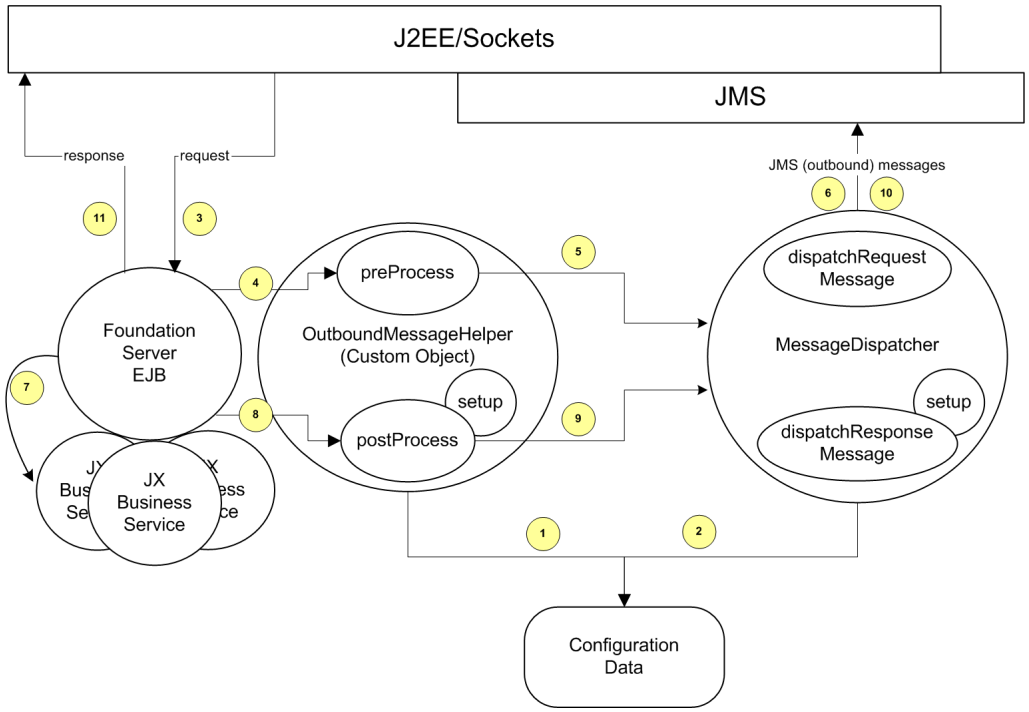


Figure 10-2: Outbound XML Messaging Reference Implementation Execution Flow

Here is the execution flow for outbound asynchronous messages:

1. During start up, Foundation Server loads a single instance of the OutboundMessageHelper, as defined in the configuration file.

   You can configure the OutboundMessageHelper to listen for requests to, and responses from, specific Chordiant business services.

   In a clustered application server environment, there is a single OutboundMessageHelper for each application server in the cluster.

2. The OutboundMessageHelper loads an instance of each unique MessageDispatcher defined in the configuration.

   In cases when there are several message filters referencing the same MessageDispatcher, the OutboundMessageHelper loads only a single copy of the MessageDispatcher.

3. A Chordiant business service is called by an application or a peer service.

4. In cases when a request filter is configured for the business service, the OutboundMessageHelper calls the preProcess method before the business service is invoked.

5. The preProcess method invokes the dispatchMessage method in the MessageDispatcher class. You can customize the implementation of the dispatchMessage method. Then you can reference the customized class as part of the configuration information.

6. The MessageDispatcher can format the message and send it to the appropriate JMS destination or it can implement whatever custom logic is appropriate to handle the request.

   The MessageDispatcher transforms the XML document into an XML string and send this string to JMS using the configuration data (such as the initial context, topic and queue specifiers, and connection information). You can customize a MessageDispatcher to handle messages other than XML.

7. The business service is invoked.

8. In cases when a response filter is configured for the business service, the OutboundMessageHelper calls the postProcess method after the business service completes.

9. The postProcess method invokes the dispatchMessage method in the MessageDispatcher.

10. The MessageDispatcher can format the message and send it to the appropriate JMS destination or it can implement whatever custom logic is appropriate to handle the request.

11. The response is returned to the application or peer service.

Inbound Messages

Figure 10-3 illustrates the inbound messaging reference implementation.

Note: The Foundation Server application server must be completely started up before the Event Server functionality is available.

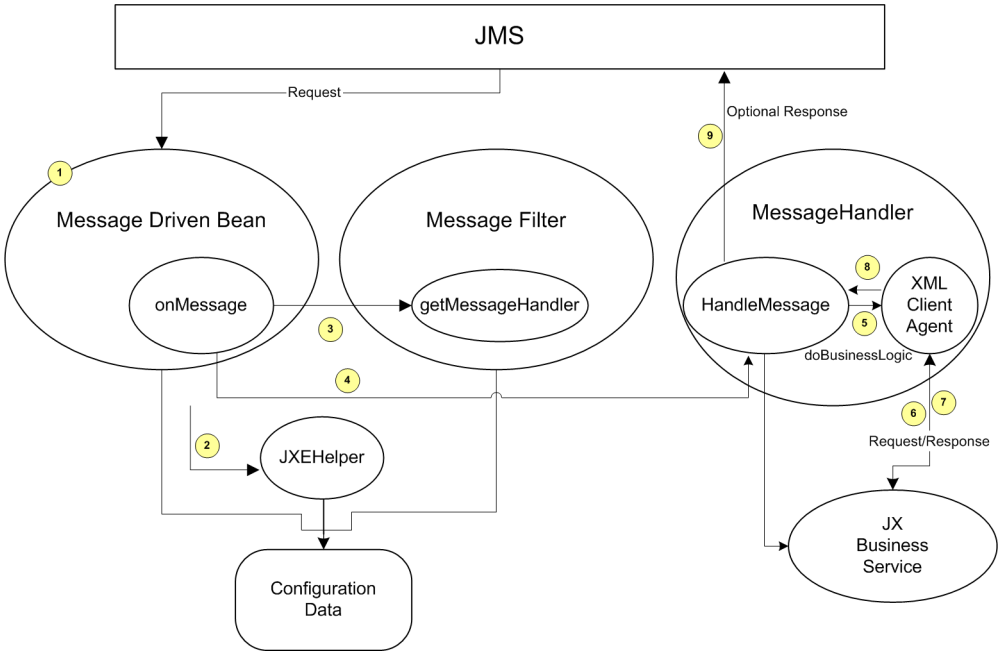


Figure 10-3: Inbound XML Messaging Reference Implementation Execution Flow

The execution flow for inbound XML asynchronous messages is:

1. When the Foundation Server application server has completely started up, a Message Driven Bean (MDB) is loaded. This MDB listens for messages from JMS. When a message arrives, the EJB container automatically routes the message to the MDB's onMessage method.
2. When the onMessage method receives a request, the MDB calls the JXEHelper to obtain a MessageFilter.
3. The MDB then calls the MessageFilter.getMessageHandler to obtain a MessageHandler for this request. The MessageFilter looks to the `JXE_MessageFilter.xml` configuration file to find which MessageHandler to provide to handle this specific message. If no match is found in the configuration file, the MessageFilter creates an instance of the default message handler, XMLMessageHandler.
4. The onMessage method on the MDB calls the HandleMessage method on the MessageHandler.

5. The handleMessage method calls the XML Client Agent to do the business logic.
6. The XML Client Agent calls the appropriate business service to do its business logic.
7. The service performs its work and returns its response to the XML Client Agent.
8. The XML Client Agent passes the response to the MessageHandler.
9. If required, you can develop a custom MessageHandler to return the response from the XML Client Agent to a JMS destination specified in the request message.

When the application is shut down, the application server shuts down the MDBs. The Event Server finishes processing the messages already in process and commits the session. Any transactions that cannot be completed are rolled back to maintain database integrity.

Security and Inbound Messages

Chordiant services require a username and authentication token before they can do their work. If you do not want to send both parameters in a message through the Event Server, you can send the username and password. In this case, the TextMessageHandler can contact the security service and obtain an authentication token and authenticate the caller with the Chordiant system.

Alternatively, you can specify the username and password from a configuration file. As in the previous scenario, the TextMessageHandler can contact the security service and obtain an authentication token. Be aware that this scenario is not as secure, since the sensitive information is located in a configuration file. This functionality is not provided, but you can customize the Event Server to provide this functionality if you require it.

Errors and Inbound Messages

If, for some reason, the MessageHandler is not able to process a message, it will throw an exception back to the MDB. The message will be removed from the incoming queue and sent to a message error queue. The MDB will then continue with the next incoming message. If there is a problem routing the message to the error queue, the Event Server will log a critical error and will shut down the current session.

Error queue functionality is provided by the application server. Configure this functionality through WebLogic or WebSphere, according to their documentation.

DIRECTING OUTBOUND MESSAGES TO QUEUES AND TOPICS

This section describes how to configure the OutboundMessageHelper and to create and configure custom Message Dispatchers to direct XML messages to JMS queues and topics within the Foundation Server environment.

To direct messages to queues and topics in JMS:

1. Configure the loading of the OutboundMessageHelper.

   You must include an entry within the `JXE_CustomObjects.xml` configuration file to load the OutboundMessageHelper, as shown in Code Sample 10-1.

```
<?xml version="1.0"?>
<!DOCTYPE Root SYSTEM "../../master.dtd">
<Root>
   <Section>CustomObjectConfiguration
      <Tag>OutboundMessageHelper.name
         <Value>com.chordiant.service.message.OutboundMessageHelper</Value>
      </Tag>
   </Section>
</Root>
```

Code 10-1: JXE_CustomObjects.xml Configuration File

2. Implement a custom Message Dispatcher, if required.

   The Message Dispatcher interface defines the outbound JMS Message Dispatchers. Classes implementing the interface are used and managed in a pool by the OutboundMessageHelper.

   Message Dispatchers manipulate XML request and response messages sent to, and between, Chordiant services. The XML messages can be transformed and routed to arbitrary destinations using JMS or other means. The OutboundMessageHelper maintains a list of all Message Dispatchers used for filtering Chordiant service requests and responses.

To implement a custom Message Dispatcher:

a. Implement the dispatchRequestMessage method.

   The dispatchRequestMessage method dispatches filtered Chordiant service request messages, enabling outbound XML messages to be sent to JMS and other potential destinations.

```
dispatchRequestMessage
   public void dispatchRequestMessage(
      java.lang.Object message)
   throws java.lang.Exception
```

Code 10-2: dispatchRequestMessage Method Signature

b. Implement the dispatchResponseMessage method.

The dispatchResponseMessage method dispatches filtered Chordiant service response messages, enabling outbound XML messages to be sent to JMS and other potential destinations.

```
dispatchResponseMessage
    public void dispatchResponseMessage(
        java.lang.Object message)
    throws java.lang.Exception
```

Code 10-3: dispatchResponseMessage Method Signature

c. Implement the ServiceControlResponse method.

The ServiceControlResponse method executes a service control on the Message Dispatcher. The ServiceControlResponse method returns a ServiceControlResponse object containing a success flag and message, and throws an exception when the service processing fails or the command is invalid.

```
serviceControl public com.chordiant.service.ServiceControlResponse
serviceControl(com.chordiant.service.ServiceControlRequest
theRequest)
    throws java.lang.Exception
```

Code 10-4: ServiceControlResponse Method Signature

You can use the XMLMessageDispatcher reference implementation as a guide for implementing a custom Message Dispatcher.

3. Configure the Message Dispatchers.

Note: Configuration is done in both the `OutboundMessage.xml` and `master.dtd` files. Relevant sections of both documents are shown in this step.

You can configure the following elements for Message Dispatchers:

— **RESOURCE_TAG_FOR_JMS_ENABLED**: Enables or disables access to JMS for the Message Dispatcher class. To enable or disable access, specify a value of true or false respectively. The default value is false, so you must change it to true for messaging to work.

Code Sample 10-5 illustrates how to enable access to JMS for the Message Dispatcher class.

```
<Tag>RESOURCE_TAG_FOR_JMS_ENABLED
    <Value>true</Value>
</Tag>
```

Code 10-5: Enabling Access to JMS

— **RESOURCE_TAG_FOR_DOCUMENT_TYPE**: Specifies the XML document, object, or message type expected by the Message Dispatcher class. The valid values are: JDOM, W3C, PAYLOAD, and STRING (not case-sensitive). There is no default value.

Code Sample 10-6 illustrates how to specify the document type of payload.

```
<Tag>RESOURCE_TAG_FOR_DOCUMENT_TYPE
    <Value>payload</Value>
</Tag>
```

Code 10-6: Specifying the payload Document Type

— **RESOURCE_TAG_FOR_TOPIC_OR_QUEUE**: Specifies whether the Message Dispatcher is to use a queue or topic. The value is a pointer to an entry that must be set in the messaging section of the `master.dtd` file.

```
<Tag>RESOURCE_TAG_FOR_TOPIC_OR_QUEUE
    <Value>&JXE_OUTBOUND_JMS_TOPIC_OR_QUEUE;</Value>
</Tag>
```

In Code Sample 10-7, the `master.dtd` file entry specifies that the Message Dispatcher is to use a queue.

```
<!ENTITY JXE_OUTBOUND_JMS_TOPIC_OR_QUEUE "queue">
```

Code 10-7: master.dtd Entry for Message Dispatcher

— **RESOURCE_TAG_FOR_OUTBOUND_TOPIC_OR_QUEUE_NAME**: Specifies the name of the outbound queue or topic. The value is a pointer to an entry that must be set in the messaging section of the `master.dtd` file.

```
<Tag>RESOURCE_TAG_FOR_OUTBOUND_TOPIC_OR_QUEUE_NAME
    <Value>&JXE_OUTBOUND_JMS_TOPIC_OR_QUEUE_NAME;</Value>
</Tag>
```

Code 10-8: Specifying the Name of the Outbound Queue or Topic

Code Sample 10-9 shows the actual name of the outbound queue is specified in the `master.dtd` file.

```
<!ENTITY JXE_OUTBOUND_JMS_TOPIC_OR_QUEUE_NAME
    "com_chordiant_jxe_OutboundQueue">
```

Code 10-9: master.dtd Entry for Outbound Queue

— **RESOURCE_TAG_FOR_JMS_FACTORY_NAME**: Specifies the name of the queue or topic connection factory. The value is a pointer to an entry that must be set in the messaging section of the `master.dtd` file

```
<Tag>RESOURCE_TAG_FOR_JMS_FACTORY_NAME
    <Value>&JXE_OUTBOUND_JMS_CONNECTION_FACTORY_NAME;</Value>
</Tag>
```

Code 10-10: Specifying the Name of the Queue or Topic Connection Factory

Code Sample 10-11 shows the connection factory definition is specified in the `master.dtd` file.

```
<!ENTITY JXE_OUTBOUND_JMS_CONNECTION_FACTORY_NAME
    "Foundation_Server_Queue_Connection_Factory">
```

Code 10-11: master.dtd Entry for Connection Factory

— **RESOURCE_TAG_FOR_MAX_SLEEP_SECONDS**: Defines the maximum number of seconds that the XMLMessageDispatcher will wait between attempts to find and create the JMS objects. The default value is 15. Any positive integer is valid.

```
<Tag>RESOURCE_TAG_FOR_MAX_SLEEP_SECONDS
    <Value>15</Value>
</Tag>
```

Code 10-12: Defining the Maximum Sleep Seconds

— **RESOURCE_TAG_FOR_MAX_SETUP_RETRIES**: Defines the maximum number of retries that the XMLMessageDispatcher will attempt to find and create the JMS objects. The default value is 10. Any positive integer is valid.

```
<Tag>RESOURCE_TAG_FOR_MAX_SETUP_RETRIES
    <Value>10</Value>
</Tag>
```

Code 10-13: Defining the Maximum Number of Retries

— **RESOURCE_TAG_FOR_JMS_PERSISTENCE**: Defines whether JMS messages are to be put to queues in a persistence fashion. Production systems should always use TRUE. If this tag is removed or commented out, the default value is TRUE.

```
<Tag>RESOURCE_TAG_FOR_JMS_PERSISTENCE
    <Value>FALSE</Value>
</Tag>
```

Code 10-14: Defining Using Queues or Persistence

Code Sample 10-15 illustrates the configuration section for the XMLMessageDispatcher reference implementation.

```
<Section>com.chordiant.service.message.xml.XMLMessageDispatcher
   <Tag>RESOURCE_TAG_FOR_JMS_ENABLED
      <Value>true</Value>
   </Tag>
   <Tag>RESOURCE_TAG_FOR_DOCUMENT_TYPE
      <Value>payload</Value>
   </Tag>
   <Tag>RESOURCE_TAG_FOR_TOPIC_OR_QUEUE
      <Value>&JXE_OUTBOUND_JMS_TOPIC_OR_QUEUE;</Value>
   </Tag>
   <Tag>RESOURCE_TAG_FOR_OUTBOUND_TOPIC_OR_QUEUE_NAME
      <Value>&JXE_OUTBOUND_JMS_TOPIC_OR_QUEUE_NAME;</Value>
   </Tag>
   <Tag>RESOURCE_TAG_FOR_JMS_CONNECTION_FACTORY_NAME
      <Value>&JXE_OUTBOUND_JMS_CONNECTION_FACTORY_NAME;</Value>
   </Tag>
   <Tag>RESOURCE_TAG_FOR_MAX_SLEEP_SECONDS
      <Value>15</Value>
   </Tag>
   <Tag>RESOURCE_TAG_FOR_MAX_SETUP_RETRIES
      <Value>10</Value>
   </Tag>
   <Tag>RESOURCE_TAG_FOR_JMS_PERSISTENCE
      <Value>FALSE</Value>
   </Tag>
   </Section>
```

Code 10-15: XMLMessageDispatcher Section of OutboundMessage.xml Configuration File

Code Sample 10-16 shows the corresponding messaging section of the `master.dtd` file.

```
!-- JXE Asynchronous Messaging Specific entities -->
<!--=========================================================================-->
<!ENTITY JXE_OUTBOUND_JMS_TOPIC_OR_QUEUE "queue">
<!ENTITY JXE_OUTBOUND_JMS_CONNECTION_FACTORY_NAME "FOUNDATION_SERVER_QUEUE_CONNECTION_FACTORY">
<!ENTITY JXE_OUTBOUND_JMS_TOPIC_OR_QUEUE_NAME "com_chordiant_jxe_OutboundQueue">
```

Code 10-16: Event Server Section of master.dtd Configuration File

4. Configure the OutboundMessageHelper to define the filters for service method request and response messages.

   You can configure the OutboundMessageHelper to specify the messages to inject into JMS queues and topics based on the specific Chordiant service method requests and responses.

   Specifically, you can configure the following elements in the OutboundMessageHelper section of the `outboundmessagehelper.xml` configuration file.

   — **RESOURCE_TAG_FOR_JMS_ENABLED**: Enables or disables outbound asynchronous XML messaging capabilities within Foundation Server. To enable or disable messaging, specify a value of true or false respectively. The default value is false, so you must change it to true for messaging to work. Code Sample 10-17 illustrates how to enable messaging.

```
<Tag>RESOURCE_TAG_FOR_JMS_ENABLED
    <Value>true</Value>
</Tag>
```

Code 10-17: Enabling Messaging

The OutboundMessageHelper can contain one or more filters in one of two formats:

— ***{service}.{method}*.response**: Specifies the fully-qualified name for the Message Dispatcher class that sends the response data initially directed to the ***{service}.{method}*** to the JMS queue or topic.

— ***{service}.{method}*.request**: Specifies the fully-qualified name for the Message Dispatcher class that sends the request data initially directed to the ***{service}.{method}*** to the JMS queue or topic.

Code Sample 10-18 illustrates how to specify the XMLMessageDispatcher as the class to send response data produced by the HelloWorldService.DOIT method.

```
<Tag>HelloWorldService.DOIT.response
    <Value>com.chordiant.service.message.xml.XMLMessageDispatcher</Value>
</Tag>
```

Code 10-18: Specifying XMLMessageDispatcher

Code Sample 10-19 illustrates the configuration section for the OutboundMessageHelper configured with the settings described in this step.

```
<Section>com.chordiant.service.message.OutboundMessageHelper
    <Tag>RESOURCE_TAG_FOR_JMS_ENABLED
        <Value>true</Value>
    </Tag>
    <Tag>AccountService.getAccount_String_String_CCICMAccount.request
        <Value>com.chordiant.service.message.xml.XMLMessageDispatcher</Value>
    </Tag>
    <Tag>HelloWorldService.DOIT.response
        <Value>com.chordiant.service.message.xml.XMLMessageDispatcher</Value>
    </Tag>
</Section>
```

Code 10-19: OutboundMessageHelper Section of OutboundMessageHelper.xml Configuration File

ACCESSING MESSAGES IN QUEUES AND TOPICS

You do not have to work with configuration files to receive incoming messages. You configure the MDBs through the application server's deployment descriptor.

Creating Additional MDBs

Each MDB can listen to one and only one destination. If you need to listen to more than one queue, you must create additional MDBs.

To create additional MDBs:

1. Create a copy of an existing MDB.
2. Give the new MDB a new JNDI name and point it to a different destination.
3. Save the MDB.

Chapter 11

Security

During execution, users (or the application run by a user) must log in using a user name and password. The Security Manager service then returns an authentication token. The authentication token is used in every business service method to identify the caller. By passing the authentication token in the method, the security authorization module can grant or deny the caller's access to the corresponding business service.

Authentication (logging onto the system) and authorization (permission to use specific objects) are two separate modules in security. This modular design enables you to customize your authentication and authorization implementation separately.

- *Authentication* implementation is registered to the Chordiant security as authentication provider. Chordiant security provides a default, LDAP-based authentication implementation. This implementation authenticates a user based on the username and password stored in external LDAP server (specifically, an Sun ONE Directory Server). This enables you to leverage an existing enterprise authentication infrastructure.

  Although the user information used by the authentication module can exist on an external LDAP server, the Chordiant authorization module requires a mirror image of the user information to be maintained in the Chordiant database to double-check a user's identity once they have been authenticated. The default Chordiant security implementation requires that only the "unique user identifier" of each user should be stored in the Chordiant database. This unique user identifier is the user's login identification, for example, the user login name or email address. In the context of an LDAP-based authentication implementation, this unique user identifier could be the relative distinguished name (RDN) of the user in the LDAP server.

