Novell exteNd Composer™ Enterprise Server

5.0

www.novell.com

USER'S GUIDE

Novell®

Legal Notices

Novell, Inc.

1800 South Novell Place

Provo, UT 85606

www.novell.com

exteNd Composer Enterprise Server ***User's Guide***

January 2004

Online Documentation: To access the online documentation for this and other Novell products, and to get updates, see www.novell.com/documentation.

Novell Trademarks

eDirectory is a trademark of Novell, Inc.
exteNd is a trademark of Novell, Inc.
exteNd Composer is a trademark of Novell, Inc.
exteNd Director is a trademark of Novell, Inc.
jBroker is a trademark of Novell, Inc.
NetWare is a registered trademark of Novell, Inc.
Novell is a registered trademark of Novell, Inc.

SilverStream Trademarks

SilverStream is a registered trademark of SilverStream Software, LLC.

Third-Party Trademarks

All third-party trademarks are the property of their respective owners.

Third-Party Software Legal Notices

Jakarta-Regexp Copyright ©1999 The Apache Software Foundation. All rights reserved. Xalan Copyright ©1999 The Apache Software Foundation. All rights reserved. Xerces Copyright ©1999-2000 The Apache Software Foundation. All rights reserved. Jakarta-Regexp , Xalan and Xerces software is licensed by The Apache Software Foundation and redistribution and use of Jakarta-Regexp, Xalan and Xerces in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notices, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "The Jakarta Project", "Jakarta-Regexp", "Xerces", "Xalan" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org. 5. Products derived from this software may not be called "Apache" nor may "Apache" appear in their name, without prior written permission of The Apache Software Foundation. THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright ©1996-2000 Autonomy, Inc.

Copyright ©2000 Brett McLaughlin & Jason Hunter. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution. 3. The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jdom.org. 4. Products derived from this software may

not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management (pm@jdom.org). THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This Software is derived in part from the SSLava™ Toolkit, which is Copyright ©1996-1998 by Phaos Technology Corporation. All Rights Reserved. Customer is prohibited from accessing the functionality of the Phaos software.

The code of this project is released under a BSD-like license [license.txt]: Copyright 2000-2002 (C) Intalio Inc. All Rights Reserved. Redistribution and use of this software and associated documentation ("Software"), with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain copyright statements and notices. Redistributions must also contain a copy of this document. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The name "ExoLab" must not be used to endorse or promote products derived from this Software without prior written permission of Intalio Inc. For written permission, please contact info@exolab.org. 4. Products derived from this Software may not be called "Castor" nor may "Castor" appear in their names without prior written permission of Intalio Inc. Exolab, Castor, and Intalio are trademarks of Intalio Inc. 5. Due credit should be given to the ExoLab Project (http://www.exolab.org/). THIS SOFTWARE IS PROVIDED BY INTALIO AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE DISCLAIMED. IN NO EVENT SHALL INTALIO OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Contents

About This Guide

Purpose

This guide describes how to use exteNd Composer Enterprise Server and its related administrative facilities, APIs, and classes to deploy and manage Composer applications. As such, it is an adjunct to the *exteNd Composer User's Guide*.

Audience

This guide is aimed at the application server administrator and/or persons tasked with deployment and amanagement of Composer services.

Prerequisites

This book assumes prior familiarity with the exteNd Composer design-time environment and Composer application-building metaphors. You should also be familiar with Java archive formats (WAR, EAR, JAR) and J2EE deployment concepts in general.

Organization

This guide is organized as follows:

Chapter	Description
Chapter 1, *Welcome to exteNd Composer Enterprise Server*	Gives a definition and overview of the exteNd suite of products.
Chapter 2, *Server Overview*	Briefly describes exteNd Composer Enterprise Server specifications and the production runtime environment.
Chapter 3, *Planning Your Deployment*	Outlines the key environmental and resource-related factors that should be considered before deploying a Composer service.

Chapter	Description
Chapter 4, *Deploying a Project*	Explains the available Service Trigger options and how to use the exteNd Composer Deployment Wizard.
Chapter 5, *Using the Deployment Framework*	Describes how to customize or extend the application server framework classes for non-standard deployments. Read this chapter if you need to use custom service triggers.
Chapter 6, *Transaction Management*	Describes options for controlling the transactional aspects of your application.
Appendix A, *Novell exteNd Application Server Dependencies*	Describes database connection-pool issues specific to deployment in the Novell exteNd Application Server.
Appendix B, *Contents of Deployments Objects*	Describes the content of the files that are installed into the application server.
Appendix C, *Deployment Framework API Documentation*	Describes the exteNd Composer Enterprise Server Java framework files.
Appendix D, *Reserved Words*	A listing of keywords that are used by Composer and should be avoided in your code.
Appendix E, *Glossary*	Definitions of terms used in this guide.

Conventions

This guide uses the following stylistic and typographical conventions.

Bold serif typeface within instructions indicate action items, including:

- Menu selections
- Form selections
- Dialog box items

Bold sans-serif typeface indicates:

- Uniform Resource Identifiers
- File names

Italic typeface indicates:

- Variable information that you supply
- Technical terms used for the first time
- Title of other Novell publications

`Monospaced` typeface indicates:

- Method names
- Code examples
- System input
- Operating system objects

Additional documentation

For the complete set of Novell exteNd Director documentation, see the Novell Documentation Web Site:

http://www.novell.com/documentation/exteNd.html

1 Welcome to Novell exteNd Composer Enterprise Server

Novell exteNd is a suite of web application development products aimed at reducing the time required to develop and deploy powerful XML-enabled, portal-aware web applications for use on J2EE app servers. The Composer suite consists of three products:

- **Composer—**a visual design environment for creating B2B integration applications
- **Composer Enterprise Server—**a runtime environment that executes the applications created in exteNd Composer
- **Composer Enterpriser Connects—**a family of products that extend the capabilities of exteNd Composer and Server to permit the XML-enablement of diverse enterprise information sources such as databases, host applications, and Java components.

The focus of this Guide is Composer Enterprise Server., which is the app-server-resident "execution engine" for Composer-built services. (Each of the above pieces has its own documentation, so please refer to the *exteNd Composer User's Guide* for information on the design-time Composer executable, and refer to the various individual User's Guides for the specific exteNd Composer Connects that you need to incorporate into your applications.)

What is Composer Enterprise Server?

Novell exteNd Composer Enterprise Server (or Composer Server, for short) is the runtime environment for applications developed with exteNd Composer. It is a Java application that runs in its own thread on a J2EE-compliant enterprise application server. It starts up when the app server starts up and shuts down when the server shuts down.

Composer Enterprise Server provides both the *runtime execution engine* for Composer-built services (interpreting and processing the XML metadata deployed from Composer), and an *application-server tailored framework* that provides integration with services provided by an application server (e.g., thread management, connection pooling, load balancing, failover, security, transaction control).

Runtime capabilities provided by Composer Enterprise Server include:

- Deployment assistance
- XML parsing
- XSL and XForms processing
- Instantiation and execution of Connect objects via installable factories
- Interpretation of XML application object metadata
- Mediation of SOAP-related interactions with the app server
- Mediation of various other container-level app server interactions
- Connection pooling and caching

Support for Popular App Servers

Composer Enterprise Server is available for (and is tested against) various popular application servers, including not only the Novell exteNd Application Server but IBM's WebSphere, BEA WebLogic, and Apache Tomcat, running on various operating systems. (For the latest support matrix, go to **http://www.novell.com/documentation/exteNd.html**.)

The application framework consists partly of base classes that are environment-independent, and partly of classes tailored to the specific application server within which exteNd Composer Enterprise Server executes. Classes that are application-server-specific include classes responsible for:

- Logging
- Connection pooling
- Transaction control (Enterprise Edition only)

Service Types

Composer applications are organized into deployable units of work called *services*. The services consist of *actions* stored in *components*. (For a more precise definition of these terms, please consult the *Composer User's Guide*.) All of the action-model logic, connection info, and miscellaneous resources that make up the components inside a service, as well as the service wrapper itself, are packaged as XML metadata. (In other words, a Composer service is *not* compiled bytecode.) This means, among other things, that you can examine any individual component of a Composer service using an ordinary text editor.

Composer Enterprise Server is the runtime piece that invokes or instantiates services based on incoming requests; executes the instructions (actions) contained in the service and its components; manages caching and connection pooling; and provides for other runtime needs of executing services.

Composer Enterprise Server handles service-invocation requests from a number of sources:

- Servlet-based trigger objects (see below)
- EJB objects
- JSPs that invoke a service via custom tags that, in turn, reference the Composer tag library
- Direct programmatic invocation by Java objects

Service Triggers

A Composer service encapsulates the logic, connection information, and resources needed to execute a unit of work. The service does *not* encapsulate any triggering mechanism; invocation is abstracted out, to another type of object, known as a *service trigger.*

The service trigger is responsible for:

- Dealing with any transport-related issues
- Data acquisition (marshalling/unmarshalling)
- Instantiation of the target service
- Passing properly formatted data to the target service

In most cases, the trigger object is a conventional HTTP servlet. But there are other possibilities, to handle non-HTTP requests. Some of the other kinds of events that can trigger a Composer service include:

- Arrival of a message at a (JMS) message queue/topic. (This kind of event is monitored by a JMS *MessageListener*.) A service that responds to this kind of triggering is known in Composer as a *JMS Service*. This functionality is available only when the JMS Connect product is installed (as in the Enterprise Edition of the exteNd suite).
- The firing of an SAP function that has been designed to use a BAPI process to trigger a Composer servlet. This functionality is available only when the SAP Connect product is installed (as in the Enterprise Edition of the exteNd suite).
- A disk-I/O “write” operation in a given path location on a storage device. (A File Trigger causes a Composer service to start up when a new file appears in a given location.)
- Arrival of e-mail in a particular mailbox at a given mail server.
- Direct invocation by a “scheduled task” daemon. (Composer supports something called a Timer trigger.)

It is possible to assign more than one trigger type to a given service. The trigger merely acquires and forwards data to the service (after instantiating the service).

Service triggers will be discussed in additional detail in a later section.

2 Composer Enterprise Server Overview

This chapter introduces some basic runtime issues you will need to know about if you intend to deploy projects to Composer Enterprise Server and administer them at runtime. Those issues include:

- Deployment Archive Contents
- Push-Model versus Pull-Model Deployment
- Hot Deployment
- Removing (Undeploying) Existing Applications
- Updating Your License

Cache management and other administrative issues are discussed in the next chapter.

NOTE: This chapter assumes that you are familiar with EAR, JAR, and WAR packaging concepts, as well as other J2EE deployment idioms. You should also be familiar with runtime and deployment concepts applicable to the particular app server you will be targeting (Novell exteNd, IBM WebSphere, BEA WebLogic, Apache Tomcat).

Deployment Archive Contents

Composer follows a standard J2EE deployment model, using the EAR (**E**nterprise **AR**chive) file type as the deployment object.

The deployment EAR wrappers all of the project-level resources and components you've chosen to deploy. The EAR is scoped to a single *project* (.spf) file. The EAR encapsulates all services that exist in your project, and the resources they use.

You can create your deployment EAR either with Composer (design time) or Director. If you are using Composer Enterprise Edition (or the Enterprise Edition exteNd suite), you can use Composer's design-time deployment wizard to deploy projects straight to the app server (or to a staging area of your choosing). In this scenario, Composer does all the packaging for you, automatically, and puts the resulting EAR on the app server. The EAR is immediately "live," with no need to restart the server.

The other way of creating deployment archives is to use Director's native J2EE packaging facilities.

NOTE: Director-native deployment is covered in the Director documentation. Likewise, Composer Enterprise Edition deployment procedures are covered in the *Composer User's Guide*. See the appropriate guide for more information. The following discussion centers on low-level descriptions of deployment artifacts.

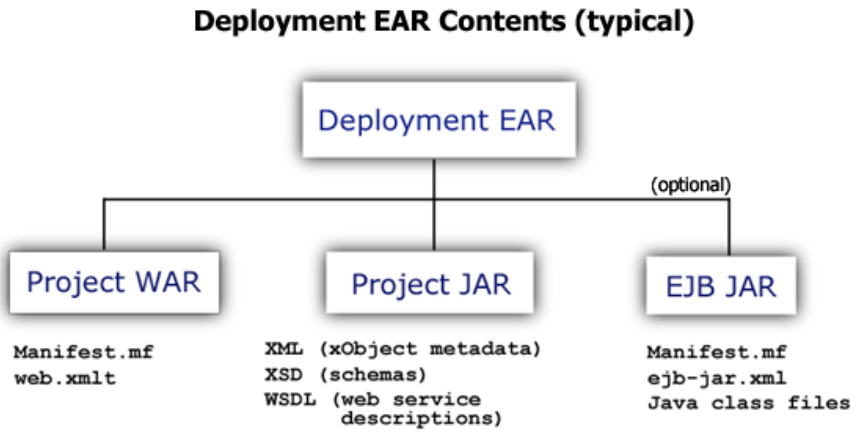
A Composer deployment EAR contains the following types of objects:

Table 2-1

Object	Use	Notes:
Project JAR File	Contains the services, components and resources of the project, in XML form.	The Components and other xObjects that comprise your services are stored in metadata form (not Java class files). Composer Enterprise Server uses these files to create your runtime objects on the app server. **NOTE**: This file is always generated, and is always packaged into the deployment EAR.
EJB Service Trigger class files	Allows services to be invoked through EJBs (potentially front-ended by JSPs), which in turn means Container services (for transaction control, etc.) are available..	EJB triggers are required for standalone use of Composer services as part of local business applications..

WAR file	Contains **manifest.mf** file (listing the JAR resources for this deployment) and **web.xml** (see below).	Required if Servlets or EJBs are created. Produced by Composer automatically.
web.xml file	Describes information necessary to install Service Trigger Java classes into the application server. The URI associated with the Servlet based Service Triggers and JNDI name for EJBs are described in these.	Created automatically and stored in a WAR file within the deployment EAR.
SilverCmd batch file called ImportObjects.bat	Contains SilverCmd utility calls to install the deployment objects into the Novell exteNd application server.	Created automatically. (And invoked automatically, if you choose the “Yes” radio button on the final screen of the Composer deployment wizard.) *This artifact is created only for the Novell exteNd Application Server.*
xc_deployment_info.xml	Contains the deployment profile from the last time the Deployment Wizard was executed.	Created automatically. Allows exteNd to restore the previous deployment information the next time a deployment is performed.

The following diagram summarizes the containment hierarchy of a deployment EAR.



(For a more detailed description of the contents of these objects, see the appendix.)

Novell exteNd App Server Database Requirement

Many J2EE application servers use ordinary disk storage as the "backing store" for app-server content. By contrast, the Novell exteNd Application Server (up through and including version 5.1) uses a database.

NOTE: For a list of supported databases and drivers, see the Novell exteNd Application Server release notes and documentation.

The app server stores its own internal classes and runtime artifacts in a default database called SilverMaster (or SilverMaster50; the name always contains the version number in the final two characters). You can deploy your own projects (your Composer and Director EAR and WAR files) directly to SilverMaster, if you wish. But for better encapsulation and easier management, you may want to create individual databases for each project. The deployment process, in this case, involves the following steps:

- Create a database and make it available to the app server (using the client-side **smc.exe** (Server Management Console) application that comes with Novell exteNd app server)
- Specify that database's name in the Server Profile corresponding to the deployment you wish to perform (in Director or Composer, use **Tools** > **Profiles . . .** to access the dialog where you can create or edit Server Profiles; and specify the target database there)
- Deploy to the app server

The deployment database for your project need only be created and installed once. You can then deploy and redeploy your project into that database as many times as needed.

NOTE: A visual UI for managing databases and drivers installed on the app server is available in the **smc.exe** (Server Management Console) app that ships with the app server. Consult the Novell exteNd Application Server documentation for details.

Push-Model versus Pull-Model Deployment

Deployment of Composer services is usually initiated from within a Composer or Director design-time environment. Director offers a variety of wizards and tools for creating and packaging deployment artifacts, including those needed to deploy Composer services. (See the Director documentation for details.) Composer Enterprise Edition has its own wizards and tools to enable direct deployment from the Composer design-time environment. (See the separate *Composer User's Guide* for details.) Composer can do a live deploy straight to a target app server, or a "packaging only" deploy to a staging area on disk.

Composer supports a push model as well as a pull model for deployment. The push model is the case described above, where you initiate deployment from the design side. In the pull model, deployment is initiated from a browser console on the server. (See the next chapter.)

Hot Deployment

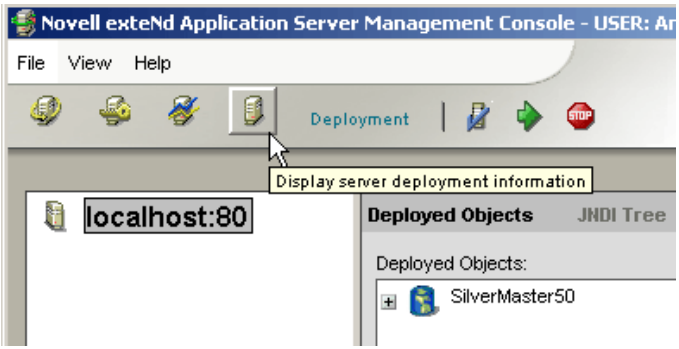
You can deploy a project to the app server while the app server is running, even if an earlier version of your project already exists on the server. (The old deployment EAR is simply overwritten.) There is no need to undeploy an existing EAR before deploying a new one. However, you should clear the cache (see "Clearing the Cache" in the next chapter) before running the newly deployed project, because it's always possible that old objects from the previous deployment are still in memory.

Removing (Undeploying) Existing Applications

Various app-server vendors offer various tools for managing deployed web applications. With the Novell exteNd Application Server, you can use the following procedure to undeploy an already deployed object (which is to say, remove it from its host database, without removing the database itself). To undeploy a deployed Composer project:

1 With the app server running, launch the Server Management Console (**smc.exe**) application.

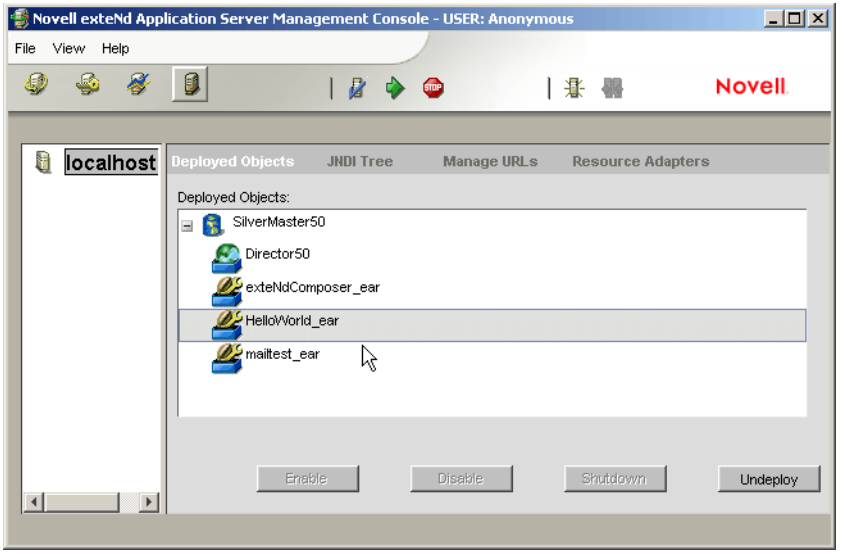
2 In the toolbar at the top of the main window, click the **Deployment** button (as shown below).