- *Authorization* implementation is registered to the Chordiant security as security policies. Multiple policies can be registered. The default implementation provided by Chordiant is a security policy based on Access Control Entities (ACE). This policy is detailed throughout this chapter.

You can administer the Security Manager service, including adding resources and administering policies, using the Tools Platform Administration Manager. For more information, refer to the *Chordiant 5 Tools Platform Administration Manager Guide*.

SECURITY ELEMENTS

The Security Manager service uses the following types of information:

- **Authentication information** — This includes the user name and password. An authentication check ensures that the user has a valid username and password on the system. Once the authentication check is completed, an authentication token is returned.
- **Authorization information** — This includes profile information and access control information. Profile information specifies the roles and groups to which users are assigned. Access control information defines object access privileges for roles, groups, or users. Access control can be applied to roles, groups, or users.

This section describes the following topics:

Authentication

Foundation Server employs a user name and password to perform authentication.

The authentication process typically has these general steps:

1. A user requests an authentication token using a user name and password.
2. The Security Manager service verifies the user name and password against an external security server using LDAP.
3. When the user name and password are verified, the Security Manager service creates an authentication token representing the user and returns it to the client.

   The authentication token is passed as a parameter for service calls throughout the Chordiant system.

The Security Manager service maintains an authentication token for each authenticated request. Each authentication token includes expiration information which is checked during authorization requests. When token expiration occurs, the system raises an exception. Client applications can then request a new authentication token, which is generated with a new expiration time. Token expiration can be turned off using the tokenexpiration tag in the security configuration file.

If you choose, you can customize the authentication token. For additional information, refer to "Customizing the Authentication Token" on page 282.

Authorization

Authorization can be performed for each available resource, including each method on each service. Access to the methods can be managed at a user, group, or role level. Access can also be managed by individual APIs or at the service level. The administration of business service API security is done by granting or denying access to the corresponding resource objects. By default, the admin role can call all business service APIs. For details on managing authorization of business services, refer to "Managing Business Services as Resources" on page 270.

Levels of Security and Principal Identifiers

Foundation Server grants and restricts security based on the following principal identifiers:

- **Role** — You can create roles within an organization and assign users to them. Examples of roles include Administrator, Director, Call Center Director and Accountant. A user can have more than one role.

  In Figure 11-1, there are two roles: the Task Force Role and the Manager Role. Users have been assigned to these roles.

  Role information is stored in the Roleinfo2 table.

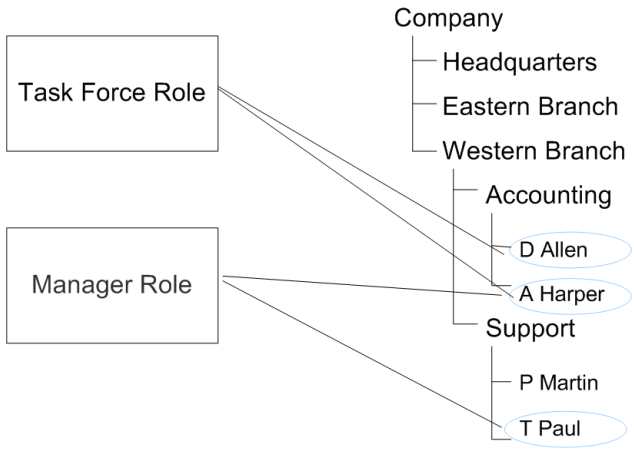


Figure 11-1: Levels of Security

- **Group** — Groups are a hierarchical, logical grouping of users. Each group in an organization, independent of whether the group is based on geography or function, has security privileges associated with it. Each user can belong to only one group, but groups can belong to other groups in a hierarchy.

  Each individual in the organization is assigned to a single group. However, they are all a part of the largest root group, Company.

  In Figure 11-1, A. Harper is a member of the Accounting, Western Branch, and Company groups.

  Group information is stored in the Groupinfo table.

- **User** — A person within an organization. A user can be assigned zero or more roles, but can only belong to a single group. However, users implicitly belong to the corresponding parent and ancestor groups within the hierarchy.

  In Figure 11-1, D. Allen and A. Harper are assigned to the Task Force role, since they are involved in this special project. In addition, A. Harper and T. Paul are both managers and so are assigned to the Manager role.

  User information is stored in the Userinfo table.

Roles and permissions are assigned using the Administration Manager tool. Refer to the *Chordiant 5 Tools Platform Administration Manager Guide* for details on using this tool.

Objects Under Security Control

The following entities can be controlled by Chordiant security. They are all administered through the Administration Manager tool. Note that in addition to being security principals, as described in "Levels of Security and Principal Identifiers" on page 255, roles, groups, and users are also objects which are under security control.

- **Roles** — You can grant or deny access to role records, including reading, creating, deleting, and modifying a role. To assign a role to a user, the user making the assignment must have the Modify right for the role. This user is often the Administrator, but other users can also have this right. Exercise caution when providing Modify rights to sensitive roles with a lot of functionality; only trusted users should be able to assign themselves or others to highly-sensitive roles.
- **Groups** — You can grant or deny access to group records, including reading, creating, deleting, and modifying groups. As with assigning roles, the Modify right gives access to assign users to groups. The user must have the Modify right to both the source group and target group for moving users between groups. Again, these functions are usually under the control of the Administrator, but any user who is given the Modify right can assign users to groups.
- **Users** — You can grant or deny access to user records, including reading, creating, and deleting users and assigning them to groups and roles.
- **Properties** — Properties define an object. For example, a user object can contain user name and password properties. A queue object can contain language, service type, and product properties. You can control access to properties, so only certain parties are granted read or write access to sensitive information like passwords.
- **Resources** — Resources are structured string objects, which by themselves do not have any meaning. They merely *represent* actual resources that are used by applications. Applications can use resources to manage application-specific security. For example, Foundation Server uses resources to manage security access to business service APIs. The CAFE framework uses resources to manage security access to desktop menus. By using resources to manage application-specific securities, applications do not have to implement their own security policy.

  Refer to "Managing Business Services as Resources" on page 270 for details on organizing information about services.

- **Queue Definitions** — The logical representation of JMS queues lives in the Chordiant database. You can grant or deny users' access to queues.

Access Control Lists and Entities

The Security Manager service uses Access Control Lists (ACL) and associated Access Control Entities (ACE) to manage access to objects. An ACE contains the following information:

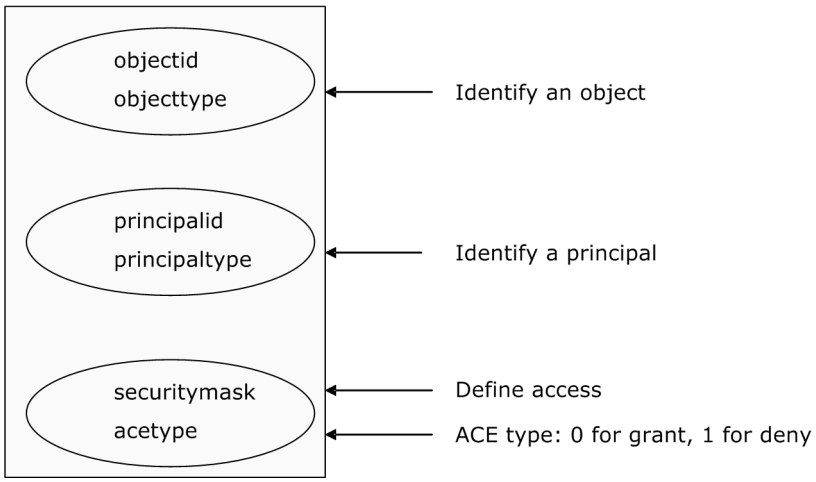


Figure 11-2: ACE Record Stored in ACE2 Table

- **objectid** — Together with the objecttype, uniquely identifies the object with which the ACE is associated.
- **objecttype** — Describes the object (see "Objects Under Security Control" on page 256).Together with the objectid, uniquely identifies the object with which the ACE is associated.
- **principalid** — Uniquely identifies the user, group, or role to which the access control information applies.
- **principaltype** — User, Group, or Role.
- **securitymask** — A 64-bit mask used to represent specific security rights. See "Security Mask" on page 258.
- **acetype** — 0=grant ACE (grants the right of a principal to an object); 1=deny ACE (denies the right of a principal to an object).

Security Mask

The security mask contains the rights for a specific resource:

- **Read** — Enables principal to read, but not modify an object. This includes, for example, access to call a service API, which is the type of access most commonly used.
- **Modify** — Enables principal to change an object. The Modify right is used by different objects for different purposes. For example, to be able to assign a role for a user, you must have Modify access to that role. Modifying a service API means being able to actually change the code. To call the service API, you only need Read access to it.
- **Delete** — Enables principal to remove the object. Use this sparingly—only certain individuals should be able to delete objects.
- **Manage security** — Enables principal to modify the security characteristics for the object. This right allows for security delegation. By default, the admin user has full access to all objects. The admin user can assign the manage security right to other users. Users who are assigned this right can then grant or deny other security access rights for the object to other users, groups, and roles.

This information is encoded in the first four bits of the security mask.

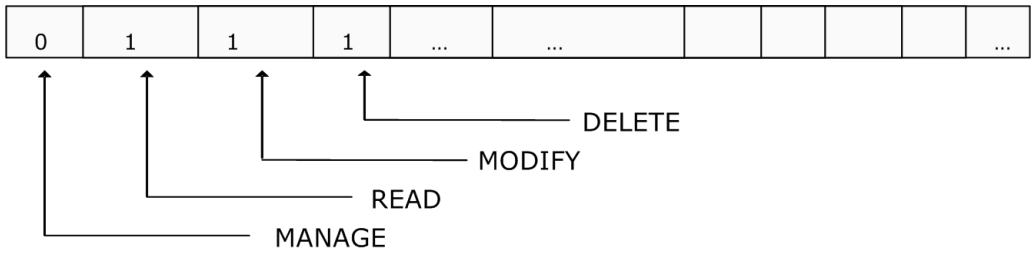


Figure 11-3: Security Rights Encoded in Security Mask

This example shows that this principal has Read, Modify, and Delete rights, but does not have Manage rights.

Unused bits in the security mask allow for future extension of object security. For example, new access privileges, like Assign instead of Modify, can be introduced for existing object types. Or new object types can be introduced with object-specific access privileges other than the currently available Read, Modify, Delete, and Manage rights.

Figure 11-4 illustrates the logical relationship between objects, ACLs, and ACEs.

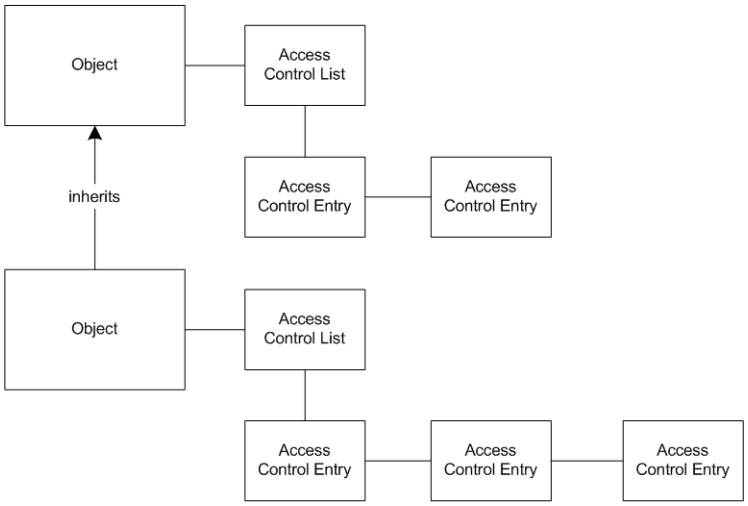


Figure 11-4: Access Control Lists and Entities

Security Resolution

Every object within Foundation Server can have an ACL, which can consist of zero or more ACEs. Each ACE specifies the rights (positive or negative) of the principal specified using the principal identifier and type.

In cases when an object does not directly have an associated ACL, it inherits the ACL from parent and ancestor objects. In addition, each ACL is sorted to place ACEs with negative rights in front. This means that the Security Manager service first checks to determine whether a principal has been explicitly denied access to a object before determining whether positive rights exist.

In cases when no security resolution can be found for an object, the default behavior is to deny access to the object to all principals.

The security resolution mechanism uses the flow illustrated in Figure 11-5.

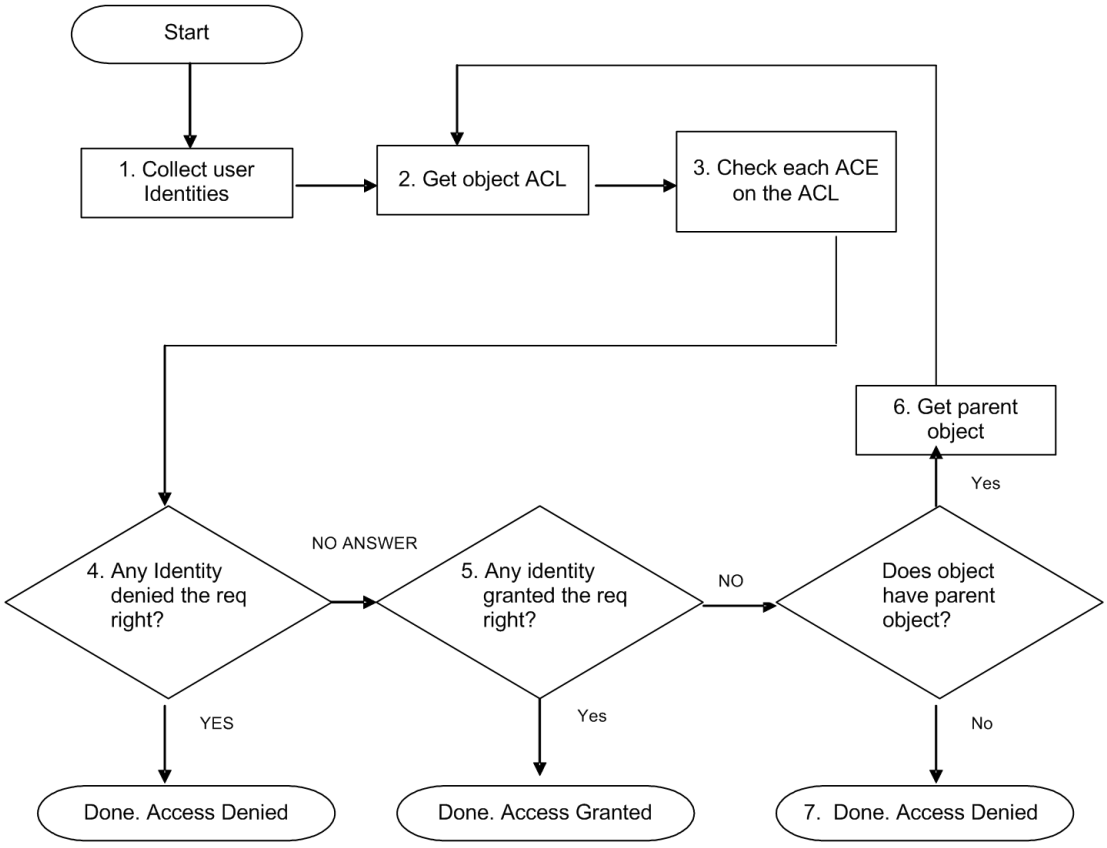


Figure 11-5: Security Resolution Flow

1. The system determines the user id, role ids, and group ids of the user.

   Users can be assigned multiple roles, such as Engineer, Customer Service Representative, or Accountant, as required. Likewise, while users can only belong to a single group, users are also implicitly associated with the parent and ancestor groups in the organizational hierarchy.

2. The system gets the ACL.

3. The system gets the ACEs associated with the ACL.

4. The system checks ACEs with negative rights, if available, to determine whether any of the principals collected in Step 1 have been explicitly denied the requested access to the object.

   There is no difference in significance between principal types. This means that a user can be denied access to an object because of the user id, role ids, or group id.

   If the requested access is explicitly denied to one of the principals, stop here. The requested access is denied.

5. The system checks ACEs with positive rights, if available, to determine whether any one of the principals has been given the requested access to the object.

   If the requested access is granted to one of the principals, stop here. The requested access is granted.

6. In cases when no answer is found in Step 4 or Step 5, either because:

   — no ACL is defined for the object, or

   — the defined ACL has not defined the requested access in any one of the ACE records,

   the system checks the ACL of the parent object, if one exists.

   This process of moving up the object hierarchy is repeated until negative or positive access can be determined.

7. In cases when no answer can be found in the object hierarchy, the system denies access to the object.

Security Resolution Example

Figure 11-6 shows the security permissions at Harmony Bank. It shows the users, groups, roles, and resources. For information about resource structure, refer to "Managing Business Services as Resources" on page 270.

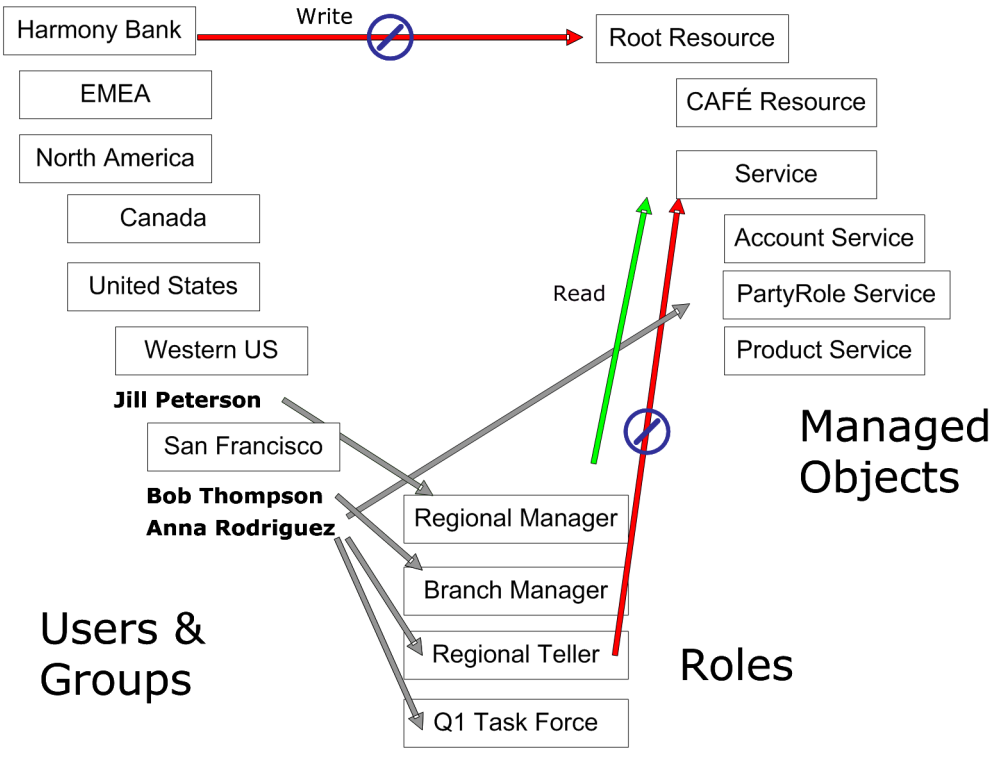


Figure 11-6: Security Resolution at Harmony Bank

First, determine the groups and roles for a specific user.

Anna Rodriguez is a member of the following groups: Harmony Bank, North America, United States, Western US, and San Francisco. She also plays the following roles: Branch Teller and Q1 Task Force.

Next, determine the rights of Anna's identity: Can Anna access the Account Service?

Look directly at the Account Service. None of her roles has any explicit grants or denials of access. So look to the parent object in the hierarchy, Service. Anna, as a Branch Teller, has an explicit denial of access to the Service group. That will hold for all services within that group as well. So Anna does not have access to the Account Service.

Does Anna have access to the PartyRole Service?

Yes. You can see from the explicit access grant from her role as a Branch Teller to the PartyRole Service that she has access. It does not matter that she is denied access to the Service resource. You only look up the hierarchy when there is no explicit information at the target level.

It is possible for a user to have both a grant and a deny at the same level, perhaps from being assigned to more than one group. In this case hierarchy is not involved. Denial always wins and the user will not have access to that particular resource.

Note that the denial of writing at the root level does not affect the reading grant at lower levels. Reading a service means that you have access to it and can use it.

Special Objects

There are several special objects within the security architecture to allow for convenience of security administration.

Special User

- **Administrator** — The special administrator user has full access to all objects. The administrator's userid=1. Regardless of the username, as long as the userid=1, this user has full access.

Special Roles

- **Everyone** — This role is automatically assigned to every user and cannot be un-assigned. This is convenient for specifying global rights. For the Everyone role, the roleid=3.
- **User** — This role is automatically assigned to every user added by Profile Manager.

Special Object

- **EveryObject** — An object with the objectid = 0 represents every object of that type. This is useful for granting or denying access to objects that do not support hierarchy, for example, the queue object.

Security Access to Non-Existent Objects

Access to objects that do not exist, for example a queue with a bogus ID, depends on how you set up your system. This provides you more flexibility in your system instead of just globally granting or denying access to all non-existent objects for all users.

Scenario

For example, say a user is trying to access a queue with an id=999. Let's say that this queue does not exist in the system. When this request goes through the authorization process, what will happen?

Use Case A: EveryObject Access

If this user has explicit access to the special object, EveryObject, as described above, this user will automatically have access to queue 999, even though this queue does not actually exist. As described in "Security Resolution" on page 259, if no ACEs are found for the object in question, the system checks upstream for specific grant or deny access. A user with an explicit EveryObject grant will be able to access even a non-existent object.

If a system is deployed like this, then depending on the specific API implementation, many API implementations on non-existent object IDs will not be operative, while other APIs can send a message saying the object does not exist.

Use Case B: No Access

If this user does not have explicit access to the special object, EveryObject, as described above, this user will be denied access to Queue 999. Since this queue object does not exist, by definition, it will not have an ACE. Without an ACE, there are no explicit grants to this object, so access is denied. A security denied exception will be thrown.

However, remember that receiving this exception could mean either that

- the object doesn't exist, or
- the object exists, but the user doesn't have access to it.

In constructing your own messages, in this situation, you can choose to write a message like this:

> "Cannot access this queue because it either has been erased or because you do not have access to it."