3 In the main window under **Deployed Objects**, locate the database that contains the deployed project you wish to undeploy. (All databases will be listed in tree view.) Toggle the plus sign next to the database "parent node" to expose its children. The child (or leaf) nodes of the tree represent deployed archives.

4 Single-click (select) the deployed archive you wish to remove. (See illustration below.)

CAUTION: *If your deployment database is SilverMaster, be careful not to select the Director EAR nor the exteNd Composer EAR. These EARs contain runtime executables for Director and Composer.*



5 Click the **Undeploy** button in the lower right corner of the window. The EAR or WAR in question will disappear from tree view and will no longer exist on the app server.

NOTE: You cannot undeploy individual Composer services one at a time. The entire project EAR will be undeployed as a unit.

For information on how to undeploy EAR and WAR files from other app servers, consult the appropriate vendor's product documentation.

Updating Your License

Should the need arise to update the license string associated with Composer Enterprise Server, you can use the **UpdateLicense.bat** file (located in your exteNdComposer\bin directory) to accomplish this. From the command line, run:

```
updateLicense product newLicense [Composer/Server]
```

where *product* is the name of the particular product (whose license you would like to update, *newLicense* is the license string, and the final argument (one of *Composer* or *Server*) specifies whether to update the design-time or runtime version of the product in question.

You can see a list of installed products by running:

```
updateLicense -L
```

3 Runtime Administration of Composer Enterprise Server

This chapter discusses subjects of importance to anyone who needs to administer deployed Composer services. Those subjects include:

- The various consoles available for managing deployed Composer services, and how to use them
- How to inspect and/or edit license-string info for Composer server-side products
- Cache management and performance-tuning issues
- Security roles
- How to publish (and control the visibility of) JAR files and custom Java classes

Runtime Administration Consoles

You can manage various aspects of Composer Enterprise Server's runtime operation through browser-based (JSP-powered) consoles. In addition to a General Properties console page where you can exercise control over settings of more-or-less global scope, there are individual consoles for the various Composer Connects (such as JDBC, LDAP, Telnet, and so on), which expose Connect-specific settings. The GUI allows easy navigation back and forth between and among the various console.

NOTE: The consoles depend, in part, for their functionality on JavaScript, so be sure scripting is enabled in your browser. Your browser should also be HTML 4.0 compliant and CSS-aware. No Java applets are used, however, so there is no need to have a Java-plugin-enabled browser.

In addition to offering a GUI for adjusting important runtime settings, the General Properties panel of the main administrative console lets you inspect and/or update your product license(s). This is discussed below.

Real-Time Update

Any console settings you wish to change or experiment with will be updated on the server in *real time*, as you adjust them, so that you do not have to restart the server. Changes to cache settings, pool settings, etc., take effect immediately.

How to Access the General Properties Console

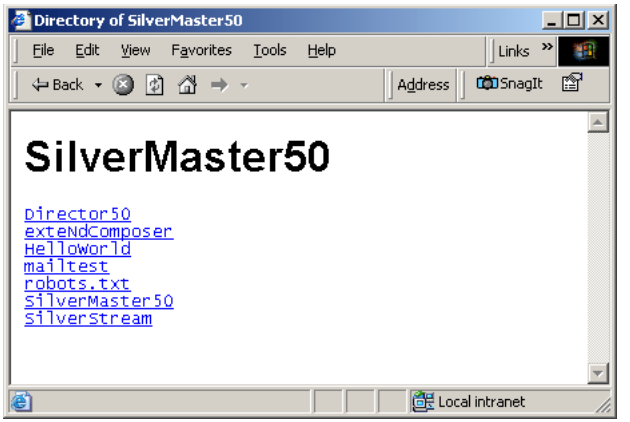
You can use the administrative console(s) at any time after the app server is running. The entry point is the General Properties page.

➢ **To access the General Properties page:**

1 Be sure the application server is running, with Composer Enterprise Server installed and operational.

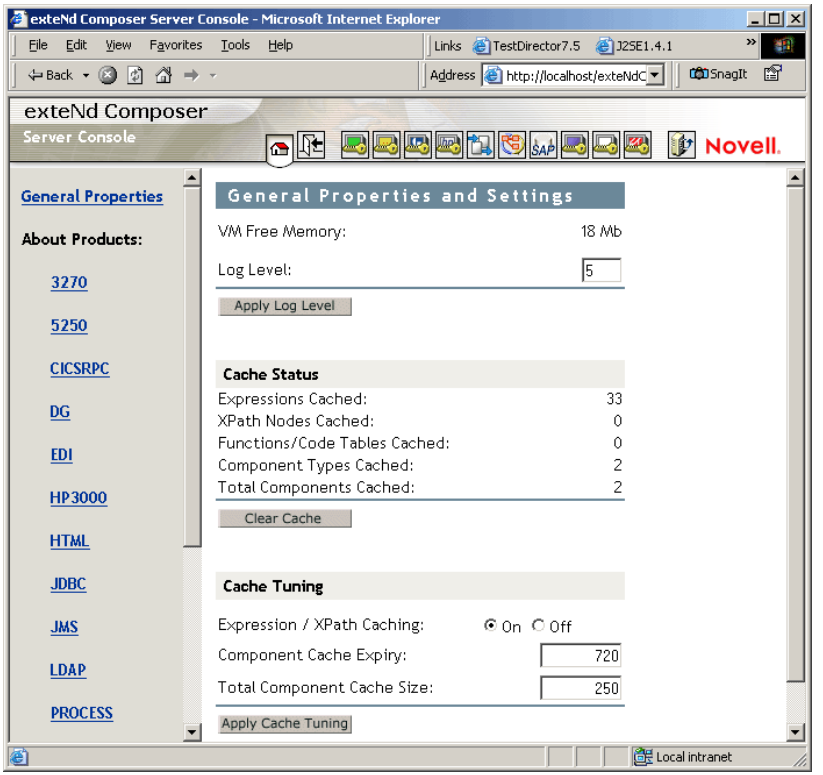
2 Launch your web browser.

3 **If the target server is Novell exteNd Application Server:** Navigate to the default host address and port (for example, **http:\\localhost:80**.) A master console window similar to the following will appear, with a list of links. Click the **exteNdComposer** link.



Other app servers: Enter the default host IP address, port, and "exteNdComposer/Console" in your browser window and hit Go. (The URL should look something like **http://localhost/exteNdComposer/Console**.)

The General Properties console screen will appear:



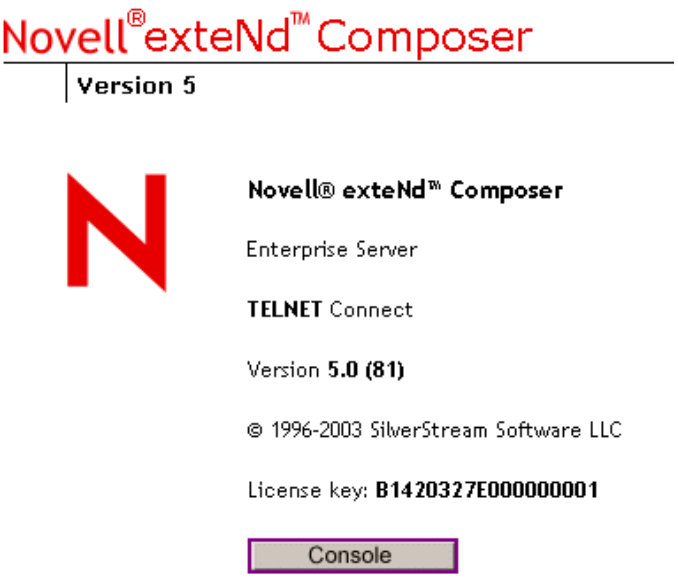
General Properties UI

The General Properties page (shown above) has a toolbar at the top, a navigator frame on the left, and a content frame with various text fields and buttons.

Navigator Frame

The navigator frame contains links for each of the Composer Enterprise Connect products that you have installed (including eval versions). Clicking any link will take you to a product-specific license-info page for the Connect in question. If the Connect in question is capable of using connection pooling, there will be a pushbutton on the license page labeled “Console.”

NOTE: An exception to this rule is the JDBC Connect, whose pooling is handled by the app server rather than by Composer Enterprise Server.



If you press the Console button, a new browser window will open, containing a console screen with information about connection pooling. (Consult the documentation for the individual Connects to learn more about the use of these connection-pooling consoles.) You can also open the connection-pooling console window(s) by use of the toolbar buttons, as described below.