SECURITY ARCHITECTURE

The Security Manager service consists of a set of components that interact with other Foundation Server services and client applications to implement the security model.

Figure 11-7 illustrates the Security Manager service and its related components.

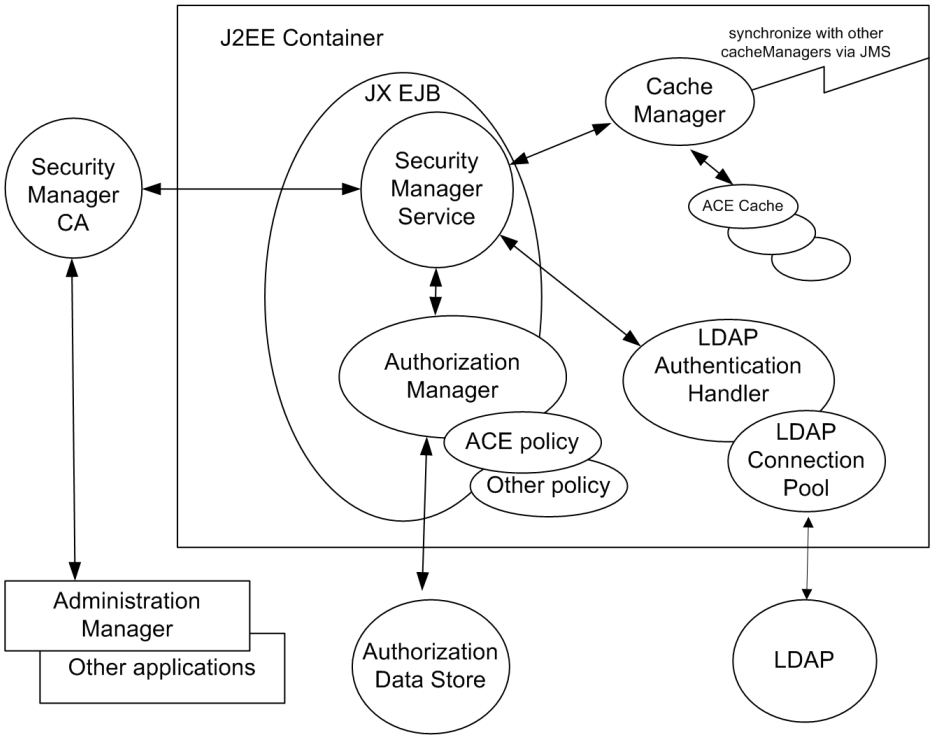


Figure 11-7: Security Manager Service Architecture

The principal entities related to the Security Manager service are:

- **Security Manager Service** — The Foundation Server service responsible for implementing the authentication and authorization model for Chordiant applications.
- **Security Manager Client Agent** — The interface used by client applications to invoke remote services on the Security Manager.
- **LDAP Authentication Handler** — Chordiant security provides a default LDAP-based authentication implementation. The Chordiant authentication implementation provides the following services:
  - — Authentication
  - — Change password
  - — Two-way synchronization between LDAP and Chordiant database
  - — Get password grace period (requires specific LDAP server configuration)

  During startup, Foundation Server loads the Security Handlers specified in the `SecurityManager.xml` file. For more information on the `SecurityManager.xml` file, see "Configuring SecurityManager.xml" on page 271.
- **Authorization Manager** — Determines whether a principal, identified by either a user id, role id, or group id, has access to a specified object within Foundation Server by accessing the authorization data store.
- **Administration Manager** — The utility used to grant and deny access and rights to objects by means of Access Control Lists and Access Control Entities. For more information, refer to the *Chordiant 5 Tools Platform Administration Manager Guide.*
- **Cache Manager** — The cache manager manages all of the caches listed below and makes sure that caches are synchronized across clusters. There is one cache manager per JVM. Caches are synchronized through JMS. To maintain security, this synchronization should take place across a secure wire.
  - — **ACE cache**: All ACE records are loaded at startup time.
  - — **Resource cache**: All resources are loaded at startup time. The Resource cache is used by the ResourceSecurity implementation for better performance.
  - — **User profile cache**: Cache of users, groups, and roles. Only users who made authorization inquiries recently are cached. So users who are least active have more chance to be discarded from cache. The cache size is configurable, so you can decide how many users can be cached in the system.

USING THE SECURITY MANAGER SERVICE

The Security Manager service enables you to use the security features of Foundation Server within your enterprise applications. The Security Manager service offers an API, accessible through the SecurityMgrBeanClientAgent, that can be partitioned according to the following functions:

- Authenticating users
- Authorizing users
- Managing ACLs and ACEs

The Security Manager service raises a SecurityTokenExpiredException when a security token expires. The exception is propagated through the calling stack, enabling the exception to be handled implicitly or explicitly by calling methods.

Applications can handle expired tokens either by renewing the token, or displaying an error message. Applications should only renew tokens when there is an active HTTP session.

Authentication token expiration is a back-end expiration mechanism. It enforces a system-level expiration of each user authentication, which is independent of any application-level "time-out" implementation. For example, for a browser-based thin client implementation, the application could implement a session time-out mechanism. Such time-out mechanisms, in general, should be no more than the system expiration time configured for authentication token expiration. This way, the application can have a chance to renew the authentication token before the token expires.

This section describes the following topics:

- "APIs for Authenticating Users" on page 267
- "APIs for Authorizing Users" on page 268
- "Managing Access Control Lists and Entities" on page 269

Notes: This section provides an overview of the methods in the SecurityMgrBeanClientAgent. For detailed information about each method, refer to the Javadoc.

These facilities are also exposed through the Administration Manager application provided with the Chordiant 5 Tools Platform. Refer to the *Chordiant 5 Tools Platform Administration Manager Guide* for details.

APIs for Authenticating Users

The Security Manager service enables applications to authenticate users, change user passwords, and renew an authentication token using the following methods:

- **authenticate** — Returns an authentication token for a user based on the specified user name and password.

```
String authenticate( userName, password )
```

- **getAuthenticationToken** — Returns an authentication token object, given the authentication token string. The password field of the token object is intentionally set to empty to obscure it from hackers.

```
AuthenticationToken getAuthenticationToken( tokenStr )
```

- **renewToken** — Returns a renewed authentication token.

```
String renewToken( userName, authenticationToken )
```

- **changePassword** — Changes the password for a user. The method requires a user name and existing authentication token. This API is generally used by the Profile Manager.

```
void changePassword( userName, authenticationToken, password, newPassword )
```

```
void changePassword( userName, authenticationToken, password )
```

- **getPasswordGracePeroid** — Determines the grace period for a user password.

```
Integer getPasswordGracePeriod( userName, authenticationToken )
```

  In order for the API to work (returning the time, in seconds, left until the password expires) the Sun ONE Directory Server must be configured appropriately:

  — The password policy on the Sun ONE Directory Server must be configured to enable password expiration. Select the Password expires after xxx days option and enter a valid number for the days.

  — The Sending warning xxx days before password expires option must have the xxx days set to be equal to the password expiration date entered above.

Note: The API will not be able to obtain a password grace period without this setup. It will return zero instead, meaning no password expiration information is available.

APIs for Authorizing Users

Applications can grant or deny access and rights, as well as determine the rights for an object using the methods listed below.

- **authorize** — Checks to see if the caller has read access to the specified resource. The resource must be specified with the complete path within the resource directory, separated by slashes (/). A second form of this method enables one user to check the read authorization rights of another user for a given resource name.

```
authorize( username, authtoken, resourcename )
```

```
authorize( username, authtoken, username, resourcename )
```

- **haveRight** — Test for a specified access right for either an object, or for an array of objects. The right is given either as a security mask or as a right. This method uses userid instead of username. The authorize method, described above, uses username.

```
boolean haveRight( userid, objid, objType, securitymask )
```

```
boolean haveRight( userid, objid, objType, testRight )
```

The following group of methods are cumulative operations, adding to existing grant or deny rights:

- **denyAccess** — Explicitly deny a right for an principal/object pair The principal can be either a user, a role, or a group.

```
void denyAccess( userName, authenticationToken, pid, ptype, objid, objtype, right )
```

- **getAllowedRight** — Get the explicitly granted right for the principal/object pair.

```
Right getAllowedRight( pid, ptype, objid, objtype )
```

- **getDeniedRight** — Get the explicitly denied right for a principal/object pair.

```
Right getDeniedRight( pid, ptype, objid, objtype )
```

- **getRight** — Determine the access right for an object, or for an array of objects.

```
Right getRight( userid, objid, objType )
```

- **grantAccess** — Explicitly grant right for the principal/object pair. The principal can be either a user, a role, or a group.

```
void grantAccess( userName, authenticationToken, pid, ptype, objid, objtype, right )
```

The following group of methods are non-cumulative operations. They set the grant or deny rights to the values specified, overriding previous rights:

- **setAllowedRight** — Explicitly grant a right for the principal/object pair. The principal can be either a user, a role, or a group.

```
void setAllowedRight(userName, authenticationToken, pid, ptype, objid, objtype, right)
```

```
void setAllowedRight(userName, authenticationToken, pid, ptype, objid, objtype,
    securitymask)
```

- **setDeniedRight** — Explicitly deny a right for the principal/object pair. The principal can be either a user, a role, or a group.

```
void setDeniedRight(userName, authenticationToken, pid, ptype, objid, objtype, right)
```

```
void setDeniedRight(userName, authenticationToken, pid, ptype, objid, objtype,
    securitymask)
```

Managing Access Control Lists and Entities

Applications can use the Security Manager service to add, remove, and retrieve ACE records from Access Control Lists associated with objects using the following methods:

- **addAce** — Adds one or more ACE records.

```
addAce( userName, authenticationToken, ace )
```

```
void addAce( userName, authenticationToken, list )
```

- **getAce** — Get one or more ACE records for an object, for a principal, or for a principal/object pair.

```
ArrayList getAce( objid, objtype )
```

```
ArrayList getAce( Principal p )
```

```
ArrayList getAce( pid, ptype, objid, objtype )
```

- **getAceByPrincipal** — Get all the ACE records created for the principal.

```
ArrayList getAceByPrincipal( pid, ptype )
```

- **removeAce** — Remove one or more ACE records, remove all ACE records for an object, or remove all ACE records for an principal/object pair.

```
void removeAce( userName, authenticationToken, objid, objtype )
```

```
void removeAce( userName, authenticationToken, ace )
```

```
void removeAce( userName, authenticationToken, list )
```

```
void removeAce( userName, authenticationToken, pid, ptype, objid, objtype )
```

- **removeAceByPrincipal** — Remove all the ACE records created for the principal .

```
void removeAceByPrincipal( userName, authenticationToken, pid, ptype )
```

Managing Business Services as Resources

When you create business services, you must specify who can access them. Chordiant security uses Resources to control access to business service APIs.

All resources are stored in the resourceInfo table. Business services are resource groups. The methods, or APIs, within the business service are individual resources. When a user has Read access to a business service or its methods, that user is authorized to call that service or method. You can grant Read access to a certain service, including all of its methods, to certain methods and not to others, or to the entire service, but specifically denying access to certain methods.

Note: The default deployment allows all users Read access to the top "Service" resources, which means all users can call all business service APIs by default. For finer security control to individual services and APIs, you need to apply security administration to individual resources that correspond to the services and APIs. See "Adding a New Service as a Resource" on page 271 for additional details.

Service API resources must be listed under the service name with the following format: servicename.APIname.

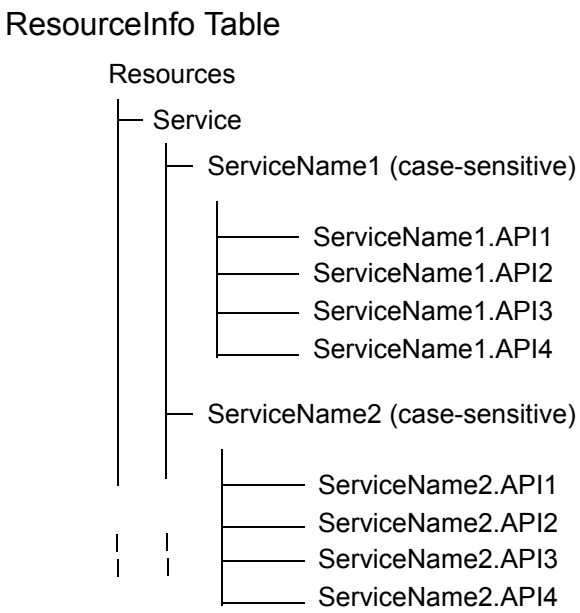


Figure 11-8: Managing Business Services in the ResourceInfo Table

Consider the following when configuring business service APIs:

- The business service name must be created as a resource under the Service resource node. The name is case-sensitive.
- Business service APIs are created as resources under the corresponding business service name, one resource for each API. The resource name should be in the format of BusinessServiceName.API_name. Names here are also case-sensitive. The API names are not necessarily the same as the Java API name. Instead, they should be the same as defined in the business service implementation constant file.

  Keep in mind that creating resources for business service APIs is optional. It is only required if you have a need to manage access to business service APIs at the individual API level.

Refer to "Security Resolution" on page 259 for details on how a user's access to a resource is resolved.

Adding a New Service as a Resource

To add a new service to security control:

1. Add an entry for the service as a new resource under services. You can manage the Resources through the Administration Manager.
2. Optional. Add some or all of the method (API) constants from the METHOD_CONSTANTS class you created with your services. (See page 115 for details on defining method constants.)
3. In the Administration Manager, decide who will have access to the service and its APIs. Refer to the *Chordiant 5 Tools Platform Administration Manager Guide* for details.

CONFIGURING SECURITYMANAGER.XML

Foundation Server stores the configuration items for the Security Manager service in the `SecurityManager.xml` file.

Code Sample 11-1 on page 272 illustrates a sample security configuration in the *{CHORDIANT_ROOT}*`/config/Chordiant/components/master/SecurityManager.xml` file. Annotations for the configuration file are to the right of the code sample.

{CHORDIANT_ROOT} corresponds to the chordiant.configuration.configurationRootDirectory parameter in your application server. Refer to "Configuration Files and the ConfigurationRootDirectory" on page 46 for more information.

For more information about configuring security for Chordiant 5 Foundation Server, refer to the *Chordiant 5 Tools Platform Administration Manager Guide*. For more information about configuration files in general, refer to Chapter 7, "Configuration Files".

Notes: The `SecurityManager.xml` file uses substitution values from the `master.dtd` file. Here, the actual values are shown. If you open the configuration file with a text editor, you will see the references to `master.dtd`.

Be sure that this and other configuration files are kept secure. This file contains the LDAP manager password and other sensitive information.

```
<Section>SecurityManager
    <Tag>authenticationHostname                 Specify the LDAP server information for Hostname and
        <Value>localhost</Value>                Portnumber for both authentication.
    </Tag>

    <Tag>authenticationPortnumber
        <Value>1389</Value>
    </Tag>
    <Tag>authenticationManagerRdn               The directory manager distinguished name (DN)
        <Value>cn=Directory Manager</Value>
    </Tag>

    <Tag>authenticationManagerPassword          The password for the directory manager.
        <Value>managerd</Value>
    </Tag>

    <Tag>loginDnPrefix                          The prefix for authenticating the login. Usually,
        <Value>uid</Value>                      this prefix is the LDAP attribute that constitutes
    </Tag>                                      the relative distinguished name (RDN) of the user
                                                record in the LDAP server. valid values: uid | cn
    <Tag>loginBaseDn                            The suffix for authenticating the login. This is the
        <Value>ou=People,ou=chordiant,o=com     location of the user records in the LDAP server.
        </Value>
    </Tag>

    <Tag>connectionpoolsize                     The number of connections in the connection pool.
        <Value>4</Value>
    </Tag>

    <Tag>tokenexpirationinseconds               The time after which the authentication token
        <Value>0</Value>                        expires. The maximum value is 64800 seconds (18
    </Tag>                                      hours). Zero (0) specifies that the expiration check
                                                is disabled. To enable the expiration check, change
                                                this value to a non-zero number.

                                                A SecurityTokenExpiredException is thrown when tokens
                                                expire.

                                                Default value: 0
    <Tag>update_external_user_data              Used to synchronize the LDAP and Chordiant databases.
        <Value>true</Value>                     If set to true, add/delete user from the Profile
    </Tag>                                      Manager Admin tool will synchronize the changes
                                                between LDAP and Chordiant databases.
                                                If false, the user will only be added/deleted to/from
                                                the Chordiant database with no changes to LDAP.
                                                Default value: true
```

Code 11-1: Sample security.xml Configuration File

The values below this point do not generally require changes.	
<Tag>authentication_on_flag <Value>true</Value> </Tag>	Determines whether authentication is activated. This can be used during development. Valid values: true | false (not case-sensitive)
<Tag>authorization_on_flag <Value>true</Value> </Tag>	Determines whether authorization is activated. This can be used during development. Valid values: true | false (not case-sensitive)
<Tag>authenticationProtocol <Value>ldap</Value> </Tag>	The communication protocol to use between the Security Service and the LDAP server.
<Tag>token.object <Value>com.chordiant.core.security. AuthenticationToken</Value> </Tag>	The class to use when creating authentication token objects. If you extend this class, you must specify your customized class here. See "Customizing the Authentication Token" on page 282.
<Tag>authentication.provider <Value>com.chordiant.core.security. LDAPAuthenticator</Value> </Tag>	The authentication driver to load to communicate with the LDAP server.
<Tag>security.policy.1 <Value>com.chordiant.server. newsecurity.AcePolicy</Value> </Tag>	The Java class of the security (authorization) policies.
<Tag>serviceUserName <Value>service</Value> </Tag>	The username required to authenticate the service. Any valid String is acceptable. Default value: service Default value if not present in SecurityManager.xml: service See "Service Username and Password" on page 273.
<Tag>servicePassWord <Value>service</Value> </Tag>	Password required to authenticate service. Any valid String is acceptable. Default value: service Default value if not present in SecurityManager.xml: service See "Service Username and Password" on page 273.
</Section>	

Code 11-1: Sample security.xml Configuration File (Continued)

Service Username and Password

To perform service to service calls, services must have their own username and password. This is similar to a client agent's using a username and password to contact a service. The values for service username and password are configured for all services in the `SecurityManager.xml` file as service/service, as shown in Code Sample 11-1.

There must be a user in the LDAP system with this username and password. The Chordiant LDAP seed data includes a user Service with the password Service. If you are using your own LDAP implementation, you can migrate this user into your own system. Alternatively, you can use a

different username and password to assign to the services. This user, perhaps an administrator, must have access to all services. You will need to change the values in the `SecurityManager.xml` file to match the username and password you choose.

Synchronizing Cache Across Clusters with JMS

The Security Manager in each cluster has its own cache. To keep the security cache synchronized across clusters, you must set up Java Messaging Service (JMS).

The Cache Manager component of security module uses the JMS topic connection factory, USERPROFILE_TOPIC_CONNECTION_FACTORY to connect to the JMS topic, UserProfile_Cache_Topic. Chordiant will automatically generate scripts during the installation to set up the appropriate JMS vendor of your choice.

Note: This cache is used internally by Chordiant 5 Foundation Server. It is not a publicly available facility.

System Security

Your company's Information Technology (IT) or Management Information Services (MIS) department protects your internal network so your back-end data stores and legacy systems are secure and only accessible to trusted clients on your network. Chordiant does not prescribe how this should be done.

See "Security and the SocketGatewayService" on page 153 for additional security information.

UNDERSTANDING INTERACTIONS BETWEEN SECURITY MANAGER AND AUTHENTICATION HANDLER

It is important to understand the interactions between the Security Manager service and the Authentication Handler. The Authentication Handler is called by the Security Manager and is responsible for creating and validating authentication tokens, which enable users to access Chordiant services.

Creating an Authentication Token

To get into the system, a client application contacts the SecurityMgrBeanClientAgent. In turn, the SecurityMgrBeanClientAgent calls the Security Manager service, which calls the Authentication Handler. The Authentication Handler is responsible for creating the authentication token.

Figure 11-9 illustrates how the authentication token is created within the Authentication Handler. The numbers correspond to numbered steps after the figure. These components are described in this section, and also in "Customizing the Authentication Handler" on page 277.

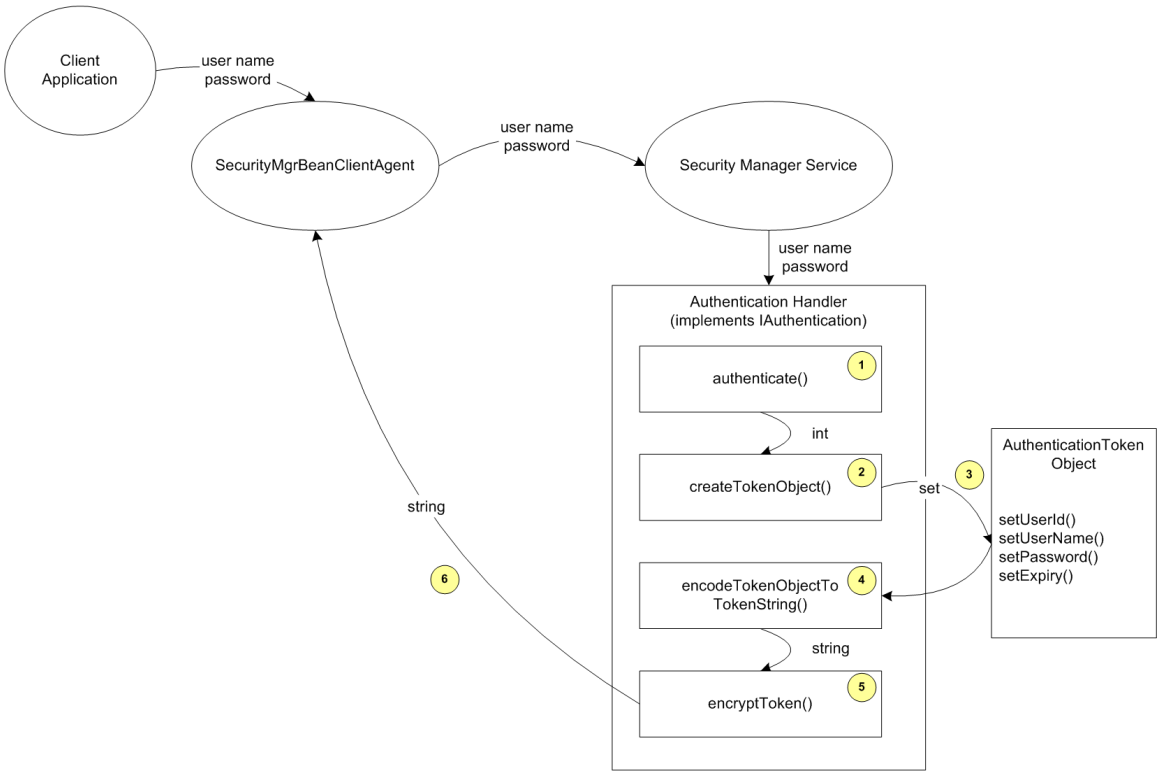


Figure 11-9: Obtaining an Authentication Token

The process of creating an authentication token includes:

1. The authenticate method receives the user name and password from the calling application. The authenticate method returns an integer, which specifies whether the user is authorized and is entitled to an authentication token.
2. Upon successful completion of Step 1, the createTokenObject method acts as a factory to create the AuthenticationToken object. The AuthenticationToken object is an instance of the class described in the token.object tag of the `SecurityManager.xml` configuration file. For more information on customizing the authentication token, refer to "Customizing the Authentication Handler" on page 277.
3. The Security Manager calls the setters on the AuthenticationToken object to set the user id, user name, password, and expiration date for the AuthenticationToken object.
4. The encodeTokenObjectToTokenString method encodes the object as a string. The parameters that were set in Step 3 are used in the encoding process such that this string can be later decoded back to the same AuthenticationToken object through the decodeTokenStringToTokenObject method. The decode method is used during validation. For additional information on this method, refer to page 278.
5. The encryptToken method receives the encoded string and further encrypts it. For additional information on this method, refer to page 278.
6. The token is sent back to the caller invoking the SecurityMgrBeanClientAgent API. The caller can hold onto this token and use it in subsequent client agent invocations.