Toolbar

At the top of the page, you'll find a row of buttons on a toolbar. The exact number and kind of buttons will depend on the number and type of Composer Enterprise Connect products you currently have installed on the server. The toolbar configuration for Composer Enterprise Edition is shown below:

Each button has a hover-tip associated with it. The tip appears above the button. In the illustration above, the cursor is hovering over the button corresponding to the 3270 Connect product. (The tooltip says "3270 Console.") Clicking the button will result in a new browser window opening, with the 3270 console showing in it.

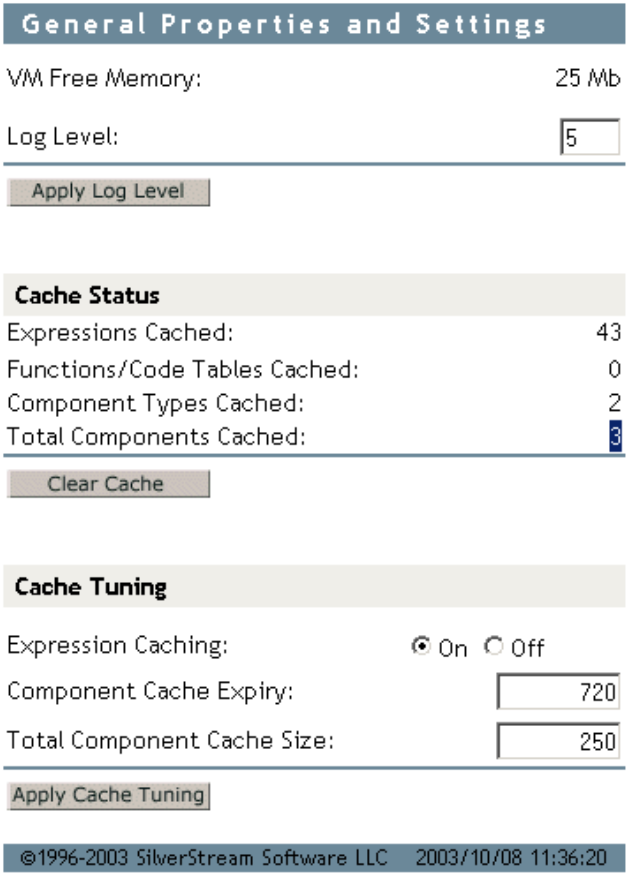
The very first button on the far left of the toolbar is a link to the General Properties page. This button is present on all Composer console pages.

The button next to the General Properties button is the Exit button. It closes the browser window.

The button at the far right of the toolbar is the Server-Based Deployment button. This button will take you to a series of deployment screens that you can use to locate and deploy a preexisting EAR, WAR, or JAR file that is ready to be retrieved from a staging area on a network drive. (In other words, this button will initiate a “pull-style” deployment.) To perform this kind of deployment requires that a deploy-ready archive (e.g., EAR) already exist somewhere on disk.

General Properties and Settings

The main frame of the General Properties page contains controls for inspecting and adjusting various runtime parameters on the fly.



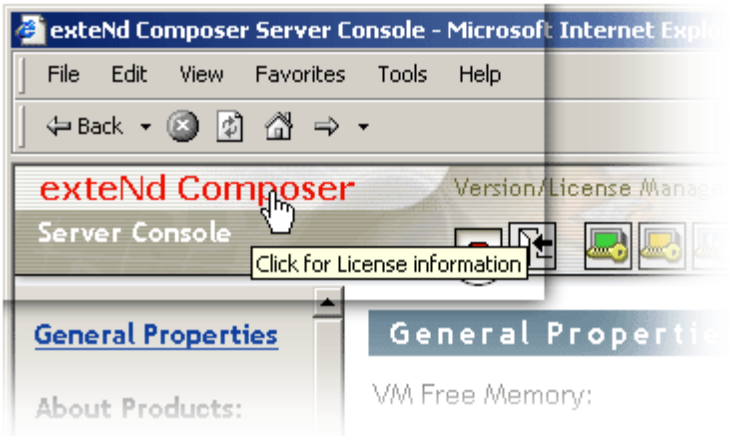
If you want to change the log-message threshold for your Composer project(s), enter a number from 1 to 10 in the **Log Level** field and click the **Apply Log Level** button. (The lower the number, the more verbose the logging.) Changes take place immediately.

Click the **Clear Cache** button if you want to purge all objects from the in-memory cache immediately. (See additional discussion below.)

You can enter new cache settings as desired (again, see discussion below), then click the **Apply Cache Settings** button to make your new settings take effect immediately.

License Manager

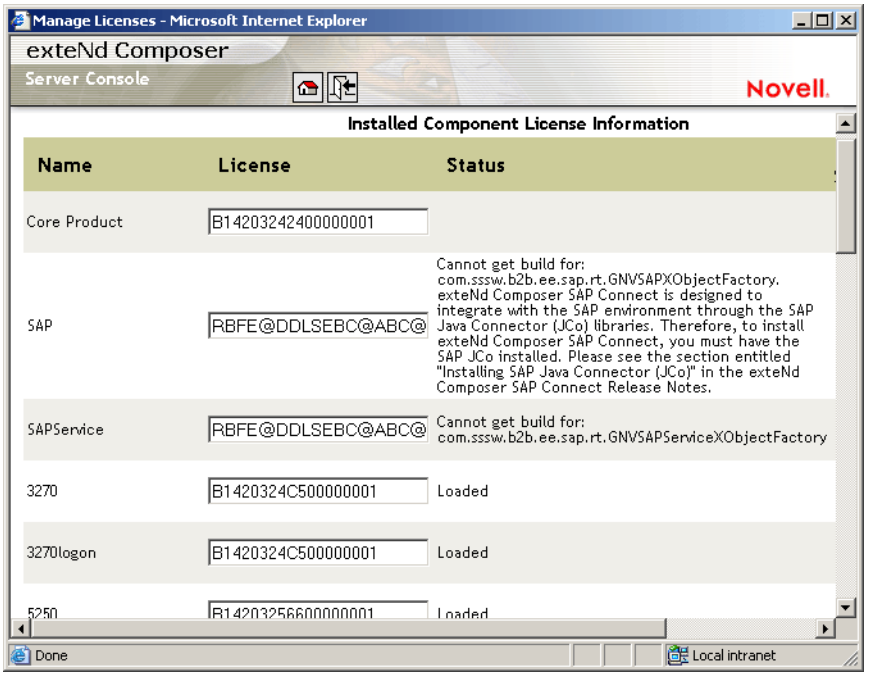
The exteNd Composer logo in the top left corner of the General Properties page is itself a button. The cursor changes to a hand when you allow the mouse to linger over the words “exteNd Composer.”



If you click the mouse when it is over the Composer logo, you will see the content area of the browser window change appearance:

This screen displays the current license key, product version and build number, and other important information. You may be asked for this information when and if you need to contact Customer Support.

At the bottom of the license summary page, there is a **Licenses . . .** button. If you click this button, a new browser window will open:



This page gives a detailed listing of license information, including Status info that may be useful for troubleshooting. In the above picture, for example, the entry for SAP Connect has a detailed status message explaining why the connector did not load. Likewise, the entry for SAP Service contains a message mentioning a specific class name. Again, you may be asked for this information when contacting Customer Support.

Caching and Cache Administration

The General Properties page of the Composer Enterprise Server console gives you the ability to inspect cache statistics as well as adjust caching parameters. This section, and the sections that follow, address the various issues you need to know about in order to use this portion of the console to best advantage.

What Is Caching?

Caching refers to temporary storage of in-memory objects that might be costly to create over and over again. It’s a technique for achieving runtime-object reuse.

The goal of caching is to enable higher performance: more units of work per second. When objects are already available in memory and don't have to be created from scratch, applications take less time to run. The trick is knowing which objects to cache, and how to *manage* the cache so as to minimize RAM usage, data-copying, garbage collection overhead, etc. These are nontrivial issues, especially in a container process that manages a heterogeneous,complex, fast-changing execution environment. Fortunately, Composer Enterprise Server does most of the hard work for you.

The down side to caching, in general, is the need for extra memory to store cached objects. Beyond this, there is the potential for performance *degradation* if cache-management overhead becomes great. The cost of managing a cache can become significant if the cache contains large numbers of objects, or if conditions are so dynamic that new objects are being "turned over" quickly.

Ideally, a cache should contain only frequently accessed items, and/or items that are costly to create. But it's not always obvious which items meet these criteria. The cache has to "know" how to identify (and retain) high-demand objects while removing infrequently accessed objects that are only taking up valuable memory.

Least-Recently-Used (LRU) Cache Algorithm

Composer Enterprise Server handles cache management automatically, via a *least-recently-used* (LRU) algorithm.

LRU means that cacheable objects, once they exist, are kept in memory until some predetermined number of cached objects has been reached or exceeded, at which point the *least recently used* objects will be removed if it is necessary to add new objects. The "predetermined number" is something you can set yourself, using the **Total Component Cache Size** control on the Composer Enterprise Server main console. Entering a large number in this field tells Composer Enterprise Server to maintain a large number of objects in memory, at the expense of available free Virtual Machine memory. Setting a *low* number means relatively few objects will be retained in memory, freeing up RAM. The default value is 250.

NOTE: A large value does not guarantee better performance: For example, routine JVM garbage collection (compaction and purging of memory) becomes more timeconsuming if the cache is large, and LRU analysis (and pruning) of the cache is more costly as well. You will have to experiment with different cache settings to find the "sweet spot" for your particular production environment.