Validating an Authentication Token

Once the client application has obtained the authentication token string, it can use the token when contacting any Chordiant service.

Figure 11-10 illustrates how a Chordiant service validates the authentication before performing its work.

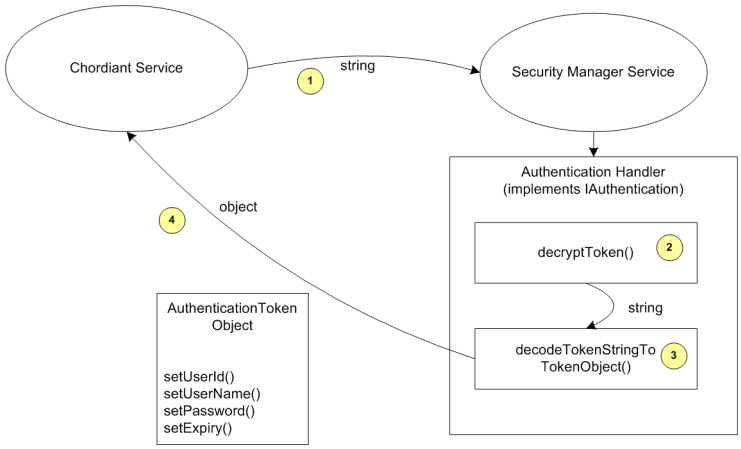


Figure 11-10: Checking the Authentication

To validate the Authentication Token:

1. The service passes the authentication token string to the Security Manager service.
2. The Security Manager calls the decryptToken method of the Authentication Handler. The decryptToken method converts the encrypted string into a clear text string. For additional information on this method, refer to page 279.
3. The decodeTokenStringToTokenObject turns the string into an AuthenticationToken object, which can then be deconstructed to determine if the user name, user ID, password, and expiration date are all acceptable. If there is a problem with any of these parameters or if the string cannot be decoded, the Security Manager throws an exception. For additional information on the decode method, refer to page 278.
4. The AuthenticationToken object is returned to the service.

CUSTOMIZING THE AUTHENTICATION HANDLER

During startup, Chordiant 5 Foundation Server loads the Authentication Handler specified in the `SecurityManager.xml` configuration file.

By default, Foundation Server uses LDAP as the external security server for authentication. You can customize the Authentication Handler to use any other security server you have in your enterprises, such as IBM SecureWay Security Server (RACF).

You can also customize the encryption and decryption of the Authentication Token.

The Chordiant 5 Foundation Server authentication module can be customized by implementing the interface com.chordiant.core.security.IAuthentication.

```
Public class MyAuthenticationHandler implements IAuthentication {
                                    …
 };
```

To specify the customized class:

1. Update the default value (`com.chordiant.core.security.AuthenticationHandler`) of the authentication.provider tag in the `SecurityManager.xml` configuration file with your new class name.

```
<Tag>authentication.provider
       <Value>com.custom.security.MyLDAPAuthenticator</Value>
   </Tag>
```

Code 11-2: Portion of SecurityManager.xml Configuration File showing authentication.provider Tag

Implement all of the methods that are required by the IAuthentication interface:

- Authenticate, createTokenObject, encode, and encrypt methods are also described in "Creating an Authentication Token" on page 275.
- Decode and decrypt methods are also described in "Validating an Authentication Token" on page 276.

1. **Authenticate Method**

   public int authenticate(Object userCredential, String securityCredential)

   This method authenticates a user based on the user's login credential and security credential. Definitions of user credential and security credential are implementation-specific. For example, for a password-based authentication system, the user credential could be the user's login name and the security credential could be the user's password.

   This method should return an integer value that is defined in the Java interface com.chordiant.core.security.AuthenticationHandlerConstants:

   - AUTHENTICATION_SUCCESS: authentication success
   - INVALID_CREDENTIALS: authentication failed due to invalid password
   - NO_SUCH_OBJECT: authentication failed due to invalid login name
   - PASSWORD_EXPIRED: authentication failed due to expired password
   - USER_LOCKED_OUT: authentication failed due to user lock-out

2. **createTokenObject Method**

   public AuthenticationToken createTokenObject()

   If the call to the authenticate method is successful, the createTokenObject method acts as a factory to create the actual AuthenticationToken. The returned AuthenticationToken object is populated by the Security Manager and fed into the encode method, which is described next.

3. **Encode and Decode Methods**

   ```
   public String encodeTokenObjectToTokenString( Object AuthenticationToken )
   ```

   ```
   public AuthenticationToken decodeTokenStringToTokenObject( String string )
   ```

   These two methods work together and are highly customizable.

   The encode method encodes the AuthenticationToken object obtained from the "createTokenObject Method"into a string. This string can be further encrypted, if desired, before it is returned to the SecurityMgrBeanClientAgent. Refer to "Encrypt and Decrypt Methods" for details on encryption.

   The decode method is used when a service needs to verify whether the AuthenticationToken string it has received from a client agent is valid. The string must be decoded into its AuthenticationToken object, from which the user id, user name, password, and expiration can be determined. If there is a problem with any one of these parameters, or if the string cannot be decoded at all, the Security Manager service throws an exception.

   It is essential that you are able to decrypt and decode the string returned from the encode and encrypt methods into a usable AuthenticationToken object. If you choose to encrypt the encoded string, you can use two-way encryption algorithms for the encrypt and decypt methods.

Tip: You can choose to use a highly secure database instead of two-way encryption. The encode method can store the AuthenticationToken object in the database, it returns a string as a unique key. This string does not contain any sensitive data, such as user names or passwords, so you can pass this string without worry. The decoding process involves using this key to locate the actual AuthenticationToken object in the database.

4. **Encrypt and Decrypt Methods**

```
public String encryptToken( String clearTxt )
```

```
public String decryptToken( String theCypherText )
```

The encrypt method is used to further encrypt the AuthenticationToken string created by the encode method. The decrypt method is used to reverse any encryption performed by the encrypt method.

You can decide that the encode method provides enough security for your needs. In this case, your encrypt method does not need to further manipulate the string value it is passed — it can just pass that string right through to the SecurityMgrBeanClientAgent.

You can, on the other hand, decide to add additional encryption to the string. These methods are highly customizable, so you can add any algorithm you choose.

The encryption algorithm you choose should generate an XML-safe string (that is, strings that are safe to pass to an XML SAX parser, such as letters and numbers). Otherwise, the token cannot be put in an XML document for web service usage.

As with the encode and decode methods, *it is essential* that you are able to decrypt the string that you encrypted.

5. **Service Control Functionality**

```
public void setup()
```

This method is called by Foundation Server at system startup time. The customized implementation can use this method to perform customized initialization tasks.

```
public void shutdown()
```

This method is called by Foundation Server when the authentication service is being shut down. The customized implementation can use this method to perform customized termination tasks.

6. Optionally, the customized implementation could also implement the following interfaces. These two interface must be implemented if you support password grace period and use Chordiant Administration Manager to manage user profiles.
   - com.chordiant.core.security.IPasswordPolicy
   - com.chordiant.core.security.IAuthenticationAdmin

   The IPasswordPolicy interface defines a method for password-related functionalities. Chordiant Security Manager calls methods defined in this interface upon request from a client application for password-related functionality. If your customization requirement does not support password-related functionality, you do not have to implement this interface.

   Currently there is only one method defined in IPasswordPolicy interface:

   ```
   public int getPasswordGracePeriod( String username, String password )
   ```

   This method returns the password grace period, which is the number of seconds before the password expires.

   If your customization does not support password grace period, or if the authentication data source does not support password expiring functionality, this method should return NO_PASSWORD_CONTROLS.

   If the password has already expired, this method should return PASSWORD_EXPIRED.

   The IAuthenticationAdmin interface defines methods called by Chordiant Profile Manager (part of the Administration Manager tool) for authentication-related administration. These methods will only be called if the registered authentication handler implements the IAuthenticationAdmin interface. From the Profile Manager, you can change a user's password, add, and delete users from the authentication data store. If you do not want to use Chordiant Profile Manager to manage your authentication data in your security data source, your customized authentication implementation should not implement this interface.

Here are the methods in the IAuthenticationAdmin interface:

- These two methods are for changing a user's password from within the Profile Manager. If you will not be using the Profile Manager for your custom implementation, you can set these methods to always return true.

```
public boolean changePassword( String username, String oldpasswd,
    String newpasswd )
```

This method updates the user's password. It should return true if it successfully updated the password. Otherwise, it should return false.

The customized implementation can perform password sanity checking before making the update.

```
public boolean changePassword( String username, String newpasswd )
```

This method updates the user's password, but it does not require the user's old password. The underlying implementation needs to be able to gain access to update the user's password first (for example, by impersonating an administrator using the password provided in configuration files).

- These two methods are called by the Chordiant Profile Manager to add or delete a user from the underlying authentication data store. They are in effect when the Profile Manager's UPDATE_EXTERNAL_USER_DATA constant is set to true. If you will not be using the Profile Manager for your custom implementation, set this UPDATE_EXTERNAL_USER_DATA constant to false to make these methods inoperative (or simply not implementing the IAuthenticationAdmin interface):

```
public void addLogin( String username, String password )
```

This method allows Chordiant Profile Manager to synchronize the changes on the Chordiant database to the underlying authentication data storage.

```
public void deleteLogin( String username )
```

This method allows Chordiant Profile Manager to synchronize the changes on the Chordiant database to the underlying authentication data storage.

Update the default value (com.chordiant.core.security.AuthenticationHandler) of the authentication.provider tag in the `SecurityManager.xml` configuration file with your new class name.

For information on customizing the Authentication Handler to allow a single sign-on, refer to the *Chordiant 5 Foundation Server Customization Guide*.

Customizing the Authentication Token

Chordiant automatically creates an AuthenticationToken object when one is requested.

This AuthenticationToken object includes four parameters, as well as getters and setters for each:

- userid
- user name
- password
- expiration date

These four parameters are required and expected by the Security Manager service and by Chordiant services. If these parameters are not available, exceptions will be thrown.

In most cases, you can use the token created by Chordiant. If you require additional information, you can extend the token to add more parameters as well as their associated getters and setters. You can also further encrypt it through the encoding and encryption methods, described in Step 3 and Step 4 on page 279.

If you choose to encrypt the authentication token, the encryption algorithm you choose should generate an XML-safe string (that is, strings that are safe to pass to an XML SAX parser, such as letters and numbers). Otherwise, the token can not be put in an XML document for web service usage.

The AuthenticationToken object created by the createTokenObject method is specified in the token.object element in the `SecurityManager.xml` configuration file. By default, this value is com.chordiant.core.security.AuthenticationToken. If you choose, you can extend this class to add your own supplementary information. You must then specify your extended class in the `SecurityManager.xml` file.

Migrating Existing Security Configurations

For information on migrating existing security configurations, refer to the *Chordiant 5 Tools Platform Administration Manager Guide*.

Chapter 12

Request Server

The Request Server offers an advanced web application infrastructure, a configurable model of application execution, and a series of best practice recommendations that enable you to develop sophisticated HTTP-based applications. These applications can range from web self-service applications, in-house programs for customer service agents, remote branch applications, and even “headless” applications.

Figure 12-1 illustrates the Request Server model supported by the Chordiant 5 Foundation Server.

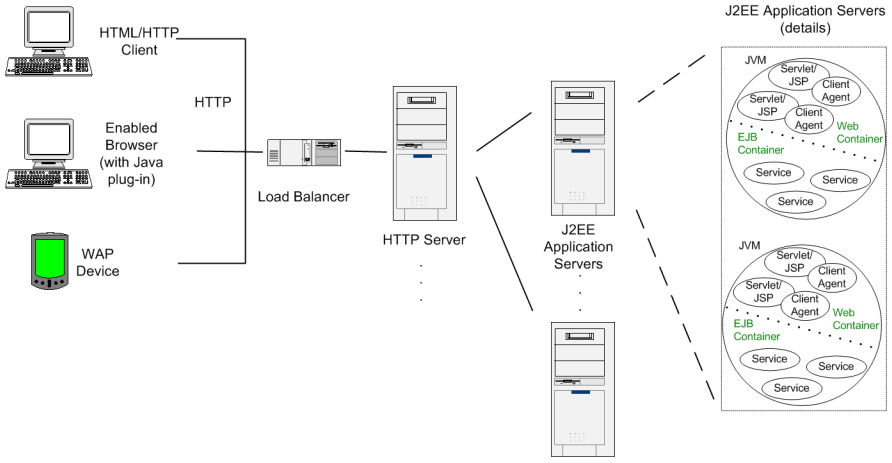


Figure 12-1: Request Server Overview

The Request Server model consists of the following components:

- **One or more J2EE application servers**

  Each physical server can host multiple application server replicates, which are the containers for the Chordiant 5 Foundation Server services, running as Enterprise Java Beans.

- **One or more web servers**

  The web server hosts the Foundation Server infrastructure, responsible for interacting with thin clients and serving as a bridge to the application servers.

- **Clients**

  These include HTML-based clients, browsers (with Java plug-in), Java applications, and mobile thin clients, such as wireless devices.

Using multiple servers offers a robust, fault tolerant, and load balanced execution environment for business applications. The system distributes client requests among application servers and application service replicates.

THE MAIN COMPONENTS

The Request Server includes several components that work together to provide the infrastructure and support for your web applications. The following is a list describing the main components of the Request Server:

- **Browser** — The generator of the HTTP Request. The "browser" can include a web browser, a thin client, a Java application, or a "headless" application.
- **Request Handler Servlet** — The server-based component responsible for accepting, handling, and forwarding the HTTP Request to other components within the system. The Request Handler Servlet acts as a dispatcher for the web application.
- **Request Context Map** — An XML-based file that encodes the context maps for the application, based on Action IDs, and maps requests to handlers.
- **Request Context Mapper Helper** — A server-based component that reads the Request Context Map.
- **Selectors Helper** — A server-based component that helps provide additional context to requests for mapping requests to selectors within the Request Context Map.
- **Device Context Mapper Helper** — A server-based component that creates selectors based on information about the device making the request. For example, a mobile thin client requires different treatment than a PC-based browser.
- **Chordiant Servlet Base Class** — A class that implements the baseline servlet functionality and performs the transformation of the presentation by calling the Transformation Helper. You will typically extend this class when creating your application logic servlets.
- **Application Logic Resource** — The application logic for the program. The Application Logic Resource runs on the application server, and interfaces with business services and other components of the system using the Chordiant 5 Foundation Server infrastructure. The application logic, which can be a servlet or a JavaServer Page (JSP), acts as a controller responsible for reading and writing business data using business services.
- **Servlets** — Servlets are modules that extend Java-enabled web servers and other request-response oriented servers. Servlets have no graphical user interface and handle client requests (HttpServletRequest) through a service method. The service method dispatches each request to a designated method, and generates an HTML string to return to the web client as an HttpServletResponse.
- **Transformation Helper** — Assists the Chordiant Servlet Base Class in the preparation of the presentation (HTML output) for the application. The Transformation Helper uses XSL-based stylesheets to output presentations, based on the specific requirements of the application.
- **Presentation Resource** — Generates the presentation, which can include HTML output and WML output. A presentation resource can be an XSL stylesheet, an HTML page, a JSP page, an XML-formatted page, dialogServer content, or a servlet.

THE EXECUTION FLOW

The Request Server consists of a coordinated execution of the main components of the web application infrastructure. This section describes the execution flow of the Request Server, with reference to Figure 12-2.

The execution flow for the web applications includes these steps:

Request Generated and Routed

1. An HTTP Request is generated on a client workstation or device.

   In many cases, the HTTP Request is generated by a user working on a thin client or web browser, including those hosted on devices such as wireless phones. However, the HTTP Request can also be generated by a "headless" client, such as a piece of software, without requiring human intervention.

2. The HTTP Request is routed to the Request Handler Servlet.

   The Request Handler Servlet is specified as part of the URL, submitted to the web server when the HTTP Request is generated. Included with the HTTP Request is a parameter specifying the Action ID, as illustrated in Code Sample 12-1.

```
http://{server-or-domain-name}/{application-path}/RequestHandler?
ActionID={action-id-value}
```

Code 12-1: HTTP Request

3. The Request Handler Servlet uses the Request Context Mapper Helper to find the context for the web application, based on the Action ID.

   Code Sample 12-2 is a segment of a Context Map file illustrating a context for a web application.

```
<ACTION_ID>
   HelloWorld
   <CONTEXT>
      <NAME>DEFAULT</NAME>
      <APP_LOGIC>/HelloWorldServlet</APP_LOGIC>
   </CONTEXT>
   <CONTEXT>
      <NAME>DEFAULT</NAME>
      <PRESENTATION>xsl/HelloWorld.xsl</PRESENTATION>
   </CONTEXT>
</ACTION_ID>
```

Code 12-2: Context Map File Segment Showing Web Application Context

Finding the Context

4. The Request Context Mapper Helper refers to information stored in the Request Context Map to locate the correct context, as specified by the Action ID.

   For more information about the Request Context Map, see "Exploring the Request Context Map" on page 291.

5. The Request Context Mapper Helper returns the value of the APP_LOGIC parameter for the appropriate context associated with the Action ID. The value of the APP_LOGIC parameter could be a servlet or a JavaServer Page. The servlet is then described and mapped in the `web.xml` file. See page 323 for more information.

Note: You can display a presentation resource using the ChordiantServletBaseClass as the application logic resource without having to define a custom application logic resource, for example, when no business logic is required to handle the request.

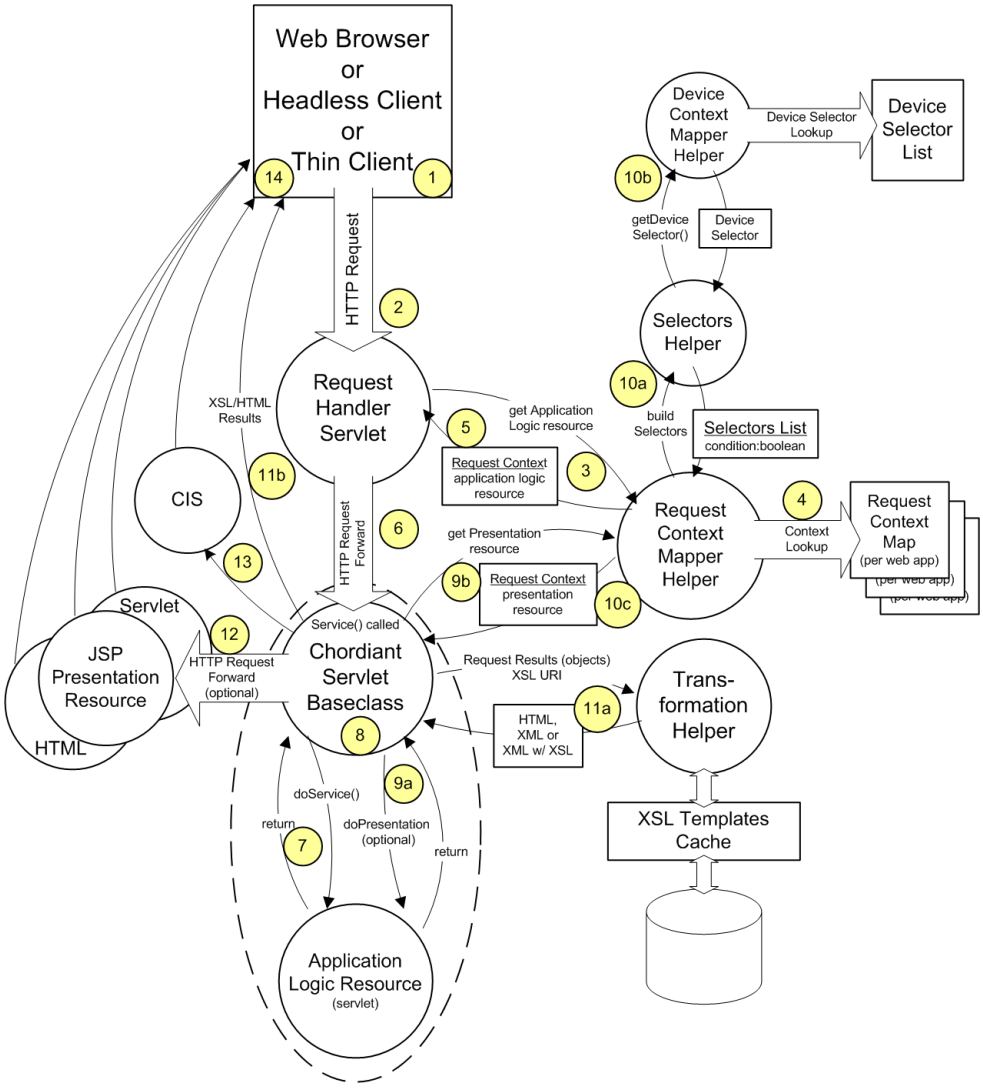


Figure 12-2: Handling Requests

Interacting with the Application Logic Resource

6. The customer-developed Application Logic Resource can take either of the following forms:
   - A servlet, derived from the ChordiantServletBaseClass
   - A JavaServer Page

   If the Application Logic Resource is a JSP page, the JSP page assumes responsibility of both the application logic and the presentation.