Cacheable Objects

Composer can cache the following types of objects:

- Components (XML Map, JDBC, LDAP, Telnet, and other components)
- Actions (Log, Map, Function, Decision, etc.)
- User-scripted functions in Custom Script resources
- Code Table resources

Composer does not cache:

- Resource XObjects other than Code Table: For example, there is no caching of WSDL Resources, Form Resources, Images, JARs, XSD, etc.
- XML Templates
- User objects (custom Java objects)

Of course, CPUs, operating systems, and JVMs all have their own caching mechanisms. It's possible (indeed likely) that objects not cached by Composer will reside in a cache of one kind or another at runtime.

Cache Scope

Composer Enterprise Server provides runtime services for all Composer-built executables deployed on the app server, regardless of which EAR, WAR, or JAR file(s) the executables come from. Accordingly, caching operates across a scope that encompasses any and all Composer deployments on a given server. This means that any time you change cache parameters in the console, you are potentially affecting all deployed services.

For example, if you've deployed five projects, with three services each, and those 15 total services contain a grand total of 400 cacheable objects, Composer Enterprise Server will cache the 250 most recently used objects (no matter what type they are or which project they came from), assuming you've kept the default Total Component Cache Size setting of 250. If you adjust the cache size up or down, Composer Enterprise Server will add to or prune the cache as appropriate, again according to LRU only, with no regard for which object came from which deployed app.

User-Adjustable Settings

The user-adjustable caching parameters available on the General Properties console screen include:

- **Expression Caching on/off**—This radio button tells Composer Enterprise Server whether to include Actions (such as Map, Decision, Function, etc.) in the cache. (Actions are considered "expressions" at runtime.) If you are using a generous Total Component Cache Size (see below) but are not seeing any performance improvement under load, try turning Expression Caching *off*.
- **Component Cache Expiry**—This setting allows you to put a maximum limit (in minutes) on the lifetime of inactive (but still cached) objects. The default is 720 minutes (12 hours), which means no inactive item will stay in memory longer than 12 hours. (The key intuition here is that if an object has been in memory for 12 hours and hasn't been used, it probably doesn't need to be in memory any longer.)
- **Total Component Cache Size**—This is the maximum number of objects (of all types) that will be stored in the cache at runtime. The default is 250.

The cache-expiry and total size limits are enforced via a daemon process—a *cache pruner*—that runs in its own thread. Every ten seconds, the pruner inspects the cache to see if any objects have "expired" (reached their inactivity time limit, or "Expiry,"as discussed above), in which case those objects are summarily purged from the cache, regardless of whether the cache is full.

IMPORTANT: The console contains a button called Apply Cache Tuning. This button applies the changes you've made (if any) to cache settings and refreshes the console. *Don't forget to click this button after you've edited any cache settings.*

Clearing the Cache

The General Properties and Settings console contains a button called "Clear Cache." This button does just what it says: It immediately removes all stored objects from cache memory. The console's Cache Status numbers will update in real time to reflect this.

You will typically use the Clear Cache button when redeploying ("hot" deploying) a project after modifying it. If old, unmodified objects from the previous deployment are still in the cache, you may not see your new project's changes take effect until the cache is cleared.

NOTE: *Undeploying* a project (using the app-server's own utilities for removing deployed objects) does *not* obviate the need for clearing the cache. See "Removing (Undeploying) Existing Applications" in the previous chapter.

The Clear Cache button is often useful in testing. For example, if you are running in-house benchmark tests to determine which of various cache settings is optimal for a given set of conditions, you would probably want to zero out the cache between runs.

Performance Tuning

Performance optimization is a complex subject because of the many variables involved and the non-obvious interactions between them. There are few hard-and-fast rules. Some issues to be aware of include the following:

- Larger cache sizes may improve application performance, but those gains can be offset by the larger amount of time spent in garbage collection (which is under control of the VM, not Composer).
- In an LRU-governed system, larger cache sizes may not have a dramatic effect if the VM is already using generational garbage collection (as is the case on the HotSpot server VM by default).
- Incremental (as opposed to generational) garbage collection can be turned on via a VM param. You may want to test performance with and without incremental GC enabled.
- *Always be sure the same VM is used on production machines and performance-test machines.* If you tune against a particular VM and then redeploy to a different VM, performance may not be what you expected.
- Be sure the VM *command-line params* used in testing are exactly the same as those on the final target machine.
- Garbage-collection algorithms generally change with each new release of a VM, so be sure to retest every time a new VM release comes out.
- Tuning requirements will differ significantly depending on whether your applications are I/O bound, compute-intensive, or memory-intensive. Deploying a new project into a set of existing projects may alter the mix of dependencies and change the performance of other apps, because the newly deployed services may be I/O-bound, whereas the preexisting services might be compute-intensive.

The only way to know which cache and pool settings are best for a given set of apps is to test.

Connection Pools

In a client/server system, one of the most resource consumptive operations is *connection management*. Allowing each transaction to open and close a connection for each request usually introduces significant overhead. To minimize this overhead, Composer Enterprise Server allows you to exploit the *connection pooling* features of your application server.

It's important to make a distinction between database connection pooling and other types of connection pooling. In general, *database* connection pooling is under the control of the app server, whereas other types of pooled connections (such as 3270 connection pools) are under the direct control of Composer. In the database case, you should consult the documentation for your app server for information of a more detailed nature than will be presented here. (The different app servers, such as Novell exteNd, WebLogic, WebSphere, etc., have different setup and administrative capabilities for managing and creating database connection pools.)

Database Connection Pools

In the Novell exteNd Application Server, database connection pools are identified by *database name*. To take advantage of the server's connection pooling, the Connection Resource for the target database must have the pool name specified. You will want to coordinate with your app server administrator on this at design time, when setting up Connection Resources for your JDBC components.

Logon Components and Non-Database Connection Pools

For connections to non-database resources, Composer Enterprise Server provides connection pooling capabilities that augment those of the application server. Composer Enterprise Server's connector-specific connection pools are configurable and manageable through separate console pages.

Some of the Composer connectors (chiefly those that emulate terminal sessions: 3270, 5250, Telnet, etc.) offer the ability not only to pool connections, per se, but to log in to a particular "start page"of an application or system (which sometimes involves navigating past several screens). The ability to pool properly pre-positioned (by "start page") connections is afforded by so-called Logon Components, which you build as part of your project in Composer at design time.

In order for Logon components to work properly, their existence needs to be made known to the application server as well as to Composer Enterprise Server. If your project uses Logon Components, you should do the following after deploying your project to the server:

➢ **To enable the use of Logon Components:**

1. Locate the Composer deployment JAR that contains your Logon Components. This will be a JAR file (bearing the name of your project) located in the **\archives** folder of your staging area's main output folder.
2. Manually copy the JAR file to the app server's **\lib** folder.

3 Follow the app-server vendor's recommendation for putting the JAR file in your server's classpath.

NOTE: If you're using Novell exteNd Application Server, you can add appropriate $SS_LIB entries in **agjars.conf** after copying the JAR files to the **lib** directory of the app server

4 Restart the server.

If you want to go ahead and initialize the logon components (thus opening all pool connections and bringing them to the proper startup screen), continue to the next two steps. Otherwise, if you are okay with letting connections and logons happen in real time as they are needed (and taking the onetime performance hit associated with that), you can skip the next two steps.

5 Navigate to the Composer runtime console (using your web browser) and click into the console for the particular Connect product in question.

6 Click the **Initialize Connection Pool** button. (This step needs to be done every time you start the server, if you want connections to be set up before going live. Otherwise, there will be a onetime speed hit as individual logon connections "start up" one by one, on demand.)

The architectural and other particulars of various types of pools differ somewhat depending on the type of back-end system involved. These issues are discussed in greater detail in the various individual User's Guides for the various Composer Connect products (e.g., 3270, 5250, CICS RPC, JMS). See the appropriate guide for more information.

Proxy Servers

If your service will be running inside a proxy server, you will need to inspect (and possibly hand-edit) certain settings in your *xconfig.xml* file.

NOTE: There are two xconfig.xml files: One for design time, and another one on the server. The design-time file can be found under **Composer\Designer\bin**. The server-side file can be found under **AppServer\Composer\lib**. Be sure Composer is not running when you make hand edits to the design-time file. (Composer overwrites the file on shutdown.) Likewise, make edits to the server-side version of this file when the server is stopped. Then restart the server.

At design time, you can modify a project's proxy-server settings in exteNd Composer via the **Designer** tab on the **Tools** > **Preferences** dialog. (See the *Composer User's Guide* for details.) When you shut down Composer, **xconfig.xml** is updated for you with respect to proxy-server settings that you made in **Tools** > **Preferences**.

On the server, you need to inspectand/or edit **xconfig.xml** manually in order to "sync up" the runtime proxy server parameters with those you used at design time. Simple go to your **AppServer\Composer\lib** folder and open **xconfig.xml** file with a text editor. Look for the PROXYSERVERINFO tag. The child elements under this tag allow you to fine-tune your proxy settings. Edit them as necessary (with the server shut down), then restart the server.

NOTE: Be sure the USEPROXYSERVER element is set to "ON" if your app will be running inside a proxy server at runtime.

Here is an example of what the relevant section of **xconfig.xml** looks like:

```
<PROXYSERVERINFO>
      <USEPROXYSERVER Desc="If on, the additional PROXY options
are enabled (valid values are on | off)">on</USEPROXYSERVER>
      <HTTPPROXYHOST Desc=" For Doc I/O, HTTP Actions etc., if
network uses a proxy enter name here."></HTTPPROXYHOST>
      <HTTPPROXYPORT Desc="Port number HTTPPROXYHOST listens
on.">80</HTTPPROXYPORT>
      <HTTPNONPROXYHOSTS Desc="List of hosts that do not
require a Proxy.  Each hostname must be seperated by a pipe
&apos;|&apos;.">localhost</HTTPNONPROXYHOSTS>
      <FTPPROXYHOST Desc=" For Doc I/O, HTTP Actions etc., if
network uses a proxy enter name here."></FTPPROXYHOST>
      <FTPPROXYPORT Desc="Port number FTPPROXYHOST listens
on.">80</FTPPROXYPORT>
<!-- Note: The following section applies only if you are
in a Windows NT Lan Manager (NTLM) security environment -->
      <NTLMCREDENTIALS>
        <NTLMUSER>MyUserName</NTLMUSER>
        <NTLMPWD>aEPUqn2YTUV+s0y/AXHwBA==
</NTLMPWD>
        <NTLMDOMAIN/>
        <PROXYNTLMPROTECTED>on</PROXYNTLMPROTECTED>
      </NTLMCREDENTIALS>
    </PROXYSERVERINFO>
```

Note that if your proxy server requires the use of NTLM Authentication, you will need to copy the NTLMCREDENTIALS portion of the PROXYSERVERINFO block (see above) from your design-time **xconfig.xml** file to your server-side **xconfig.xml** file. This block will exist in your design-time **xconfig.xml** file if and only if you have set your NTLM credentials in the dialog at **Tools > Preferences > Designer > Advanced > Setup.** (You may have to exit Composer in order to see the changes show up in **xconfig**.)

Security Roles

Security Roles (a J2EE feature supported by most app servers) provide a highly granular, inheritance-based mechanism by which you can set and enforce access privileges to deployed services that use connections and connection pools. With security roles, constraints can be placed on HTTP actions for particular URL patterns. Roles are also common in database connection pool scenarios.

Security Roles for container-scoped objects are created and administered at the application-server level (rather than in Composer). You should consult your app server documentation for detailed information on how to set up and manage roles on your particular server. In Composer, you use role names to identify a particular service with a role so that when the service acts as a client (to obtain connections, invoke beans, etc.) it can identify itself appropriately.

Most of the service-trigger property sheets in Composer's design-time environment have a field in which you can specify the Role required in order to run the servlet/trigger in question.

NOTE: Service-trigger property sheets are visible only in Composer Enterprise Edition.

When you specify a Role name in a trigger property sheet, you are essentially limiting access to the Composer service. The role of the caller must match the Role required by the service, *or* it must inherit from a role with appropriate access rights, in order for the caller to invoke the target service. In this scenario, the Composer service is the *target* of the request and uses the role mechanism to decide whether the caller is qualified to trigger the service.

You can also specify a "Run As" role for Composer services that will execute other services. In this scenario, the Composer service is the client, rather than the target. The "Run As" role gives the Composer service a Role (an identity for security purposes) to be known by when it calls other services.

Publishing XML Resources

When establishing a business-to-business process, you may need to publish (or expose) certain files that are required by other services, or perhaps by your business partners. Examples of these files include XSL style sheets for rendering an invoice and DTD/schema files for validating documents sent by your site.

For management and maintenance purposes, it is usually more effective to prepare these files in their own dedicated JAR and deploy them to the application server. A URI can then be associated with the JAR and its contents published .

The use of special-purpose JARs can also be an effective strategy for resource files needed by your services, since they allow you to deploy and maintain ancillary files (and the services that use them) separately. In creating special-purpose JARS, you need to plan ahead and indirect all references to these resources through exteNd Project Variables.

Publishing Java Classes

You may find it convenient or necessary to use non-Composer-built Java classes or JARs in your service. If you do require additional Java classes in your application, you must make them available (visible) to Composer Enterprise Servce and/or the app server.

If your JARs or classes need to be visible to Composer Enterprise Server, you can edit or create <JAR> elements under the <RUNTIME> block of *xconfig.xml*. (You can locate the xconfig.xml file for the runtime environment in Composer Enterprise Server's **\lib** directory. On the design-time side, look in Composer's **\bin** directory.)

NOTE: You must do this when the server is not running, since Composer overwrites **xconfig.xml** at shutdown.

If JARs need to be visible to the app server, and you're using Novell exteNd Application Server, you can add appropriate $SS_LIB entries in **agjars.conf** and copy the JAR files to the **lib** directory of the app server; or you can add classes directly to the server's application database.

Other application servers have their own classpath exposure points, generally involving .bat or .sh files and/or config files and/or custom environment variables. You can read about these in the appropriate vendor's documentation.

For development purposes, you can always set the system environment classpath variable to point to your classes or JARs, using operating-system utilities. This should be done only for development work, however. In a production environment, you should limit the scope of JAR/class access to just the applications that need access.

Controlling Access to JAR and Class files

In J2EE, there are five ways in which JARs and/or classes can be installed such that they can be found by client processes within an app-server environment:

- As individual classes within a web archive's **WEB-INF/classes** folder. These classes are visible only to processes that live within the same archive. If the classes are general-purpose utility classes, this may not be the best location because the classes might not be functionally related to the archive that contains them. A higher-level scope might be more appropriate so that the classes do not need to be put inside multiple WARs that need them.

- As a JAR file within a web archive's **WEB-INF/lib** folder. Again, this is a good place to put utility classes functionally related to the applications in the WAR. But since these JAR files will be visible *only* from within the WAR, this is not a good place for utility JARs that might be needed by multiple modules. You could end up putting multiple copies of the JAR inside numerous WARs, creating a maintainability nightmare.

- As individual classes within an EJB module. Although the classes are visible from other modules that use a manifest file, this is not something you should strive for, because the utility classes may not be functionally related to other code in the EJB module.

- As a JAR stored within the enterprise application archive (the deployment EAR). The classes are then visible to any module within the application that has a valid manifest file. This is usually a good solution, as it keeps the classes neatly packaged in their own JAR file, which is usable by any services in the EAR. In Composer, the easiest way to accomplish this kind of JAR-within-EAR packaging is to bring a JAR into your project at design time using the JAR Resource wizard. (See the chapter on Resources in the *Composer User's Guide*.) From that point on, the JAR gets deployed with your project automatically.

- As JARs or individual classes on the application server's global classpath. This is by far the easiest solution, since it makes classes visible to *any* applications running on the server. But from a design standpoint, it's a bad idea, for the following reasons.

    - **Portability issues:** Because the classes live outside of the EAR or WAR, they represent files that must be copied along with the project. (The project is no longer self-contained.) It also means changing the global classpath of each server to which the project or JAR is deployed.

    - **Compatibility and Maintainability issues:** It forces all client processes running on the server to use the same version of the classes. If the external classes are updated, all client applications must be upgraded and/or retested.

    - **Visibility issues:** The classes are visible to *all* applications running on the server. This is usually not what you want.

The classpath mechanism is a high-level, coarse-granularity mechanism for controlling class and package visibilities. If the goal is to restrict runtime *access* to code rather than design- and runtime *visibility* of code packages, it may be appropriate to consider using the programmatic and/or declarative role-based security models available for EJBs and WARs. (WAR security is a J2EE 1.3 concept.) If remote method invocation is an option, many access-control models are available.

The issue of how best to share “shared code” is a notoriously difficult one, regardless of the control mechanism(s) available. As with performance tuning, there are no hard and fast rules that apply for all situations.

4 The Runtime Framework

Most of the time, you will find Composer's native deployment facilities and packaging options more than adequate to meet the architectural requirements of your business applications. But if your development needs are such that it's essential to be able to manipulate Composer-built services on a programmatic level, you will need to know how to write code that leverages Composer's Framework API for low-level Java integration.

The Composer framework is a set of classes (in source code form) for working with, or extending, Composer runtime objects. Its features are discussed in some detail later in this chapter.

In many cases, you can create your own custom service-trigger objects without hand-writing any "setup" code. Novell exteNd Director has code-generation wizards that can create servlet, EJB, JSP, and Java stub files for you, which you can then customize. (See the Deployment chapter of the main *Composer User's Guide* for more detailed information on how to use these wizards.) But to fully understand the generated skel-code, you need to be familiar with the basic architectural assumptions and API requirements of Composer's runtime layer. The information in this chapter will give you the essential background info you need in order to create classes that interact with Composer runtime objects.

NOTE: This chapter is aimed at intermediate-level (or higher) Java programmers who are interested in understanding the application programming interface for code-level access to Composer runtime objects. To benefit from this chapter, you should be thoroughly familiar with servlet and bean programming, and J2EE app server runtime idioms in general.

This chapter will be of help to you if you need to:

- Invoke Composer services programmatically from your own Java classes
- Augment existing Composer "data input" functionality by providing your own support for transports, protocols, or data formats not natively supported by Composer

- Create service triggers that respond to *events* not natively supported by Composer's existing trigger types
- Obtain direct access to a service's *output* art runtime so that you can perform custom post-processing of data or do some kind of custom dispatching of data, etc.

NOTE: The following discussion deals with runtime issues only. A software development kit (SDK) for creating your own pluggable *design-time* artifacts in Composer is available by special request through the Novell exteNd marketing organization.