   If the APP_LOGIC parameter specifies a servlet which is described in the `web.xml` file, the Request Handler Servlet forwards the HTTP Request to the appropriate servlet, running as the Application Logic Resource.

   The custom application code is implemented in the doService method, which is called by the service method in the ChordiantServletBaseClass.

   The forwarded HTTP Request contains the Action ID and any form data collected from the original submitted form using the web browser, thin client, or headless client.

7. The doService method, running as the Application Logic Resource, interfaces with business services and databases to complete the useful work of the application.

   When interacting with business services, the Application Logic Resource uses the standard Foundation Server Architecture to complete the operations. This includes getting a Client Agent using a ClientAgentHelper, and communicating with the remote service using this Client Agent.

8. Once the doService method has completed, control returns to the service method in the ChordiantServletBaseClass.

9. The service method in the ChordiantServletBaseClass invokes the doPresentation method, if one is defined in the Application Logic Resource.

   The doPresentation method, if available, is responsible for generating the response output. Otherwise, the system uses the default implementation, which either uses the Presentation resource, if it is in the initial context (a), or invokes the Request Context Mapper (b) again to locate the appropriate presentation, identified by the PRESENTATION parameter, if it is not present in the initial context.

   The Request Server enables you to separate the application logic from the presentation of the output, and specify this distinction as part of the application context using the Request Context Map.

Finding the Device Context

10. The Request Context Mapper Helper uses the Selectors Helper (a) and the Device Context Mapper Helper (b) to determine the specific presentation to use, in cases when the presentation was not found during the initial lookup.

    You can include selectors in the Request Context Map to enable the system to select an appropriate presentation at run-time. Likewise, the system uses the Device Context Mapper Helper to distinguish between the various types of output devices.

    Code Sample 12-3 shows a segment of a Context Map file illustrating the use of context selectors.

```
<CONTEXT SELECTORS="MIME:XML;DEVICE:IE5.5">
    <NAME/>
    <TRANSFORM_TYPE>client</TRANSFORM_TYPE>
    <PRESENTATION>xsl/HelloWorldIE5.5.xsl</PRESENTATION>
</CONTEXT>
<CONTEXT SELECTORS="MIME:WML;DEVICE:NokiaWAPToolkit">
    <NAME/>
    <PRESENTATION>xsl/HelloWorldWML.xsl</PRESENTATION>
</CONTEXT>
```

Code 12-3: Context Map File Segment Showing Context Selectors

    This is useful because the presentation of the output can vary depending on the characteristics of the device, such as desktop PC versus a wireless phone. Similarly, the presentation can depend on whether the output is meant for a regular user or a headless client application.

    The request context containing the presentation resource is passed to the ChordiantServletBaseClass (c).

Creating the Presentation

The default implementation of the doPresentation method in the ChordiantServletBaseClass creates the presentation for the output, using one of the following mechanisms:

- Using the Transformation Helper to provide XSL Translation, creating to output or returning XML to the client. Described in Step 11.
- Forwarding to an HTML page, a JSP, or another servlet. Described in Step 12.
- Using Chordiant Interaction Server to forward HTML to the client. Described in Step 13.

11. If the presentation specifies an XSL-based stylesheet, the system uses the Transformation Helper to generate the results.

    The Transformation Helper uses data available through the business objects and performs a transformation using the XSL-based stylesheet (a). The system transforms Java objects from the Results object on the Request into an XML representation on which the XSL-based stylesheet operates. The transformed output is returned directly to the client or browser (b).

    For a headless or thin client where presentation results are not required, you can choose to have the Application Logic servlet return the string "true" or "false" as an indication of results. In other words, the response contents from an Application Logic Resource can provide meaningful results to the requester that are not necessarily directly presentable in a display.

    The Request Server offers the following XSL transformation types:

    — **None**: This results in XML content.

    — **Server**: This is the default when the presentation resource type is XSL. This XSL transformation is performed on the server.

    — **Client**: If the presentation resource is XSL, a reference to the XSL is inserted in the XML. This is useful for clients which can perform their own transformations. For example, Microsoft Internet Explorer 5.5 and higher can perform their own XSL transformations. However, versions earlier than IE 6.0 might not have the proper XML parser components installed. For instructions on obtaining the proper parser components, refer to "MSXML Parser" on page 337.

    — **dialogServer** is another available transformation type. See Step 13 for details.

12. Depending on the presentation for the application context, dispatching the request to any of the following:

    — an HTML page

    — another servlet

    — a JSP

      In the case of the JSP, the system does not use the Transformation Helper. The JSP Presentation resource has access to the business object (Application Logic results) through the HTTP Request and session, which was forwarded from the ChordiantServletBaseClass.

13. Dispatching the request to the Chordiant Interaction Server. The Chordiant Interaction Server performs "dialogServer" translation to produce HTML, which is then forwarded to the client.

Receipt by the Client

14. The browser or client device receives the output and handles it accordingly.

UNDERSTANDING REQUEST CONTEXT MAPPING

Chordiant 5 Foundation Server uses an XML-based file to encode the Context Maps for an application. A server-based component known as the Request Context Mapper Helper reads the Request Context Map and maps requests to handlers within your application.

This section describes the following topics related to the Request Context Mapping:

Application Logic and Presentation Resources

The system uses the following two resources when processing a typical request/response interaction between a browser and the Request Server:

- **An Application Logic Resource**

  An Application Logic Resource is the software component responsible for performing the useful work within the web application. You can implement an Application Logic Resource as either a servlet or as a JavaServer Page.

- **A Presentation Resource**

  A Presentation resource is the software component responsible for formatting the information in a style and structure that is suitable for a certain class of output device.

  In many cases, this involves formatting and structuring information based on known physical or logical characteristics of an output device, such as a wireless phone, a PC web browser, or a headless client application.

When you implement an Application Logic Resource as a JSP page, the page assumes the responsibility of both application logic and presentation (the contents of the HTTP Response). Servlets, on the other hand, normally provide only the application logic. In the case of servlets, the presentation is defined through a separate presentation resource, such as a JSP, XSL-based stylesheet, or Chordiant Interaction Server dialog.

You specify the mapping of a particular request, generated by the browser on the client device, using a Request Context Map.

Exploring the Request Context Map

When you create a web-based front end to a server-based application, you must establish a connection between the actions that users can perform through the user interface and the application logic and presentation resources of your application.

The Request Server enables you to specify this relationship between user actions and application logic and presentation resources through an XML-based file known as the Request Context Map.

Using Action IDs, you create contexts for the user action which define both the useful work (Application Logic Resource) as well as the structure and style of the output (the Presentation Resource). You can define several contexts for each Action ID, as required, together with a selector attribute which enables the system to determine the appropriate context dynamically based on run-time values.

By default, the Request Context Map is stored in *{application_path}*`/WEB-INF/ContextMap.xml`. However, you can modify this default location by specifying a parameter in the `Web.xml` application descriptor file. Code Sample 12-4 illustrates the Request Context Map format.

```
<ROOT>
   <ACTION_ID>
      MyActionId
      <CONTEXT>
         <NAME>DEFAULT</NAME>
         <APP_LOGIC>ServletName</APP_LOGIC>
      </CONTEXT>
      <CONTEXT SELECTORS="SelectorString">
         <NAME/>
         <TRANSFORM_TYPE>TransformType</TRANSFORM_TYPE>
         <PRESENTATION>XSL_StyleSheet</PRESENTATION>
      </CONTEXT>
   </ACTION_ID>
</ROOT>
```

Code 12-4: Format of Request Context Map

Table 12-1 describes the elements of the Request Context Map. Figure 12-5 on page 293 provides an example of a Request Context Map where you can see these tags used.

TAG	DESCRIPTION
<ACTION_ID> </ACTION_ID>	A string identifier for the user action.
<CONTEXT> </CONTEXT>	An application context, which defines the application logic resource, presentation resource, and transformation type. You can define multiple application contexts based on selectors, which can include MIME types and user agent values.

Table 12-1: Request Context Map Tags

TAG	DESCRIPTION
<NAME> </NAME>	The name of the context. You can assign the special identifier DEFAULT to indicate the default context.
<APP_LOGIC> </APP_LOGIC>	The logical name of the Application Logic Resource, which can be either a servlet or a JSP page. The servlet is then described and mapped in the web.xml file. See page 323 for more information on the web.xml file.
<PRESENTATION> </PRESENTATION>	A presentation resource can be an XSL stylesheet, an HTML page, a JSP page, a servlet, an XML-formatted page, or dialogServer content.
<TRANSFORM_TYPE> </TRANSFORM_TYPE>	The type of the Transformation Helper.The following are the valid values for this parameter: • client • server • none • dialogServer

Table 12-1: Request Context Map Tags (Continued)

Code Sample 12-5 illustrates a sample of a Request Context Map.

```
<ROOT>
   <ACTION_ID>
      HelloWorld
      <CONTEXT>
         <!--
         Application-logic context for all device- and mime-types.
         -->
         <NAME>DEFAULT</NAME>
         <APP_LOGIC>/HelloWorldServlet</APP_LOGIC>
      </CONTEXT>
      <CONTEXT>
         <!--
         Default presentation - i.e., basic server-side XSL transformation.
         -->
         <NAME>DEFAULT</NAME>
         <PRESENTATION>xsl/HelloWorld.xsl</PRESENTATION>
      </CONTEXT>
      <CONTEXT SELECTORS="MIME:XML;DEVICE:IE5.5">
         <!--
         Context specifying a transformation type "client."
         That is, give IE 5.5 the XML and a reference to
         the XSL and make the browser do the XSL transformation.
         -->
         <NAME/>
         <TRANSFORM_TYPE>client</TRANSFORM_TYPE>
         <PRESENTATION>xsl/HelloWorldIE5.5.xsl</PRESENTATION>
      </CONTEXT>
      <CONTEXT SELECTORS="MIME:WML;DEVICE:NokiaWAPToolkit">
         <!--
         Server-side XSL transformation of XML to WML content.
         -->
         <NAME/>
         <PRESENTATION>xsl/HelloWorldWML.xsl</PRESENTATION>
      </CONTEXT>
   </ACTION_ID>
</ROOT>
```

Code 12-5: Sample Request Context Map

Request Context Mapping Execution Flow

The Request Context Mapper Helper is responsible for mapping an HTTP Request, identified by an Action ID, to the contexts required to handle the request. The Request Context Mapper Helper uses information in the Request Context Map for the application, as well as the services of the Selectors Helper to the determine the context information for the application

This section describes the process of mapping an HTTP Request to the contexts needed to handle the request, as illustrated in Figure 12-3 on page 295.

Here is the execution flow of the Request Context Mapper **and** Selectors Helper**.**

1. Upon receiving an HTTP Request from the browser, the Request Handler Servlet calls the Selectors Helper.

   The Selectors Helper looks for selectors, in the form of HTTP Request parameters and attributes of the correct format, and builds the selectors HashMap.

2. The Selectors Helper calls the Device Context Mapper Helper to interpret device-specific information extracted from the HTTP Request.

   The Device Context Mapper Helper creates selectors based on information about the device making the request. Information about the device is stored in the HTTP Request header at the time the request is initiated by the user or a software agent.

   The Selectors Helper uses the Device Context Mapper Helper, which reads the `DeviceContextMap.xml` file to create device-based selectors.

3. The Request Handler Servlet calls the Request Context Mapper to determine the context for the request.

   The Request Context Mapper uses the Action ID and the HashMap of selectors created in Step 1 to retrieve the appropriate values for the context.

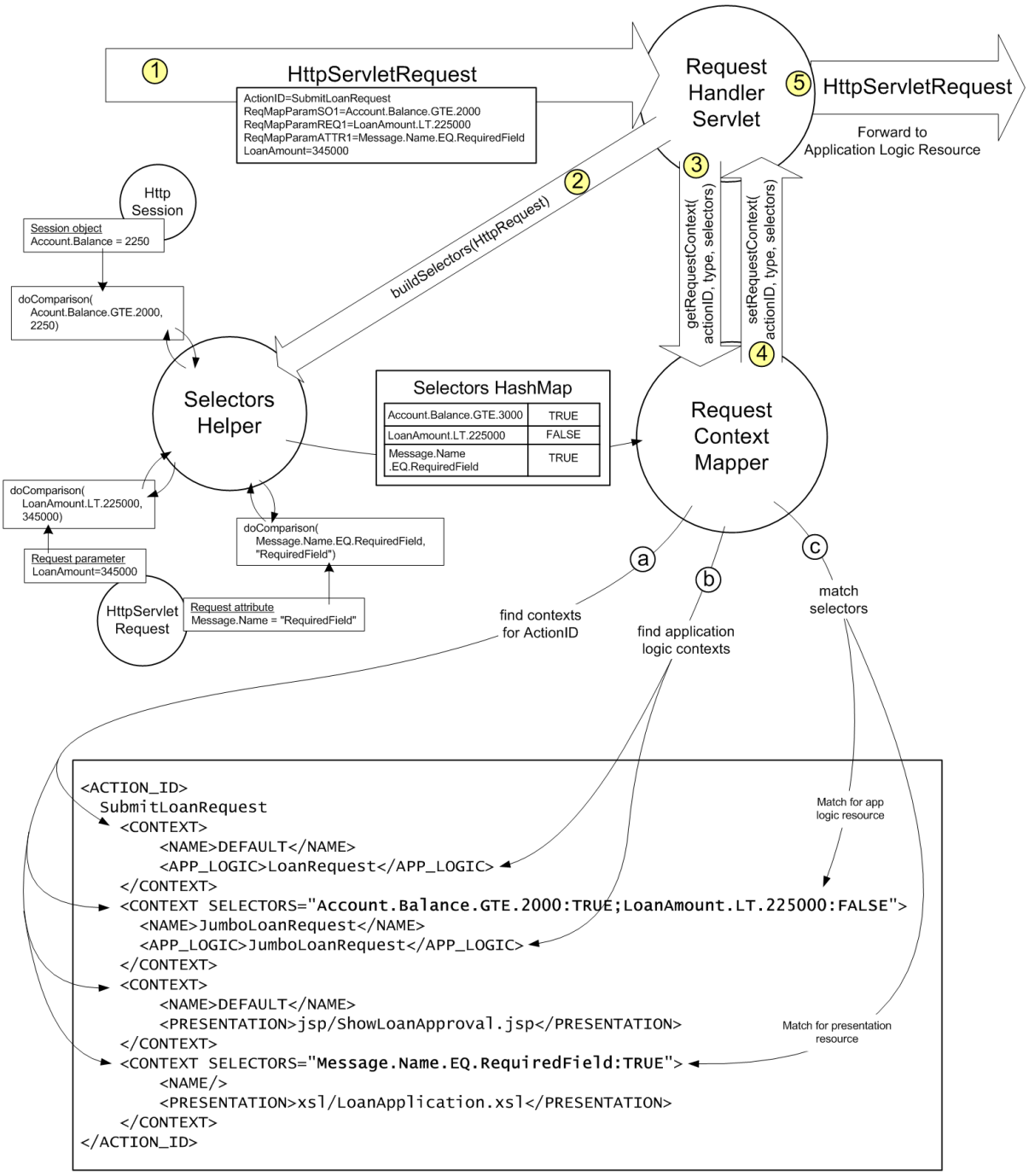


Figure 12-3: Mapping a Request to a Context

4. The Request Context Mapper returns the context to the Request Handler Servlet.

   The context is set as an attribute on the HttpServletRequest. Setting the entire context as an attribute precludes the need to perform a second mapping if a presentation resource is later requested. Likewise, additional elements in the context are made available to the system, should they be required later.

5. The Request Handler Servlet forwards the request (HttpServletRequest) to the Application Logic Resource.

   The Application Logic Resource can be either a servlet or a JSP page.

Understanding Selectors and the Selectors Helper

In addition to the Action ID, an application can influence the mapping of a request to its resources through the use of selectors. Selectors are optional attributes of a Context element, and consist of a String value of a specific pattern.

Note: If no selectors are provided with the request, the system selects the resource with the Name element value set to DEFAULT.

Selectors provide a dynamic mapping behavior that is sensitive to data from a variety of sources. The run-time value can be supplied in any combination of the following ways:

- **As an HTTP Request parameter**

  For example, the run-time value can be a value entered in an HTML form.

- **As an HTTP Request attribute**

  For example, the run-time value could be a discount percentage calculated from the order total of a shopping cart, expressed in BigDecimal form. Note that an HTTP Request attribute differs from an HTTP Request parameter in that an attribute can be an object of any type, whereas an HTTP Request parameter is always a String.

- **As a session object attribute value**

  For example, the run-time value could be the order total attribute of a shopping cart object stored in the current HTTP session.

- **Some combination of a user-agents list and MIME type**

  The source of these selectors can be generated by the device, or browser and the Context Device Mapper Helper.

Exploring the Parts of a Selector

Selectors include the following components:

- The HttpServletRequest parameter or attribute

  This is also known as the selector request.

- The context selectors attribute

  This is the attribute that appears in the Request Context Map file.

- The object (and attribute) or request parameter referenced by the selector request
- Operators

  The operators enable you to specify value comparisons.

Selector Request

The selector request can be one of the following:

- A ServletRequest parameter
- A ServletRequest attribute

ServletRequest parameters are contained in the query string or posted form data, and are always strings. An application can retrieve ServletRequest parameters from the request using the following method:

```
String ServletRequest.getParameter(java.lang.String name)
```

ServletRequest attributes, in comparison, are Java objects of any class. Your application can place ServletRequest attributes on the request using the following method:

```
void setAttribute(java.lang.String name, java.lang.Object o)
```

To be a selector request, the names must be of the proper form, namely include the prefix ReqMapParam.

Note that request parameters must come from the requesting device, such as an HTML form, while request attributes are supplied by the servlet code. This provides an opportunity for application logic code to inject selectors for the subsequent mapping to a presentation resource.

When the system builds the Selectors table, the Selectors Helper iterates over the HttpRequest attribute names and the HttpRequest parameter names. If the HttpRequest attribute object is not of type String, the system ignores it and flags it as an error.

Context Selectors

Selectors in the Context Map file are an attribute of the Context element.

```
<CONTEXT SELECTORS="Account.Balance.GTE.2000:TRUE;LoanAmount.LT.22500:FALSE">
```

You can specify more than one selector within a context, delimited by semicolons. If you specify more than one selector, all selectors must match. The Request Context Mapper Helper must match each piece of the selector string to a key/value pair in the selectors HashMap.

Referenced Object Attributes and Request Parameters

The object attribute or request parameter is the entity upon which the selector is based. For example, LoanAmount is the request parameter for the following request parameter selector.

```
ReqMapParamREQ1=LoanAmount.LT.22500
```

The Context Mapper is responsible for finding the object or request parameter that is referenced by the selector request.

Operators

Operators specify comparison operations within selectors. Table 12-2 outlines the operators you can use when building selectors.

OPERATOR	DESCRIPTION
EQ	Equal to
GT	Greater than
GTE	Greater than or equal to
LT	Less than
LTE	Less than or equal to
LIT	Literal. The Literal operator does not perform a comparison, but instead passes the value through to the mapping attribute.

Table 12-2: Selector Operators

Deferred Presentation Resource Mapping

The Request Context Mapper can defer the selection of the presentation resource until after the application logic resource has completed. This happens when a presentation resource was not found to be present with the Application Logic Resource during the initial lookup. This allows the application logic to influence the selection by placing objects in the HTTP Request or session.

For example, in the sample Context Map shown in Code Sample 12-6, the application logic code could add a LoginMessage object to indicate a failed login for the doLogin Action ID. If present, the Context would then map to a JSP page, which could present the message text and prompt the user to check the user name and password and try again.

```
<?xml version="1.0" encoding="UTF-8"?>
<ROOT>
    <ACTION_ID>
        doLogin
        <CONTEXT>
            <NAME>DEFAULT</NAME>
            <APP_LOGIC>/DoLoginServlet</APP_LOGIC>
        </CONTEXT>
        <CONTEXT>
            <NAME>DEFAULT</NAME>
            <PRESENTATION>/xsl/NetworkPresenceContainer.xsl</PRESENTATION>
        </CONTEXT>
        <CONTEXT SELECTORS="LoginMessage.MessageName.EQ.LoginMessage:TRUE">
            <NAME>LoginFailed</NAME>
            <PRESENTATION>/jsp/Login.jsp</PRESENTATION>
        </CONTEXT>
    </ACTION_ID>
</ROOT>
```

Code 12-6: Using Selectors

Using the range of selector types together with the deferred presentation resource mapping provides a powerful and flexible tool for personalization, error handling, and multi-channel support.

Building Selectors

The Selectors Helper reads the Request Context Map, looks for special, predefined servlet request parameter forms, and returns the processed values in a HashMap. To specify a selector for use in a mapping, the HTTP Request must contain one or more request parameters or attributes of the following forms (where [n] is typically an integer which makes the parameter name unique):

- ReqMapParamATTR[n]=RequestAttributeName.AttributeName.Operator.ComparisonValue

  This designates an HTTPServletRequest attribute object attribute.

- ReqMapParamREQ[n]=RequestParameterName.Operator.ComparisonValue

  This designates an HTTP Request parameter.

- ReqMapParamSO[n]=SessionObjectName.AttributeName.Operator.ComparisonValue

  This designates an HTTP Session object attribute.

Example 1

If you want to create a selector to check whether there is an account object in the HttpSession with a "balance" attribute having a value greater than or equal to 2000, you could use the selector request parameter, session object, and context selector outlined in Table 12-3.