Composer Runtime Architecture

The core functionality of Composer Enterprise Server is provided by the classes in **xcs-all.jar** (in Composer's **\lib** directory, under the app server install path), plus the three dozen or so accompanying technology-specific JAR files in the **\lib** directory. The classes in **xcs-all.jar** provide all of the essential "core services" your deployed Composer apps need in order to run on the server, including:

- Instantiation of service objects
- Data conversion (preprocessing) in advance of service execution
- Actual execution of service logic
- Basic support functions, like XML parsing, XSL processing, etc.
- Access to app-server services
- Support for various kinds of connectivity (LDAP, JDBC, etc.)
- Caching and cache management

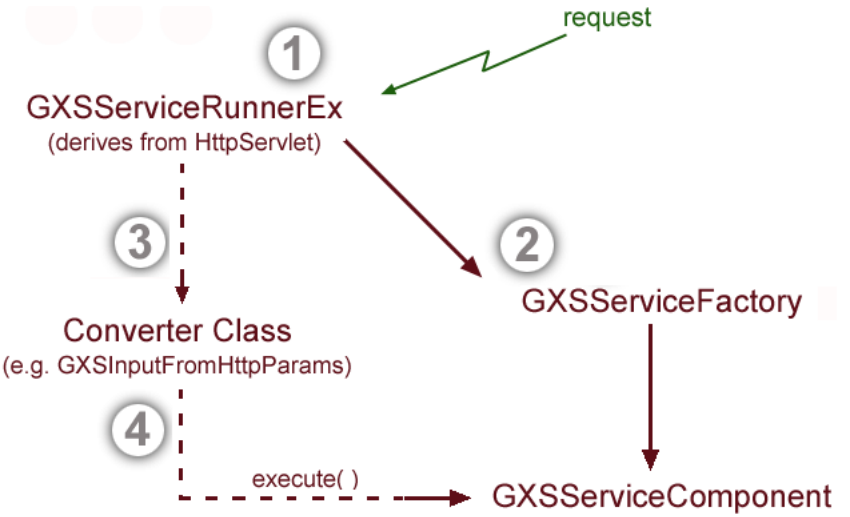
Instantiation and execution of service objects is done through decoding and deserialization of the metadata stored in your deployment archives. When you create a service or component in Composer (design time), you are actually creating an XML file that wrappers the actions in your service or component's Action Model. If you've ever examined the contents of a Composer-created deployment archive, you will probably have noticed that it contains no compiled classes (except if the deployment involves EJBs).

Instead of bytecode, each action in each component's Action Model consists of a metadata description. Composer Enterprise Server understands how to convert that description into executable code at runtime. The classes that do this are opaque: They are not exposed in the Framework API (see below), except for the main *execute( )* method of `GXSServiceComponent`.

Invocation of a Composer service typically occurs through a servlet. But (again) you'll notice there are no servlets in your deployment WAR or EAR. Invocation is handled by a "master servlet" already residing on the server. Your deployment archive contains only a metadata description of how to call the server-resident "trigger servlet." (That description is in the **web.xml** file in the WAR.) The metadata description contains *initialization parameters* for the servlet. Those parameters include the name of the service that needs to be run, the name of the "converter class" that should be used for preprocessing arriving data, whether to instantiate the service as an EJB, etc.

Typical Request-Handling Scenario

From Composer Enterprise Server's point of view, the events that typically accompany invocation of a Composer service include the following:



1 A request arrives at the app server: e.g., XML arrives via HTTP POST. The server notifies the appropriate servlet, in this case **GXSServiceRunnerEx** (a pre-installed, always-present Composer Enterprise Server class that handles most servlet-based requests for Composer services).

2 The service-runner servlet uses Composer Enterprise Server's **GXSServiceFactory** class to obtain an instance of the desired kind of service (represented by the **GXSServiceComponent** shown above).

3 The service runner calls on the appropriate *converter class* (one of several core Composer Server utility classes) to fetch arriving data and put it in String or String-array format. Converter classes are discussed in more detail below.

4 Finally, the service runner calls the service component's *execute( )* method. In the typical case, this method returns a Java String containing the XML output of the service. (Various overloaded versions of the method exist, each with its own return type.)

Once the service has finished executing, the servlet performs any necessary post-processing on the output data (for example, last-minute XSL transformations), in its *processResponse( )* method.

There are many possible variations on the scheme just described. The above diagram describes *one* common scenario, involving servlets and HTTP requests. It is intended to illustrate important Composer architectural idioms, such as:

- The use of a "service runner" object (in this case, a servlet) to run a Composer service
- The use of a *factory* to obtain the instantiated service. Delegation through a factory object makes it possible for Composer to do behind-the-scenes housekeeping (including things like cache management) in a way that's transparent to the service runner. It also simplifies working with EJB deployments, since the service factory can obtain a service as a regular Java object or as an EJB, based on the request parameters.
- The separation of data-prefetch logic from service invocation logic by means of *converter classes* (which handle the details of collecting XML input from various kinds of HTTP payloads)

Obviously, not all data travels by HTTP, and it's not always convenient to invoke services from a servlet. Other scenarios need to be taken into account.

Alternative Request-Handling Scenarios

One useful variation on the above invocation scheme is afforded by the *GXSServiceComponentBean* class, wherein a bean implements the *IGXSServiceRunner* interface. The *GXSServiceComponentBean* provides extra isolation between the client/request layer and the invocation-target layer, so that it becomes possible for a single type of Java object (the bean) to field requests from many potential types of client objects (servlets, JSPs, arbitrary Java objects). Experienced developers will recognize features of the well-known Proxy and Facade design patterns in this approach.

Remote access to Composer services can also occur through EJBs. The *GXSEJBServiceComponent* class implements *javax.ejb.EnterpriseBean, IGXSServiceRunner, java.io.Serializable,* and *javax.ejb.SessionBean*. Likewise, there is an EJB equivalent of *GXSServiceComponent*, called *GXSEJBService*. Enterprise Java Beans make possible the use of any number of well-known design patterns.

In addition to the familiar "request-response" paradigm, of course, it's possible to enlist Composer services in other operational flows. For example, you might have a Composer service that starts up in response to a scheduling daemon of some kind and executes at regular timed intervals. It might not use any input data; it may or may not produce any output. Perhaps it performs a recurring maintenance function. This type of specialized invocation scenario can be supported through the use of a custom trigger object (your own, or derived from a framework object) that implements the *IGXSServiceRunner* interface.

Source code for many of the classes and interfaces just mentioned can be found in the Composer Enterprise Server framework distribution archive, **xcs-src.jar** (see next section). The main classes are discussed in more depth below, but for definitive information you should consult the source code or the Javadoc.

Framework Classes

To facilitate working with Composer deployment and runtime objects, Novell provides a set of *framework classes* that can be used to create custom Service Triggers for Composer services, alter the Composer JSP tag library, change the way data is passed, etc. This framework comprises a runtime API for working with Composer services.

Where to Find the Source Files and JavaDoc

You will find the framework files in the **AppServer\Composer\lib** path under your main **\exteNd** install directory. Look for these two files:

- **api-xs.zip:** This archive contains the JavaDoc files (HTML) for the framework API.
- **xcs-src.jar:** This archive contains Java source code for the approximately 130 classes that make up the framework. (Included in this set of files are the sources for the custom JSP tag library that comes with Composer. For a description of the tag library, see the appendix in the main Composer User's Guide.)

Packages of Interest

Unless you have unusually far-reaching requirements, it's unlikely that you will work with more than a handful of the 130+ classes in **xcs-src.jar.** Nevertheless, a great deal of useful example code can be found there for working with Composer services using servlet technology, EJB technology, SOAP, JSP taglib, transaction managers, etc.

Some of the more interesting packages include:

- **com.sssw.b2b.xs.deploy.wl70:** Helper classes to install J2EE components into the WebLogic Server 7.0, utilizing capabilities of the *DeployerRuntimeMBean* class.
- **com.sssw.b2b.xs.deploy.ws50**: Support classes for deploying to WebSphere, utilizing AppManager features.
- **com.sssw.b2b.xs.bean**: This package contains Java beans that can instantiate and utilize a deployed Composer service. The classes provide for separation of input *conversion* from component *execution*.
- **com.sssw.b2b.xs.ejb:** This package provides an EJB session bean class for obtaining remote access to Composer service components, as well as home and remote interfaces for same.
- **com.sssw.b2b.xs.service.conversion:** Contains various helper classes for obtaining XML data by way of various transports and packagings. (These classes will be discussed in further detail below.)
- **com.sssw.b2b.xs.mail:** Contains classes that make an entry point from SMTP/MIME/POP3 to deployed services.
- **com.sssw.b2b.xs.tl:** JSP custom tag library implementation.
- **com.sssw.b2b.xs.deploy2.tc4:** Deploy handlers for Tomcat 4.1 platform.
- **com.sssw.b2b.xs.soap:** Provides an implementation of a service trigger that responds to SOAP requests, utilizing Novell exteNd WSSDK technology.
- **com.sssw.b2b.xs.util:** A grabbag of utility classes, including classes to manipulate JARs, a vulture class that watches a certain directory for incoming files, a class to represent the *manifest.mf* file found in Java archives, and classes with miscellaneous static convenience methods.

Static Constants

See the file called **constant-values.html** in **xcs-src.jar** for a comprehensive list of constants used in the framework classes.