ENTITY	VALUE
Selector Request Parameter	ReqMapParamSO1=Account.Balance.GTE.2000
Session Object	Object name: Account Attribute name: Balance
Context Selector	<CONTEXT SELECTORS="Account.Balance.GTE.2000:TRUE">

Table 12-3: Selectors Example 1

Example 2

If you want to create a selector to check whether an HttpRequest parameter with the name "LoanAmount" has a value less than 300000, you could use the selector request parameter, HTTP Request parameter, and context selector outlined in Table 12-4.

ENTITY	VALUE
Selector Request Parameter	ReqMapParamREQ1=LoanAmount.LT.300000
HTTP Request Parameter	Request parameter name: LoanAmount
Context Selector	<CONTEXT SELECTORS="LoanAmount.LT.300000:TRUE">

Table 12-4: Selectors Example 2

Example 3

If you want to create a selector to check whether an HttpRequest attribute object with the name "ErrorMessage" is found with a "MessageName" attribute and the value "RequiredField", you could use the selector request parameter, HTTP Request attribute, and context selector outlined in Table 12-5.

ENTITY	VALUE
Selector Request Parameter	ReqMapParamATTR1=ErrorMessage. MessageName.EQ.RequiredField
HTTP Request Attribute	Name: ErrorMessage Attribute name: MessageName
Context Selector	<CONTEXT SELECTORS="ErrorMessage. MessageName.EQ.RequiredField:TRUE">

Table 12-5: Selectors Example 3

Example 4

If you want to create a selector to place the country of a customer's home address into the mapping attribute, you could use the selector request parameter, HTTP Request parameter, and context selector outlined in Table 12-6.

ENTITY	VALUE
Selector Request Parameter	ReqMapParamREQ2 = CustomerHomeCountry.LIT.
HTTP Request Parameter	Request parameter name: CustomerHomeCountry
Context Selector	<CONTEXT SELECTORS="CustomerHomeCountry.LIT.:USA">

Table 12-6: Selectors Example 4

Understanding the Selectors Helper

The Selectors Helper looks for special, predefined name forms of servlet request parameters and request attributes, and returns their processed values in a HashMap. A HashMap object is always returned even though it might not contain any entries.

The SelectorsHelper class includes the following protected methods, used in this order, to complete the evaluation of the expression:

- **tryDateComparison**

  The tryDateComparison method attempts to convert the values to dates, using the convertStringToDate method, and perform a date comparison. The result parameter contains the comparison result as a the first array element, and returns a boolean with value true if the conversions and comparison worked correctly.

```
protected static boolean tryDateComparison(java.lang.String[] result,
    java.lang.String operator, java.lang.String value,
    java.lang.String comparator)
```

Code 12-7: tryDateComparison Method Signature

- **tryNumericComparison**

  The tryNumericComparison method attempts to convert the values to numerics and perform a numerical comparison. The result parameter contains the comparison result as a the first array element, and returns a boolean with value true if the conversions and comparison worked correctly.

```
protected static boolean tryNumericComparison(java.lang.String[] result,
    java.lang.String operator, java.lang.String value,
    java.lang.String comparator)
```

Code 12-8: tryNumericComparison Method Signature

- **doStringComparison**

  The doStringComparison method performs a comparison of two string values. The operator specifies how to compare the strings, and the result is returned as a string, such as "TRUE." Table 12-2 on page 298 describes the operators available for use with the doStringComparison method.

```
protected static boolean doStringComparison(java.lang.String[] result,
    java.lang.String operator, java.lang.String value,
    java.lang.String comparator)
```

Code 12-9: doStringComparison Method Signature

HashTable Examples

This section offers examples of entries created within the HashTable for specific selectors.

Example 1

Assume the following incoming request parameter:

```
ReqMapParamSO1=Account.Balance.GTE.10000
```

If there in an object in the HttpSession named Account that has a balance attribute with a value of 22501.63, the resulting HashMap entry would be:

```
key: Account.Balance.GTE.10000
value: TRUE
```

Example 2

Assume the following request-parameter selector:

```
ReqMapParamREQ1=LoanAmount.LT.225000
```

If an HttpRequest parameter with the name LoanAmount has the value of 345,000, the resulting HashMap entry would be:

```
key: LoanAmount.LT.225000
value: FALSE
```

Example 3

Assume the following request-parameter selector:

```
ReqMapParamATTR1=Message.Name.EQ.RequiredField
```

If an HttpRequest attribute with the name Message has a name attribute with the value of RequireField, the resulting HashMap entry would be:

```
key: Message.Name.EQ.RequiredField
value: TRUE
```

Understanding the Device Context Mapper Helper

The SelectorsHelper class uses the Device Context Mapper Helper to create selector values that identify devices based on the MIME type and user agent values.

The Device Context Mapper Helper collects candidate Device Contexts that have a MIME type value that matches any of the MIME types from the accept header of the requesting device. The Device Context Mapper Helper then returns the first candidate that has a matching user-agent header value.

Code Sample 12-10 illustrates a segment of a sample Device Context file.

```
<DEVICE>
    <NAME>Nokia WAP Toolkit</NAME>
    <DEVICE_TYPE>Developer's Tool Phone Simulator</DEVICE_TYPE>
    <MIME_TYPE>text/vnd.wap.wml</MIME_TYPE>
    <USER_AGENT_LIST>
        <USER_AGENT>Nokia-WAP-Toolkit/2.1</USER_AGENT>
    </USER_AGENT_LIST>
    <MIME_SELECTOR_VALUE>WML</MIME_SELECTOR_VALUE>
    <DEVICE_SELECTOR_VALUE>NokiaWAPToolkit</DEVICE_SELECTOR_VALUE>
</DEVICE>
```

Code 12-10: Sample Device Context Map

EXPLORING THE PRIMARY CLASSES

The Request Server provides a set of classes that implement a particular pattern for responding to HTTP Requests. These classes fall into the following two categories:

- The RequestHander and the ChordiantServletBaseClass Servlet Classes

  These classes implement the request handling pattern, processing the incoming request and returning the response. For more information about the ChordiantServletBaseClass, see "Using the ChordiantServletBaseClass" on page 307.

- The RequestContextMapperHelper and SelectorsHelper Classes

  These classes provide the connection between the infrastructure and application-specific features and functionality.

- The ApplicationInitializerServlet Class

  The init method of the ApplicationInitializerServlet provides a initialization point for the application and the associated infrastructure by calling the serviceControl method in the Static Helper with the following parameter:

  ```
  StaticHelperBaseClass.SERVICE_CONTROL_COMMAND_SETUP
  ```

- The ChordiantSessionHelper Class

  This is the default session helper implementation. The ChordiantSessionHelper class provides methods for getting, creating, and removing sessions.

  For more information about the ChordiantSessionHelper class, see "Using the Session Helper" on page 310.

- The LoginHelper Class.

  The LoginHelper class provides functionality for logging in and logging out users. The LoginHelper class uses the Security Helper to authenticate the user name and password and puts the user name and authentication token on the session.

  For more information about the LoginHelper class, see "Using the Login Helper" on page 311.

- The GenericDialogServerServlet Class

  The GenericDialogServerServlet class provides generic and extensible handling of dialogServer content. You can use the GenericDialogServerServlet in the following ways:

  — As a generic class for handling interactions in the Foundation Server environment, in situations where Chordiant Interaction Server displays are being used.

  In this case, the GenericDialogServerServlet class extracts inputs from a Chordiant Interaction Server display and preserves the inputs for the presentation context.

  — To handle any non-Chordiant Interaction Server display inputs.

  You can override the getInputs method to add any special processing that might be required by the execute method.

  For more information about how to use the GenericDialogServerServlet class, see "Integrating Foundation Server with Chordiant Interaction Server" on page 326.

- The RegisterNetworkPresence Class

  The implementation of the Thin Client command to register and deregister the Thin Client network presence. The RegisterNetworkPresence class handles an HTTP Request by completing:

  — Checks the following request parameter to determine whether to register or deregister.

```
REGISTER_NET_PRESENCE_PARAM_NAME
```

  — In the case of a request to register, the implementation retrieves the value of the following two HTTP Request parameters:

```
NETWORK_PRESENCE_KEY_PARAM_NAME
CONNECTION_URL_PARAM_NAME
```

  The implementation then calls the NameServiceHelper.rebind method. If no exception is thrown, the implementation sets the value of the HTTP Response contents to "true."

  — In the case of a request to deregister, the implementation retrieves the value of the following HTTP Request parameter:

```
NETWORK_PRESENCE_KEY_PARAM_NAME
```

  The implementation then calls the NameServiceHelper.unbind method. If no exception is thrown, the implementation sets the value of the HTTP Response contents to "true."

Using the ChordiantServletBaseClass

The ChordiantServletBaseClass supplies the basic functionality for the Application Logic Resource servlets that you create when developing applications for Chordiant 5 Foundation Server. By deriving your class from the ChordiantServletBaseClass, you can participate in the Chordiant application server architecture and use the recommended patterns and common capabilities.

You must implement the doService method and, optionally, the doPresentation method of any classes that you derive from ChordiantServletBaseClass. For more information about the execution flow of Foundation Server-based web applications, or about how to create web applications, see "The Execution Flow" on page 285 and "Building Web Applications" on page 316 respectively.

In addition to the doService and doPresentation methods, you can use the following methods within the ChordiantServletBaseClass when developing your servlets:

- getErrorMessageStringResource — This method retrieves an error message string from a property resource file. You can use the getErrorMessageStringResource method to incorporate custom error messages in your application. To do so, complete the following steps:
  - — Create a PropertyResourceBundle class to load the properties file.
  - — Define the resource string name constants in the resource bundle class.
  - — Create a properties resource file with the error messages.
  - — Call the getErrorMessageStringResource method from your application to retrieve an error message string.

```
protected static java.lang.String
getErrorMessageStringResource(java.lang.String resourceBundleName,
java.lang.String errorMessageResourceName)
```

Code 12-11: getErrorMessageStringResource Method Signature

The resourceBundleName is the name of the bundle in which the string is located, while errorMessageResourceName is the key of the resource string.

Code Sample 12-12 illustrates a sample resource bundle class.

```
public class MyAppErrorMessagesResourceBundle extends PropertyResourceBundle {

protected final static
    String ERROR_MESSAGES_RESOURCE_FILE_NAME =
    "MyAppErrorMessagesResourceBundle.properties";
    public MyAppErrorMessagesResourceBundle(InputStream stream)
    throws IOException {
        super(MyAppErrorMessagesResourceBundle.
            class.getResourceAsStream
            (ERROR_MESSAGES_RESOURCE_FILE_NAME));
     }

    public final static String MY_ERROR_MESSAGE_EMRS =
        "MyErrorMessageEMRS";
}
```

Code 12-12: Sample Resource Bundle Class

Code Sample 12-13 illustrates sample using the resource bundle class.

```
String errorMessage = getErrorMessageStringResource(
    "com.mycodepackage.MyAppErrorMessagesResourceBundle",
     MyAppErrorMessagesResourceBundle.MY_ERROR_MESSAGE_EMRS);
```

Code 12-13: Retrieving an Error Message

Code Sample 12-14 illustrates a properties resource file, stored in the MyAppErrorMessagesResourceBundle.properties file:

```
MyErrorMessageEMRS = This is my error message resource string.
AnotherErrorMessageEMRS = This another error message resource string.
```

Code 12-14: Sample Properties Resource File

Note that the format of the file consists of separate lines, each containing an entry of the form name=message.

You should name the resource file the same as the resource bundle class file, and append the extension "properties." For example, you could use the following resource file name: MyAppErrorMessagesResourceBundle.properties. When calling this method, pass in the full package and class name as the resourceBundleName parameter.

- getRequestResults — This method retrieves an object from the request with the name defined by the constant REQUEST_RESULTS_OBJECT_NAME. Typically, this is a Hashtable containing result objects, such as data objects from the service APIs. However, the item retrieved can be any object explicitly placed there by the application logic servlet, such as an org.w3c.dom.Document, for example.

  Note that the getRequestResults method creates a Hashtable if no object already exists.

```
protected static java.lang.Object
getRequestResults(javax.servlet.http.HttpServletRequest request)
```

Code 12-15: getRequestResults Method Signature

- getSession — The getSession method uses the SessionHelper class to provide the method for getting the session. For more information, refer to "Using the Session Helper" on page 310.

  The getSession method uses the SessionHelper to get the HttpSession object for the request. The method provides a single point for this class to access the session.

```
protected static javax.servlet.http.HttpSession
getSession(javax.servlet.http.HttpServletRequest request)
```

Code 12-16: getSession Method Signature

Code Sample 12-17 illustrates the use of the getSession method.

```
HttpSession session = getSession(request);
if (session != null) {
    session.setAttribute(ResultsHelper.TAB_TO_SHOW,
        ResultsHelper.MONITOR_TAB_LAYER_NAME);
}
```

Code 12-17: Using the getSession Method

- getAuthenticationToken — The getAuthenticationToken method accesses the session object that the LoginHelper added to the session when the user was successfully logged in. For more information about the LoginHelper class, see "Using the Login Helper" on page 311.

```
public static java.lang.String getAuthenticationToken(
    javax.servlet.http.HttpServletRequest request)
```

Code 12-18: getAuthenticationToken Method Signature

Code Sample 12-19 illustrates the use of the getAuthenticationToken method.

```
String authenticationToken = getAuthenticationToken(request);
```

Code 12-19: Using the getAuthenticationToken Method

- getUserNameFromSession — The getUserNameFromSession method accesses the session object that the LoginHelper added to the session when the user was successfully logged in. For more information about the LoginHelper class, see "Using the Login Helper" on page 311.

```
public static java.lang.String getUserNameFromSession(
    javax.servlet.http.HttpServletRequest request)
```

Code 12-20: getUserNameFromSession Method Signature

Code Sample 12-21 illustrates the use of the getUserNameFromSession method.

```
String userName = getUserNameFromSession(request);
```

Code 12-21: Using the getUserNameFromSession Method

Note: We encourage you to use the addToResults (see page 312) and setResultObject convenience methods to assist in processing request results.

Using the Session Helper

The SessionHelper class provides APIs for session management. The methods are defined by the SessionHelperInterface. The methods in this class are passthroughs to an implementation of the interface as defined by the configuration section CONFIG_SECTION_SESSION_HELPER and the configuration item CONFIG_ITEM_SESSION_HELPER_INSTANCE. The SessionHelper class ensures that the implementing class is instantiated when needed.

The ChordiantSessionHelper class is the default implementation of the Session Helper, offering implementations of the following methods:

- ensureSessionExists — Determines if a session exists. If no session exists, the ensureSessionExists method creates one. Note that you should call this method before any output is written to the response.

```
public boolean ensureSessionExists(javax.servlet.http.HttpServletRequest request)
```

Code 12-22: ensureSessionExists Method Signature

The ensureSessionExists method returns false if a session could not be found or created.

- getSession — Retrieves the HttpSession object for the request. The getSession method does not create a session if one does not already exist.

```
public javax.servlet.http.HttpSession getSession(javax.servlet.http.HttpServletRequest
request)
```

Code 12-23: getSession Method Signature

Code Sample 12-24 illustrates the use of the getSession method.

```
HttpSession session = getSession(request);
if (session != null) {
    // Use the session information, as appropriate
}
```

Code 12-24: Using the getSession Method

- removeSession — This method invalidates the session.

```
public boolean removeSession(javax.servlet.http.HttpServletRequest request)
```

Code 12-25: removeSession Method Signature

The removeSession method returns false if an error occurred while trying to invalidate the session.

Using the Login Helper

The LoginHelper class provides the functionality for logging in and logging out users, and includes implementations of the following methods:

- doLogin — Authenticates a user and puts the user name and authentication token on the session. The doLogin method returns null if successful. Otherwise, if the user name and password are not accepted, the doLogin method returns an error message.

  Note that if the user name or password is null, the method returns a string with an error message.

```
public static java.lang.String doLogin(javax.servlet.http.HttpServletRequest request)
```

Code 12-26: doLogin Method Signature

Code Sample 12-27 illustrates the use of the doLogin method.

```
String errorMessage = LoginHelper.doLogin(request);

if (errorMessage != null) {
    ApplicationUserMessage message = new ApplicationUserMessage();
    message.setMessageName("LoginMessage");
    message.setMessage(errorMessage);

    // Make login failure message available to context mapper and JSP.
    request.setAttribute("LoginMessage", message);
    request.setAttribute("ReqMapParamATTR1",
        "LoginMessage.MessageName.EQ.LoginMessage");
    throw new GoBackException();
}
else {
    // Perform some useful work
}
```

Code 12-27: Using the doLogin Method

- doLogout — This method logs a user out of the system, and invalidates the session. The doLogout method throws the javax.servlet.ServletException and java.io.IOException.

```
public static java.lang.String doLogout(javax.servlet.http.HttpServletRequest request)
throws javax.servlet.ServletException, java.io.IOException
```

Code 12-28: doLogout Method Signature

Code Sample 12-29 illustrates the use of the doLogout method.

```
errorMessage = LoginHelper.doLogout(request);

if (errorMessage == null) {
    // Successfully logged out.
    errorMessage = getErrorMessageStringResource(null,
        ApplicationErrorMessagesResourceBundle.
        LOGOUT_SUCCESS_MESSAGE);
}
```

Code 12-29: Using the doLogout Method

UNDERSTANDING APPLICATION LOGIC RESULTS

Application Logic Resources, specifically classes derived from ChordiantServletBaseClass, designate their results (the data that is available for display or that is to be included in the response to the request) by placing objects on the request as an HttpServletRequest attribute.

The ChordiantServletBaseClass defines the name of the request attribute,.

```
ChordiantServletBaseClass.REQUEST_RESULTS_OBJECT_NAME
```

You can place objects on the request as an HttpServletRequest attribute using the base class method addToResults method. You can use the addToResults method to collect multiple result objects, as required.

```
public static void addToResults(javax.servlet.http.HttpServletRequest request,
java.lang.String name, java.lang.Object object)
```

Code 12-30: addToResults Method Signature

The addToResults method creates a Hashtable for the container the first time this method is used in each request. Code Sample 12-31 illustrates a sample use of the addToResults method.

```
Object someBusinessObject = doSomeBusinessLogic();
addToResults(request, "MyBusinessObject", someBusinessObject);
```

Code 12-31: Using the addToResults Method

Note: When using the addToResults method, you must not include any spaces in the string you pass to the method. The string argument must consist only of letters and underscores. Embedded spaces in the string produces unpredictable results.

Alternatively, you can place objects on the request as an HttpServletRequest attribute directly through the HttpServletRequest.setAttribute method, using the attribute name as defined.

Using the HttpServletRequest.setAttribute method directly might be useful in cases where the result object is an org.w3c.Document object, for example. Note that once an object other than a Hashtable is set as the result object, no other objects can be added to the results.

Note: The org.w3c.Document is only supported by the ObjectTreeToDOM transform type SimpleLiteral. For more information, see the ObjectTreeToDOM configuration section in the `TransformHelper.xml` configuration file.

You can then use the getAttribute method to reference the Application Logic Result objects, on a JSP page, for example, using this code.

```
request.getAttribute("RequestResults");
```

Examining the XML Instance Document

You can use the debugging features to examine the form of the XML instance document that is produced when the presentation type for the current context is a server-side XSL transformation. Alternatively, you can set the transform type of the presentation to client to have the XML returned directly to the browser.

You can activate debugging for the TransformHelper component by creating a new log filter configuration section. Code Sample 12-32 illustrates how to activate debugging for the TransformHelper component. You might find it convenient to keep this new section in a separate file that you can place in the configuration directory as needed.

```
<Root>
   <Section>Log
      <Tag>Filter
         <Value>FilterTwo</Value>
      </Tag>
   </Section>
   <Section>LogConfiguration
      <Tag>LOG_DEBUG_ON
         <Value>true</Value>
      </Tag>
   </Section>
   <Section>FilterTwo
      <Tag>filterclass
         <Value>com.chordiant.core.log.LogFilter</Value>
      </Tag>
      <Tag>criteria
         <Value>com.chordiant.core.transform</Value>
      </Tag>
      <Tag>level
         <Value>debug</Value>
      </Tag>
      <Tag>writer
         <Value>com.chordiant.core.log.LogWriterStandardOut</Value>
      </Tag>
   </Section>
</Root>
```

Code 12-32: Activating the Debugger

Understanding Exception Handling

You should include exception handling in your applications to make it easier for users to determine when things are not functioning as expected, and to provide feedback on alternate ways to complete their work.

The Request Server offers the following capabilities to help you include exception handling in your applications:

- Pre-defined exceptions
- ErrorPage Action ID
- ErrorHandlingMechanism configuration parameter

Table 12-7 describes the types of exceptions available when developing web applications.

EXCEPTION	DESCRIPTION
GoBackException	Throw this exception when your application detects an error that can be remedied by returning to the previous page. Note that if a page is likely to be the target of a "GoBack," you must code the page to handle displaying error messages and related functionality.
SendErrorException	Throw this exception when your application detects an unrecoverable error. When you throw the SendErrorException, the system invokes error handling routines. Alternatively, you can invoke the sendError method, which throws the SendErrorException, to handle an unrecoverable error.

Table 12-7: Web Application Exceptions

To include exception handling in your application:

1. Include logic in your servlet to detect errors, as appropriate.
2. In the case of errors that can be remedied through user intervention, include code in your servlet to throw a GoBackException.

   Throwing a GoBackException instructs the system to re-display the current page, enabling users to modify inputs to the page, or perform other operations to remedy the error. For example, in the case of invalid input, the system can display an error message and offer users an opportunity to retry the operation using different data.
3. When using a GoBackException, include logic within the page to handle displaying error messages.

   You must also store any error messages on the session or request before throwing the GoBackException. The system does not store error messages; your application is responsible for performing this work.

4. Check whether a servlet is being invoked because of a GoBackException.

   The Request Server offers the following flag available on the session object enabling you to determine if the servlet is being invoked because of a GoBackException.

   ```
   ChordiantServletBaseClass.HANDLING_GO_BACK_FLAG_SESSION_OBJECT_NAME
   ```

   The flag is null in cases when the servlet is not being invoked as part of a GoBackException.

   Your servlet should check whether it is being called as part of a GoBack operation. The Request Server detects the case of an attempt to GoBack while handling an existing GoBack operation. If this is the case, the system invokes the ErrorPage Action ID. By checking for this condition within your servlet, you can modify the way in which the condition is handled.

   You can use code similar to Code Sample 12-33 to determine whether the servlet is being invoked through a GoBackException.

   ```
   Boolean doingGoBack = theSession.getAttribute(
       ChordiantServletBaseClass.HANDLING_GO_BACK_FLAG_SESSION_
       OBJECT_NAME);
       . . .
       // Perform parameter validation, if required
       if ( !isValidNumber(inputValue) ) {
           // Check whether we are performing a GoBack
           if (doingGoBack != null && doingGoBack.booleanValue() ) {
               // Process the field as needed, but don't throw a
               // GoBackException.
           }
           else { // Not handling a GoBack; perform normal error handling
               // Process as an error, and throw GoBackException if required
           }
       }
   ```

   Code 12-33: Determining if GoBackException is Used

5. In the case of unrecoverable errors, include code in your servlet to throw a SendErrorException.

   Alternatively, you can invoke the SendError method to throw the exception. The SendError method packages the error message and the error code on to the exception before it is thrown.

6. When using a SendErrorException, your application should define a custom servlet for the ErrorPage Action ID.

   The system provides a pre-defined ErrorPage Action ID that invokes a simple servlet to display an error message to the browser listing the attributes and parameters stored on the session and request objects.

   You can define your own ErrorPage Action ID in the Context Map file to override this servlet to perform actions more appropriate for your application. For example, in the case of a severe error, your application could display the initial login screen with a message instructing users about the nature of the error and requesting them to login and start over.

7. When using a SendErrorException, optionally configure the ErrorHandlingMechanism.

   The ErrorHandlingMechanism parameter is in the WebToolkit.xml configuration file, and enables you to control the behavior of the handleSendErrorException method. The following are the valid values for the ErrorHandlingMechanism parameter:

   — **ErrorPage:** Forwards the request to the ErrorPage Action ID.

   — **SendError:** Uses the response.sendError method to report errors to the browser. The response.sendError method enables you to display generic browser error messages.

8. When overriding the doPresentation method of the ChordiantServletBaseClass, save the current Action ID as the GoBack target and clear the GoBack session object.

   You can override the doPresentation method to implement customized presentation handling.

   Code Sample 12-34 illustrates how to save the current Action ID and clear the GoBack session object.

```
// Save the request for go back and clear the flag to show we're NOT
// in the midst of doing a 'go back'.
HttpSession theSession = getSession(request);
if (theSession != null) {
    theSession.setAttribute(
        PREV_ACTION_ID_SESSION_OBJECT_NAME, actionID);
    theSession.removeAttribute(
        HANDLING_GO_BACK_FLAG_SESSION_OBJECT_NAME);
}
```

Code 12-34: Saving the Action ID and Clearing the GoBack Session Object

BUILDING WEB APPLICATIONS

Chordiant provides an advanced infrastructure for creating web applications: the Request Server. When developing a web application, you must complete a series of steps related to planning the business operations and services, designing the flow of the application, and programming to the available interfaces.

This section describes the following topics related to building web applications to run on the Foundation Server:

- "Understanding Developer Goals" on page 317
- "Example of Building an Application Logic Resource" on page 319

Understanding Developer Goals

Before you begin creating your web application, you should complete a series of steps related to determining the required business services and mapping them the application logic that will appear in your application code.

Table 12-8 outlines the developer goals and procedures for creating web applications for Foundation Server. Refer to the *Chordiant 5 Tools Platform Getting Started Guide* for more information.

DEVELOPER GOAL	PROCEDURE
Creating new and customized business services	You should complete the following tasks: • Analyze the Enterprise Offerings and Business Processing Rules • Identify the required business objects and Business Services • Develop new (or customize existing) Business Services
Creating Action IDs	You should complete the following tasks: • Analyze the Enterprise Offerings and Business Processing Rules • Identify the required business objects and Business Services • Combine information to create preliminary Application Logic Resources, including identification of the inputs and outputs, and proposed names • Create the new Application Logic Resource in Java
Creating Java Client Agents	You must complete the following tasks: • Analyze the Enterprise Offerings and Business Processing Rules • Identify the required business objects and Business Services • Create the Java Client Agents for new Business Services with stubbed APIs • Create Java Client Agents for new Business Services

Table 12-8: Developer Goals

DEVELOPER GOAL	PROCEDURE
Create the Request Context Map file	You should complete the following tasks: • Analyze the Enterprise Offerings and Business Processing Rules • Identify the offerings • Create the HTML storyboard (page flow) • Identify necessary Application Logic Resources names for each page transition • Identify necessary Action IDs and mapping attributes • Convert the HTML to XSL, and create an HTML storyboard in preliminary XSL form (with references to sample XML data) • Test and modify the graphical user interface using Internet Explorer 5.5 or higher • Identify top-level XSL file names • Enter the following information where appropriate into the Request Context Map file: Action IDs and mapping attributes, Application Logic Resources, top-level XSL file names
Create the HTML storyboard in final XSL form (with references to final XML contexts or data templates)	You should complete the following tasks: • Analyze the Enterprise Offerings and Business Processing Rules • Create the HTML storyboard (page flow) • Convert the HTML to XSL, and create an HTML storyboard in preliminary XSL form (with references to sample XML data). • Test and modify the graphical user interface using Internet Explorer 5.5 • Modify the references to the final XML data templates, if required • Combine the XML data templates to create the final HTML storyboard

Table 12-8: Developer Goals (Continued)

Example of Building an Application Logic Resource

This section illustrates the steps you can follow to create an application logic resource for a single Action ID for use with Chordiant 5 Foundation Server.

To build a application logic resource:

1. Create your Application Logic Resource.

   If you are creating a servlet, define a class that extends the ChordiantServletBaseClass. Code Sample 12-35 illustrates how you can extend the ChordiantServletBaseClass.

```
package com.mywebapp;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.chordiant.application.ChordiantServletBaseClass;

public class myService extends ChordiantServletBaseClass {
   . . .
}
```

   Code 12-35: Extending the ChordiantServletBaseClass

   If you are creating a JSP page, you can skip forward to Step 7 on page 320 since you do not need to override methods in the ChordiantServletBaseClass, nor do you need to create a separate presentation resource, since this is handled in your JSP page.

2. Override the doService method in your derived class, and implement your application logic in this method.

   The service method in the ChordiantServletBaseClass invokes the doService method.

3. Use the addToResults method in your doService method to put results in the HttpServletRequest.

   Putting result objects in the HashTable object on the HttpServletRequest enables the application resource logic to make the objects available to the presentation resource. Code Sample 12-36 illustrates how you can put results in the HttpServletRequest using the addToResults method.

```
public void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    addToResults(request, "TestString", "It works!");
}
```

   Code 12-36: Using the addToResults Method

   For more information about using the addToResults method, see "Understanding Application Logic Results" on page 312.

4. Optionally, override the doPresentation method in your derived class, and implement the presentation handling.

   In most cases, you can rely on the code in the ChordiantServletBaseClass to perform the standard presentation services using the Presentation Context defined for the Action ID in the Context Map.

5. Implement the exception handling for the Application Logic Resource.

   The Request Server provides two pre-defined exceptions: the GoBackException and the SendErrorException. You should use the GoBackException for errors that users can resolve by returning to the previous page, and the SendErrorException or severe errors that perhaps requires further processing.

   For more information about handling exceptions in your web applications, see "Understanding Exception Handling" on page 313.

6. Create a Presentation Resource.

   A presentation resource can be an XSL stylesheet, an HTML page, a JSP page, a servlet, an XML-formatted page, or dialogServer content.

   For example, you can create an XSL stylesheet to serve as the Presentation Resource. Code Sample 12-37 is an example of an XSL stylesheet.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="*"/>
<xsl:template match="ROOT_ELEMENT">

<!-- !DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" -->

    <html>
    <head>
        <title>ShowThinger</title>
    </head>

    <body bgcolor="White" leftmargin="0" topmargin="0" marginwidth="0" marginheight="0">

    <xsl:value-of select="TestString"/>

    </body>
    </html>

</xsl:template>
</xsl:stylesheet>
```

Code 12-37: Sample XSL Stylesheet

7. Create the Context Map for the application.

   You specify the actions implemented by your application using the XML-based Request Context Map file. For each action, you can specify one or more contexts containing information about the Application Logic Resource, Presentation Resources, and Transform

Types. Save the Context Map using the file name `ContextMap.xml`. You must include a description and mapping of the Application Logic Resource in the `web.xml` file. See Step 10 on page 323.

Code Sample 12-38 illustrates a sample Request Context Map file:

```
<?xml version="1.0" encoding="UTF-8"?>
<ROOT>
   <ACTION_ID>
      RetrieveThinger
      <CONTEXT>
         <NAME>DEFAULT</NAME>
         <APP_LOGIC>/RetrieveThingerServlet</APP_LOGIC>
      </CONTEXT>
      <CONTEXT>
         <!--
         Default presentation - i.e., basic server-side XSL transformation.
         -->
         <NAME>DEFAULT</NAME>
         <PRESENTATION>xsl/ShowThinger.xsl</PRESENTATION>
      </CONTEXT>
   </ACTION_ID>
</ROOT>
```

Code 12-38: Sample Request Context Map File

Note that in the sample, the first <CONTEXT> element contains an Application Logic Resource reference to the servlet implementing the action. The second context sets the value of the Presentation resource to an XSL stylesheet that describes how to prepare the HTML output for the web browser.

For more information about creating the Context Map file, see "Exploring the Request Context Map" on page 291.

8. Use selectors to define alternative presentation contexts, if required.

   For example, you could define an alternative presentation context using a selector called SIZE1, based on an object within your application called myThinger. Code Sample 12-39 illustrates additions you can make to the Context Map to define the selector:

```
<CONTEXT SELECTORS="myThinger.Size.EQ.1:TRUE">
   <!--  Context for alternate presentation  -->
   <NAME>SIZE1</NAME>
   <PRESENTATION>xsl/ShowThinger2.xsl</PRESENTATION>
</CONTEXT>
```

Code 12-39: Defining a Selector

For more information about defining selectors, see "Understanding Selectors and the Selectors Helper" on page 296.

9. Use the Chordiant Application Administrator to flush the Chordiant system cache.

   You must flush the Chordiant system cache, using the Chordiant Application Administrator, after you make modifications to any of the following:

   — Request Context Map

   — Device Context Map

   — Any XSL file

   You can use following commands, passed as query parameters to the Chordiant Application Administrator servlet, to administer the Application Server (note that the equals sign is required). You can also use the JX Admin application to initiate these same commands.

   — Flush Templates

   Flushes the cache of XSL Transformation Templates. To issue the Flush Templates command, use the following URL:

   ```
   http://[web _server_name]/servlets/com.chordiant.application.
   ApplicationAdminstrator?FlushTemplates=
   ```

   — Flush Request Context Map

   Flushes the Request Context Map from memory. To issue the Flush Request Context Map command, use the following URL:

   ```
   http://[web_server_name]/servlets/com.chordiant.application.
   ApplicationAdminstrator?FlushRequestContextMap=
   ```

   — Flush Device Context Map

   Flushes the Device Context Map from memory. To issue the Flush Device Context Map command, use the following URL:

   ```
   http://[web_server_name]/[applicationname]/servlets/com.chordiant.application.
   ApplicationAdminstrator?FlushDeviceContextMap=
   ```

   — Help

   Displays the usage patterns for the commands. To issue the Help command, use the following URL:

   ```
   http://[web_server_name]/servlets/com.chordiant.internet.servlets.
   CCSWebServerAdminServlet?Help=
   ```

10. Modify the `web.xml` file.

    The `Web.xml` application descriptor file enables you to supply application description information to the application server. Table 12-9 outlines the elements that you should modify in the `web.xml` file.

ELEMENT	DESCRIPTION	
<display-name>	Set the value to the name of your application.	
<description>	(Optional) A description of your application. Note that the Request Server does not use this parameter.	
<context-param>	<param-name> and <param-value>	Set the <param-name> element to "application-name", and the <param-value> to the name of your application. The Request Server uses this value to determine the application name. You can optionally add an additional context parameter to turn off device context mapping by setting the <param-name> element to UseDeviceMapping, and setting the <param-value> element to "false"
	<description>	(Optional) Set this value to describe the specific <context-param>.
<servlet>	The section that describes the servlet referred to in the ContextMap.xml document.	
	<servlet-name>	The name of the servlet.Must be identical to the name used in the ContextMap.xml document.
	<description>	(Optional) Set this value to describe the specific <servlet-name>.
	<servlet-class>	The fully-qualified path to the servlet class. For example, com.hello.world.application.HelloWorld

Table 12-9: Web.xml Parameters

ELEMENT	DESCRIPTION	
<servlet-mapping>	The section that describes the mapping for the servlet in the <servlet> section above.	
	<servlet-name>	The name of the servlet.Must be identical to the servlet name specified in the <servlet> section above
	<url-pattern>	The fully-qualified relative name that will invoke the servlet.

Table 12-9: Web.xml Parameters (Continued)

The `Web.xml` file also contains references to the ApplicationInitializer and the RequestHandler servlets.

The application initializer servlet initializes the Request Server and is set to load on startup. This servlet provides an initialization point for the application and the associated infrastructure. So it is important that at least one web application in the EAR has this servlet set to preload. For example, see the sample `Web.xml` file in Figure 12-41 on page 325.

The `Web.xml` file describes the servlet used in the `ContextMap.xml` file and creates a mapping of its URL pattern.

Code Sample 12-40 illustrates how you can use the `Web.xml` file to specify the location of the Context Map file for your application.

```
<context-param>
   <param-name>RequestContextMapFileURI</param-name>
   <param-value>file:C:\JX\MyWebApp\ContextMap.xml</param-value>
   <description>
   The initialization parameter specifying the location of the context map.
   </description>
</context-param>
```

Code 12-40: Specifying the Context Map File in Web.xml

Code Sample 12-41 illustrates a segment of a sample `Web.xml` configuration file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
   <display-name>My Web Application</display-name>
   <description/>
   <context-param>
      <param-name>application-name</param-name>
      <param-value>My Web Application</param-value>
      <description>
         The name of the application - basically, a way to get
         to the display-name from the servlets.
      </description>
   </context-param>
   <context-param>
      <param-name>UseDeviceMapping</param-name>
      <param-value>false</param-value>
      <description>
         A flag to specify whether to use
         the device context selectors.
         This overrides the component configuration
         file setting on a per-application basis.
      </description>
   </context-param>
   <!-- Begin Application Initializer Servlet -->
   <servlet>
      <servlet-name>ApplicationInitializer</servlet-name>
      <description>
         This servlet provides a initialization point for…
      </description>
      <servlet-class>
            com.chordiant.application.ApplicationInitializer
      </servlet-class>
      <!-- Load this servlet at server startup time -->
      <load-on-startup>5</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>
         ApplicationInitializer
      </servlet-name>
      <url-pattern>
         /InitApp
      </url-pattern>
   </servlet-mapping>
   <!-- End Application Initializer Servlet -->
   <!-- Begin Request Handler Servlet -->
   <servlet>
      <servlet-name>RequestHandler</servlet-name>
      <description>
         This servlet is the controller that handles…
      </description>
      <servlet-class>
         com.chordiant.application.RequestHandler
      </servlet-class>
   </servlet>
```

Code 12-41: Sample Web.xml Configuration File

```
    <servlet-mapping>
           <servlet-name>
                  RequestHandler
           </servlet-name>
           <url-pattern>
                  /req
           </url-pattern>
    </servlet-mapping>
    <!-- End Request Handler Servlet -->

<!-- Begin Hello World Servlet -->
    <servlet>
       <servlet-name>HelloWorldServlet</servlet-name>
       <description>
       </description>
       <servlet-class>
          com.hello.world.application.HelloWorld
       </servlet-class>
    </servlet>

    <servlet-mapping>
       <servlet-name>
          HelloWorldServlet
       </servlet-name>
       <url-pattern>
          /HelloWorldServlet
       </url-pattern>
    </servlet-mapping>
</web-app>
```

Code 12-41: Sample Web.xml Configuration File (Continued)

11. Compile the servlet.

INTEGRATING FOUNDATION SERVER WITH CHORDIANT INTERACTION SERVER

You can integrate Chordiant 5 Foundation Server applications with Chordiant Interaction Server and Chordiant Interaction Designer to streamline the way you create web-based forms and process business information submitted using these forms.

Using data model descriptions encoded with XML Schema Definitions (XSDs) and Java classes, Chordiant 5 Foundation Server enables you to easily interpret and repurpose information contained in either XSD instances or Java instances.

You can also use Chordiant Interaction Server to map XSDs to web-based forms, and integrate information submitted through these forms with your applications using the GenericDialogServerServlet and the dialogServer Transform Type.

To integrate Chordiant Interaction Server with Chordiant 5 Foundation Server applications (general summary):

1. Create the XSD and Java class files using Rational Rose. Note that the Business Component Generator will automatically create the XSD and Java class files from your object model. Refer to the *Chordiant 5 Foundation Server Application Components Developer's Guide* for details.
2. Create the web-based form using Chordiant Interaction Designer, based on the XSD you created earlier.
3. Within the Context Map, specify the web-based form you generated using Chordiant Interaction Designer as the Presentation and dialogServer as the Transform Type for the appropriate context.

   Code Sample 12-42 shows the context within the Context Map for the application that you can use to engage the Chordiant Interaction Server to generate the appropriate HTML output.

```
<CONTEXT>
    <NAME>DEFAULT</NAME>
    <APP_LOGIC>/ChordiantServletBaseClassServlet</APP_LOGIC>
    <PRESENTATION>/presentations/html/customer_info.htm</PRESENTATION>
    <TRANSFORM_TYPE>dialogServer</TRANSFORM_TYPE>
</CONTEXT>
```

Code 12-42: Specifying Chordiant Interaction Server in the Context Map

4. Subclass the GenericDialogServerServlet to process the Chordiant Interaction Server form input, and override the execute method in your derived class.

   The Chordiant Interaction Server passes the document object to the execute method, enabling you to get access to the information submitted through the web-based form.

   You can use the execute method to perform special processing that could include the following:

   — Saving user inputs in the session for use by other servlets

   — Updating or retrieving database information

   — Saving data in the results field of the request object for use by the presentation context

   In the execute method signature, obj is the user input obtained using the getInputs method:

```
public void execute(java.lang.Object obj,
    com.chordiant.application.HttpServletRequest request,
    com.chordiant.application.HttpServletResponse response)
```

Code 12-43: execute Method Signature

   This getInputs method extracts inputs from a Chordiant Interaction Server display and returns them as an org.w3c.dom.Document object. You can override the getInputs method by an extending class to handle inputs from non-Chordiant Interaction Server display.

```
public java.lang.Object getInputs(
    com.chordiant.application.HttpServletRequest request)
```

Code 12-44: getInputs Method Signature

Code Sample 12-45 illustrates how to use the default behavior of the GenericDialogServerServlet to process the form input, in this case returning the form input as XML:

```
<CONTEXT>
    <NAME>DEFAULT</NAME>
    <APP_LOGIC>/GenericDialogServerServlet</APP_LOGIC>
    <PRESENTATION/>
</CONTEXT>
```

Code 12-45: Using the Default Behavior of the GenericDialogServerServlet

The getBusinessObject method is a powerful way to obtain Java objects, for example, business objects, directly from the user inputs (that is, the XML instance created from the CIS form by the CIS engine). The parameters are:

- the XML user inputs in XML instance document form
- the XPath that identifies the node that contains the desired business object instance description

You can then optionally perform additional operations, such as updating information, using this data from the business object.

5. Optionally, you can initialize a Chordiant Interaction Server form with a business object by setting it as the result of the application logic, and using the dialogServer Transform Type.

   When the transformation type for the presentation context is dialogServer, Chordiant 5 Foundation Server uses the request result object of the application logic resource as initialization data in the presentation. This is performed in the ChordiantServletBaseClass by first transforming the object to its XML representation and then providing the representation to the Chordiant Interaction Server engine as initialization data.

   The Chordiant Interaction Server engine uses the XSDs with which the HTML form was designed to map the nodes of the XML representation to the fields in the form.

Note: For examples of how to integrate Chordiant 5 Foundation Server with Chordiant Interaction Server, see the source code for the ColorShape and Harmony Bank sample implementations.

Chapter 13

Network Presence

Thin-client applications can receive asynchronous events from services or other applications in the same manner that applications send requests to services. This is achieved by establishing a Network Presence and registering for events. The complete infrastructure for enabling a thin client for Network Presence is provided by a variety of Java classes, XSL, and some JavaScript.

Contents of the Network Presence in the Browser

The Network Presence component was designed to be very lightweight, to keep content size and load time to a minimum and to keep application and business functionality on the server. The Network Presence component minimally requires just three pieces of code: HTML to contain the applet, the applet itself, and Java Script.

The sample `AppletFrameSource.xsl`, Code Sample 13-1 on page 333, shows the parameters that are required in an HTML frame to contain the presence. The `AppletFrameSource.xsl` file is not provided in the Chordiant base code. You must provide the HTML, JSP, or JavaScript, including the specified parameters to interface with your specific application.

The `AppletFrameSource.xsl` references, and causes to be loaded, the other two components which are included in the Chordiant base code:

- the Java Scripts file (`NWP_API.js`, size =3KB)
- the applet JAR file (`NWPThinClient.jar`, size =37KB).

Refer to the "JavaScript-Function Event Handlers" on page 334 for a description of how to add handlers for the incoming, asynchronous events. When designing handlers, try to stay consistent, keeping the browser contents to a minimum and keeping the application and business logic on the servers, typically by dispatching HTTP requests, rather than writing a lot of Java Script.

ESTABLISHING A NETWORK PRESENCE

The Network Presence for the thin client is embodied in the applet class com.chordiant.application.thinclient.NetworkPresenceBaseClass. Applets have a life cycle that is driven by the browser. When the page containing an applet is first displayed, the init method is invoked. The applet uses this event to generate a network presence key and register with the name service of the J2EE Application Server (JNDI). When the applet is destroyed, the Network Presence is automatically deregistered.

Note: The Java Console loaded on client machines must match, or be close to, the version of the JRE used by the application server. If the versions are not synchronized, the client application will display a "security error" message. The Java Console on the application server will show security/permissions exception error messages stating that the browser was unable to create the network presence socket server.

Register and Deregister Requests

The SendRegisterNetworkPresenceCommand method is automatically called from the applet init method and always creates a new socket server. (You can specify the listen port for the socket server or it can be automatically determined by the Operating System to some available port.) If successful, it attempts to establish a Network Presence by sending an HTTP Request to the Request Handler with the Action ID value of RegisterNetworkPresence.

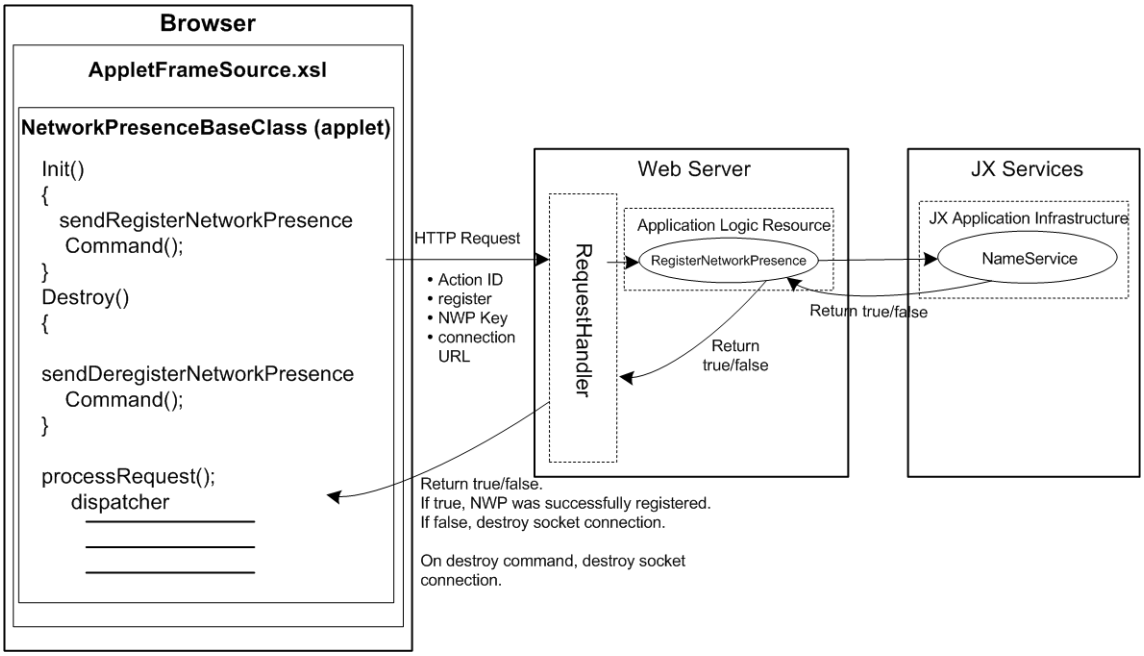


Figure 13-1: Establishing Network Presence from a Browser

The context mapped to this ID references the Application Logic Resource servlet class com.chordiant.application.RegisterNetworkPresence.

```
<ACTION_ID>
    RegisterNetworkPresence
    <CONTEXT>
        <NAME>DEFAULT</NAME>
        <APP_LOGIC>/RegisterNetworkPresenceServlet</APP_LOGIC>
        <!-- No Presentation is required as servlet will create
  response contents and not call super.service(). -->
    </CONTEXT>
</ACTION_ID>
```

The request provides the following HTTP request parameters:

- ActionId=RegisterNetworkPresence
- Register=register
- networkPresenceKey — A network presence key

  The networkPresenceKey is a string used to identify a particular network presence. It is created in the applet init method.

  The format of the networkPresenceKey is UN_BIPA_BPID, where:

  — UN:The username

  — BIPA: The browser's IP address

  — BPID: The browser's process ID

- connectionURL — The URL connection parameter.

  The connectionURL is a string that describes how to connect back to the Network Presence.

  The format of the string is socket://BIPA:BSN, where:

  — BIPA: The browser's IP address, obtained from an applet initialization parameter

  — BSN: The browser's socket server port number, obtained from the SocketServer

The HTTP request is directed at the Request Handler for the web application context which vended the current page. If the register command fails, either with an error or as indicated by the String value "false" being returned in the HTTP response, the socket server is shut down. The return value is the contents of the HTTP response, typically "true" if successful and "false" otherwise.

The HTTP Request provides information sufficient for the servlet to successfully register the applet as a Network Presence. Some of the required information is obtained from applet initialization parameters.

The SendDeregisterNetworkPresenceCommand method is automatically called from the applet destroy method. The method sends an HTTP request and provides two request parameters: ActionId=RegisterNetworkPresence and Register=deregister. The method always shuts down the socket server, regardless of the contents returned in the request response.

The Applet HTML Frame

The applet defines the names of applet initialization parameters that are required for Network Presence. Their values are provided by the servlet that vends the page containing the applet frame. The Network Presence Tester application's com.chordiant.application.nwptester.servlets.BuildAppletFrameResults servlet and `AppletFrameSource.xsl` resources work together to create an HTML page that provides the required information. The file `AppletFrameSource.xsl` is a stylesheet that transforms the results of the servlet into an HTML page with the necessary structure and information.

Required Applet Initialization Parameters

USER_NAME_APPLET_INIT_PARAM – "UserName" (industry standard)

USER_AUTH_TOKEN_APPLET_INIT_PARAM – "AuthenticationToken" (industry standard)

CLIENT_IP_ADDR – "ClientIPAddr" (Chordiant-specific)

These parameters are also described in the Javadoc for the NetworkPresenceBaseClass, located in the com.chordiant.application.thinclient package.

Optional Applet Initialization Parameters

LOGGING_ON_PARAM_NAME – "LoggingOn" – 'true' or 'false'

SOCKET_PORT_PARAM_NAME – "SocketPort"

Specifies the socket port number to use for listening for callbacks. Zero, or the absence of the parameter, means the operating system should select an available port.

If the requested port is unavailable, or some other error prevents the SocketServer from successfully initializing, a JavaScript alert is displayed, an error message is displayed in the status bar (if the status bar is available in the browser), and error messages with additional information are written to the Java console.

Debugging information is written to the Java console.

Code Sample 13-1 shows a sample `AppletFrameSource.xsl` with the required parameters.

Note: This is a sample file. Chordiant does not provide this file in the code base. You must create your own HTML to encapsulate this information, including the required parameters described in "Required Applet Initialization Parameters" on page 332, for your own application.

```
<html>
    <head>
        <title>Network Presence Applet Frame</title>
        <script type="text/javascript" language="javascript" src="../NWP_API.js">
            <xsl:comment>this comment required to workaround bug in IE </xsl:comment>
        </script>
    <body>
        <object classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
            id="NetworkPresenceApplet"
            name="NetworkPresenceApplet"
            width="0" height="0"
            codebase="http://java.sun.com/products/plugin/1.3/jinstall-13-win32.cab#Version=1,3,0,0">
            <param name="code" value="com.chordiant.application.thinclient.NetworkPresenceBaseClass.class"/>
            <param name="archive" value="/A_NWP_Tester//NWPThinClient.jar"/>
            <param name="type" value="application/x-java-applet;version=1.3"/>
            <param name="scriptable" value="true"/>
            <param name="MAYSCRIPT" value="true"/>
            <param name="LoggingOn" value="true"/>

<!-- 'SocketPort' param value specifies the desired port number.
        Zero (or absence of param) means let the OS choose. -->
<!--    <param name="SocketPort" value="0"/> -->

            <param name="UserName">
                <xsl:attribute name="value">
                    <xsl:value-of select="UserName"/>
                </xsl:attribute>
            </param>

            <param name="AuthToken">
                <xsl:attribute name="value">
                    <xsl:value-of select="AuthToken"/>
                </xsl:attribute>
            </param>

            <param name="ClientIPAddr">
                <xsl:attribute name="value">
                    <xsl:value-of select="ClientIPAddr"/>
                </xsl:attribute>
            </param>
```

Code 13-1: AppletFrameSource.xsl

JavaScript-Function Event Handlers

The Network Presence applet implements the SocketServerRequestHandler interface, which requires it to provide a single method, processRequest[1]. This is the method the socket server calls when data is received through the socket connection. The Network Presence applet's implementation is the entry point for all asynchronous calls. The incoming data is in payload form and contains the event class (service name) and event user data[2].

The processRequest method calls the `networkPresenceEventHandler` method (`NWP_API.js`) which dispatches the event and event data to the registered JavaScript handler function.

Applications register JavaScript functions with the Network Presence client to provide functionality for responding to the events. Registration is achieved by calling the JavaScript function registerEventHandler (in `NWP_API.js`).

```
<body onload="top.appletFrame.registerEventHandler('PeerMessageClientAgent',
IncomingChatMessageEventHandler);>
```

The processRequest method parses the incoming event data and calls the JavaScript function with four parameters.

- eventClass – event or service name (also used to map to the handler function)
- eventData – function name
- eventDataFormat – DEPRECATED and no longer used.
- eventUserData – contains the complete contents of the event in payload data form.

```
function IncomingChatMessageEventHandler(eventClass, eventData, eventDataFormat,
    eventUserData)
{
    // perform application-specific functionality
    alert("inside the IncomingChatMessageEventHandler, with eventClass = " +
          eventClass +
       ", eventUserData = " + eventUserData + ", eventDataFormat = " +
          eventDataFormat +
       ", and eventData = " + eventData);

    // parse the payload data and get the message text to be displayed
    var theData = parseEventData(eventData);
```

1. Note that the data formatting rules for the "payload" that is sent to and returned from the network presence processRequest method is exactly the same as the PayloadData rules for the Chordiant EJB (SOAP-encoded XML). Detailed information on PayloadData can be found in "Passing Payload with PayloadData" on page 140.
2. If the value of the eventUserData is "PING", the applet immediately returns an acknowledgement using payload data predefined by the com.chordiant.service.constants.ServiceConstants. No JavaScript handler functions are called.

Code Sample 13-2 shows these four parameters:

```
Get the serviceName and functionName parameters of the PayloadData.
If this is a ping request (that is, function name is 'Ping'), return OK message and do no
event dispatching.
Otherwise the request must get routed (one event thread at a time) to the specified
callback handler JavaScript method.
       args[0] = eventClass;
       args[1] = data;
       args[2] = payloadFormat;
       args[3] = applicationUserData;
       callResult = appletWindowJSObject.call(
          "networkPresenceEventHandler", args);

Allow next event thread to continue.
Send return value from JavaScript method back to sender of NWP event.
```

Code 13-2: processRequest Method Pseudocode

Event handler functions must return a SOAP-encoded XML String containing a PayloadData object. An empty return value will cause a socket read error. Refer to the com.chordiant.service.constants.ServiceConstants class for two methods that build an appropriately formatted response String value.

Optionally, an application can designate a single handler rather than registering individual handler functions by providing the function networkPresenceEventHandler(eventClass, eventData, eventDataFormat, eventUserData) and not include the `NWP_API.js` file. In this case, the application would not use the register and deregister JavaScript methods. Instead, it would completely take over callback handling in any way it sees fit.

Serialized Events

While NWP events coming into the browser can occur in parallel, dispatching events to the JavaScript handler functions is serialized because thin clients are not able to coordinate multiple simultaneous callbacks safely. Therefore, your JavaScript functions for handling NWP events should be very fast in their processing and should never perform blocking operations, such as modal dialogs.

Payload Data

The structure of event data is SOAP-encoded XML. The root element contains a single "payload" object. Java code can create the required form by instantiating a com.chordiant.service.PayloadData object and then using the TransformHelper to serialize the object into XML. Code Sample 13-3 illustrates a sample XML document:

```
<?xml version='1.0' encoding='UTF-8'?>
<root
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'
    xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'>
    <payload id='id0'
        xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
        xsi:type='ns1:PayloadData'>
        <fieldData id='id1'
            xmlns:ns1='http://www.themindelectric.com/collections/'
              xsi:type='ns1:vector'>
            <item id='id2'
                xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
                xsi:type='ns1:ParameterPair'>
                <fieldName xsi:type='xsd:string'>name of data item goes here</fieldName>
                <fieldData xsi:type='xsd:string'>value of data item goes here</fieldData>
            </item>
            <item id='id3'
                xmlns:ns1='http://www.themindelectric.com/package/com.chordiant.service/'
                xsi:type='ns1:ParameterPair'>
                <fieldName xsi:type='xsd:string'>additional data item name</fieldName>
                <fieldData xsi:type='xsd:string'>additional data item value</fieldData>
            </item>
        </fieldData>
    </payload>
</root>
```

Code 13-3: Sample Payload XML Document

MSXML Parser

One way that thin client applications using Network Presence can leverage the XML format of the callback event/response data within the browser is to use Microsoft's XML parser - MSXML component.

Note: Versions of Internet Explorer prior to 6.0 might not have the MSXML.dll installed. You can download it from Microsoft's Download Center: http://www.microsoft.com/downloads

This is an optional component that is not required by Chordiant. It is a suggestion for a tool you can choose to use.

Code Sample 13-4 demonstrates an example usage of the MS XML parser.

```
function parseEventData(theEventData)
{
    var results = new Object();
    var theData = new String(theEventData);
    var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
    xmlDoc.async="false";
    xmlDoc.loadXML(theData);
    var objNodeList = xmlDoc.getElementsByTagName("item");
    alert("objNodeList.length: " + objNodeList.length);
    for (var i=0; i<objNodeList.length; i++)
    {
        var fieldName =
            objNodeList.item(i).getElementsByTagName("fieldName").item(0).text;
        var fieldData =
            objNodeList.item(i).getElementsByTagName("fieldData").item(0).text;

        results[fieldName] = fieldData;
    }

    return results;
}
```

Code 13-4: JavaScript Sample Using the MS XML Parser

SECURITY AND NETWORK PRESENCE

In the Chordiant system, callbacks involving Network Presence are secure for two reasons: the location of the browsers and the location of the logic processing.

- **Location of "Smart" Browsers:** "Smart" browsers are those that include Network Presence. Network Presence is not required on all clients. For example, an account holder checking her balance from home does not require Network Presence on their personal browser. Only clients who will be receiving callbacks, for workflows and other push functionality, require Network Presence. These clients are usually located within a branch or call center. As such, these browsers are usually located physically within a protected network, behind a firewall, or are connected to the trusted network logically via Virtual Private Network (VPN). Within the protected network, all calls back to the browser are secure.
- **Location of Logic Processing:** When a call from the server reaches the "smart" client, the call is usually passed back to the server side, where its logic is processed. All service calls on the server side must go through the authorization process, since all service APIs require an authentication token for input. Only minor processing occurs on the browser itself.

Browser Security

The Network Presence component requires the Java plug-in for the browser to run a socket server in the browser. This Java plug-in is installed into the browser with the JDK/JRE installation that you have already performed.

The Network Presence component returns the location of the client to the application server, so your application can make a call back to the client.

In addition to the client's IP address, you may also want to use the client machine's host name in an HTTP session. For example, the Chordiant Teller application uses the host name to determine which devices are attached to a client's machine.

There are two ways to grant the security privileges:

- "Choosing a Signed Network Presence Plug-In" on page 339
- "Modifying the java.policy File" on page 340

Performing either one of these modifications will grant the privileges. Both modifications are not required.

Choosing a Signed Network Presence Plug-In

Chordiant provides two Network Presence plug-ins in CAFE:

- **A *Signed* Network Presence applet plug-in** — Defines the Network Presence applet to use the client machine's host name, which requires additional security in the form of a digitally-signed applet. In this plug-in, the `networkpresenceapplet.jsp` file's useHostName parameter is true. The certificate is located in the `NWPThinClient.jar`.
- **An *Unsigned* Network Presence applet plug-in** — Defines the Network Presence applet without using the host name. No additional security is required. In this plug-in, the `networkpresenceapplet.jsp` file's useHostName parameter is false.

For most applications, the unsigned applet plug-in is installed by default. For applications that require the client's host name, like Chordiant Teller, the signed applet is installed by default.

To override the default settings, you can select which plug-in to use in your desktop configuration, following the standard CAFE customization guidelines.

To specify the signed Network Presence applet:

1. Locate the CAFE desktop XML configuration file in the `/iAdvisorWeb/Preferences/config` directory.
2. Update the values as shown in Code Sample 13-5.

```
<plugins>
   ...
   <plugin name="nwpplugin">
      <source>/Advisor/iAdvisorWeb/plugins/networkpresence-signed/nwpplugin.js</source>
      <unLoad>unLoadNWPPlugin()</unLoad>
      <persist>none</persist>
   </plugin>
   ...
</plugins>
```

Code 13-5: Specifying the Signed Network Presence CAFE Plug-in

3. Save the configuration file to the `xAdvisorWeb/Preferences/config` directory.

To specify the *unsigned* applet, change the value of the <source> tag to /Advisor/iAdvisorWeb/plugins/networkpresence/nwpplugin.js. The name of the file is the same, but it is located in a different directory.

Notes: Do not mix both the signed and unsigned Network Presence applets within your application desktops. If you are using a desktop that requires the signed applet, such as Chordiant Teller, update your other desktops for that application to require the signed applet.

Setting the <persist> tag to none requires the applet to be reloaded and, if it is signed, recertified.

Refer to the *Chordiant 5 CAFE Client Developer's Reference Guide* for more information on these tags and on specifying the desktop configuration.

The signed plug-in contains a certificate with Chordiant's signature. If you use this certificate, users will see the security warning dialog box illustrated in Figure 13-2 when the applet is loaded.

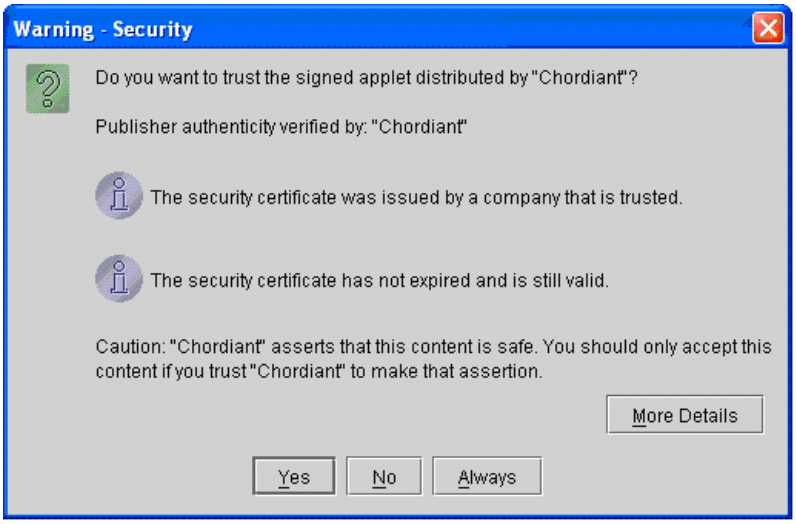


Figure 13-2: Security Warning Dialog Box

You may replace Chordiant's signing certificate with your own company's signing certificate, if required by your company's security policy. This is a standard Java procedure. The Java JDK includes utilities to create a self-signing certificate or request a Certification Authority-signed certificate, and to use a certificate to re-sign an applet. You can use these tools in conjunction with a Certification Authority company.

For information on working with certificates, refer to
http://java.sun.com/j2se/1.4.2/docs/guide/plugin/developer_guide/contents.html

Modifying the java.policy File

The security configuration that must be set for the Java plug-in will allow inbound socket connections to the Network Presence applet. This configuration is made in a the `java.policy` file, a text file which lives in the Java plug-in installation directory. You must manually edit the `java.policy` file to modify a particular line in it that specifies socket permissions.

The location for this file is:

{JRE PATH}`\lib\security\java.policy`

To find the path of the JRE you are using:

1. Open the Windows **Control Panel** from the **Start** menu.
2. Double-click the **Java Plug-in**.

3. Select the **Advanced** tab to see the JRE location.

To change the security permissions through the `java.policy` file:

1. Locate the following line of code, which is commented out by default:

```
//permission java.net.SocketPermission "localhost:1024-", "listen";
```

2. Uncomment the line and change the last argument to open up the socket security.

```
permission java.net.SocketPermission "localhost:1024-", "accept, connect, listen, resolve";
```

3. If the host name parameter is set to true, it requires the socket permission to be set to:

```
permission java.net.SocketPermission "*:1024-","accept, connect, listen, resolve";
```

Note: The configuration changes in Step 2 and Step 3 must be made on every browser client that needs to run thin client applications with Network Presence capabilities.

4. Determine which Java plug-in is installed (or not installed) for the Microsoft Internet Explorer browser.

   From the **Tools** menu, select **Internet Options**. On the **Advanced** tab, look under **Java (Sun)**. Your settings should look similar to this:

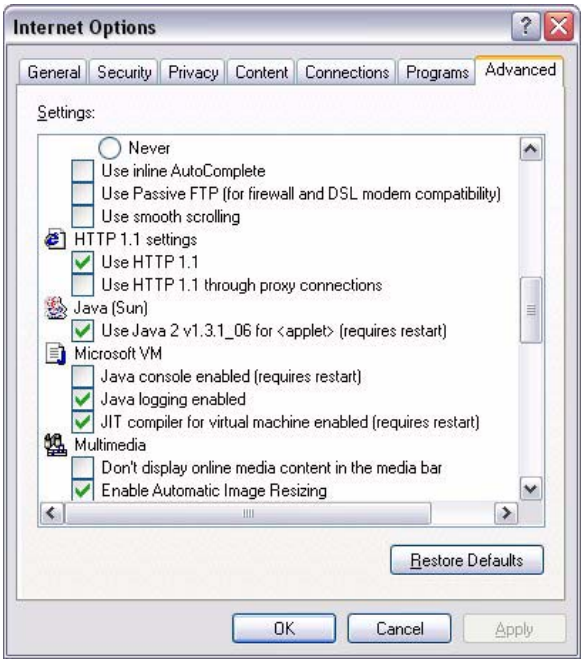


Figure 13-3: Verifying Java Plug-in

Note that the Microsoft VM should NOT be selected as Chordiant does not specifically certify on it, although it might work.

Additional Scenarios Requiring Security Privileges

You must also use the signed applet or update the `java.policy` file if you want to use either the host name or IP address in the following circumstances:

- If you are running the HTTP server and the J2EE application server on different machines.
- If there is a Network Address Translation (NAT) device, like a proxy server, between the browser and the web server.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X