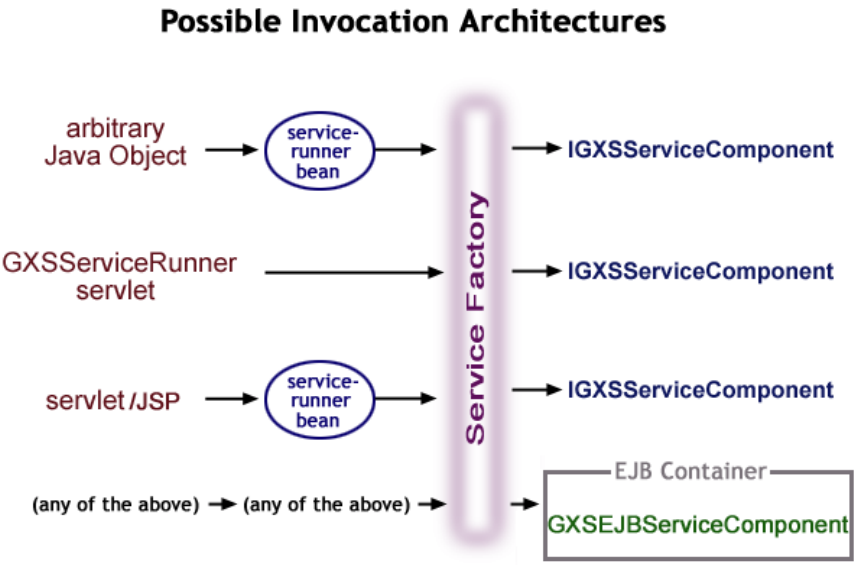
What Types of Programming Needs Does the Framework Address?

The framework allows you to use your own objects to instantiate and execute Composer services. This capability can be important for many development scenarios. For example:

- You can use your own objects to perform custom *pre-processing* of data (perhaps converting non-XML data to XML) before passing it to a Composer service.
- You can *post-process* a service's output in some custom fashion, perhaps altering its mime-type.
- The framework makes it easy to augment Composer's invocation layer. For example, you might have legacy CGI scripts (in Python or PHP, say) that need to be able to call Composer services directly.
- If your development efforts involve operating-system-level calls, you may have C++/Java crossover points that require direct access to Composer services.
- The framework also makes it easier to customize your deployments to take advantage of special app-server services. This can sometimes be important if you're deploying to a platform that's not currently supported by Novell, or you need to "bridge across" to a non-J2EE server API of some kind.
- For performance profiling, you may want to create test routines that can call Composer services directly (eliminating servlet-engine and network-stack overhead) so that you can benchmark different cache configurations, for example, without clouding the results with non-cache-related issues (browser/router/proxy latencies and such).
- If you need to implement certain design patterns in your J2EE projects, it might be necessary (or convenient) to extend various framework classes.

High-Level Architecture

The framework affords a great deal of flexibility in choosing how to invoke a service. A few of the possible choices are depicted in the diagram below.



The choice of how to set up your invocation layer will probably be dictated by architectural concerns related to:

- Whether you are composing large, distributed web apps with reusable components, or small, “low-cost” apps that are self-contained
- Whether you need to support remote invocation across machines (via RMI rather than SOAP)
- Whether your data will mostly arrive by HTTP as opposed to other transports
- The need to implement certain J2EE design patterns
- Possible enlistment of services in transactions
- Your personal programming style

The invocation patterns shown in the foregoing diagram are all supported, in one way or another, by the design-time deployment options of Director and Composer. If you are using the framework, it’s presumably because you need to *customize* some aspect of the invocation layer (by extending one or more of the classes shown). That’s what this discussion will focus on.

Input and Data Conversion

Most (but not all) Composer services operate on input data of some kind. Composer services expect to receive input data (if any) in one of the following forms:

- XML string (java.lang.String containing raw XML)
- A Java array of XML strings
- A DOM object (of type org.w3c.dom.Document)
- An array of DOM objects
- A pair of String arrays: one representing SOAP body parts, another representing SOAP header parts.

If your input data will be arriving via HTTP, you may find it convenient to use or extend one of the framework's existing converter classes, which are designed to handle the most common HTTP transport scenarios. (See the Javadoc and/or source code for the **com.sssw.b2b.xs.service.conversion** framework package.)

Whether your input data arrive by HTTP or not, and whether you choose to use the framework converter classes or not, your code must be prepared to pass input data to your service in one of the formats described above.

Service Names within Framework Objects

When referring to a service name within a framework object (such as a service runner servlet), you should use only the full-context name of your service: That is to say, you should combine the deployment context with the service component name.

The following is an example of a fully qualified service name:

```
com.yourcompany.composer.ProductInquiry
```

Where:

- `com.yourcompany.composer` is the deployment context specified during deployment
- `ProductInquiry` is the Composer service component name

NOTE: Novell recommends, as a best practice, that you include "composer" in the deployment context of every Composer-created artifact, and "director" in the context of every Director-built artifact. This is not only to provide namespace separation of artifacts that might be built by different development team members working remotely, but to make debugging easier. (At stack-trace time, it's valuable to be able to see, at a glance, which product the artifact was created in.)

Obtaining a Service Instance

You will generally use the static *createService( )* method of the *GXSServiceFactory* object to obtain a reference to a so-called service component This overloaded methods comes in three flavors, with signatures as follows:

```
IGXSServiceComponent createService(java.lang.String
  fullServiceName)

IGXSServiceComponent createService(IGXSServiceRunner aOriginator)

IGXSServiceComponent createService(javax.naming.InitialContext
  aContext, java.lang.String aJNDIName)
```

The first case is simplest: You can obtain a (non-EJB) service by name. In the second case, the caller (an *IGXSServiceRunner*) passes a reference to itself; the factory inspects the caller's properties to obtain initialization parameters, then instantiates and configures the service.

The third method produces a service component as an EJB (assuming the service was deployed that way to begin with). The factory needs to know the initial JNDI context and JNDI Name of the service's home interface in order to obtain a reference to the EJB (or its accessor object). After that, the factory takes care of any communication with the EJB container.

Executing the Service

The code for executing a service directly is straightforward. First, obtain an instance of the desired service by means of a service factory object. Then call the *execute( )* method of the service object. The *execute* method returns the service's output document(s) as native XML in String form.

Code for calling a service can be as simple as:

```
String inputDoc =
       “<?xml version=\"1.0\" encoding=\"UTF-8\"?><root/>”;
String outputDoc = “”;
String serviceName = "com.acme.composer.ProductInquiry";


try {
// Obtain an instance of the desired service:
IGXSServiceComponent myService =
       GXSServiceFactory.createService( serviceName );


// Execute the service:
outputDoc = myService.execute( inputDoc );
}
catch( GXSException gxsEx )
 {
```

```
    // Do something with exception
}
```

Using this kind of code, you can invoke a Composer service from any kind of custom Java object (not just a servlet). Of course, it's the caller's job to obtain the input data for the service, so it can be passed directly in the execute method. In the bare-minimal code shown above, you are passing a single input document as a native-XML string. If you need to pass more than one document, perhaps as a DOM object (i.e., an object of type *org.w3c.dom.Document*), you can call one of the other variants of execute( ) or executeEx( ); see the discussion under "Data-Passing Options" below.

Delegating Service Calls Through GXSServiceComponentBean

Instead of calling *execute( )* on a factory-obtained service instance, you might find that a more flexible and architecturally robust way of doing things is to delegate service operations through an accessor object: namely, a bean. (Not an EJB, but a regular Java bean.) In this strategy, you instantiate a general-purpose bean directly, use the bean's setter methods to specify the desired service name, input document(s), and other parameters, then call *execute( )* on the bean. (The bean then delegates the call to the service.)

The framework provides a utility bean for this purpose, in a class called *GXSServiceComponentBean*. Code for utilizing this bean typically looks similar to that shown below.

```
private static final String SERVICE_NAME =
     "com.composer.MyService";

// Legal values here are "Normal" or "EJB":
private static final String SERVICE_TYPE = "Normal";

// Instantiate the bean
GXSServiceComponentBean lService =
     new GXSServiceComponentBean();

// Configure it
lService.setInputXMLDoc( aXML );
lService.setServiceName( SERVICE_NAME );
lService.setServiceType( SERVICE_TYPE );

// Now execute the service:
try {
            lService.execute();
}
catch ( GXSException e ) {
            System.out.println( e );
        }
```

```
// Obtain the service's output:
String myOutput = lService.getOutputXMLDoc();
```

The bean mechanism offers a great deal of flexibility. The bean itself is generic: It can be "configured" dynamically to bind to any service. It implements the *IGXSServiceRunner* interface, which means that through a variety of setter methods, you can specify XSL resource info, converter class name, and other config parameters for the service before invoking it. Likewise, you can use a wide variety of "getters" to obtain information back from the service after it executes. In addition, the *GXSServiceComponentBean* class has utility methods, such as *getXPath( )* and *findDocByPartName( )*, that can be helpful in manipulating output data.

The service-runner bean (*GXSServiceComponentBean*) allows you to specify, via *setServiceType( )*, whether to use EJB access to obtain and execute the target service (assuming it was deployed in EJB fashion), or non-EJB ("Normal") access. This hides some of the complexity of working with services deployed as EJBs.

The custom tag library used in Director-generated (and Composer-generated) JSP code is built around usage of the *GXSServiceComponentBean* object. (Source code for the tag library itself is part of the framework.)

NOTE: The *GXSServiceComponentBean* class inherits from a utility class called *GXSServiceComponentBase* (which in turn implements the service-runner interface). Consult the source code and/or Javadoc for these two classes to learn more about the numerous setter, getter, and utility methods they offer.

Data-Passing Options

The *execute( )* method on *GXSServiceComponent* is overloaded to allow you to pass and receive XML data in various ways. Variants of this method exist to allow passing more than one input document (as either a String array or an array of DOM objects), or passing input as a *java.io.Reader.* In each case, the return type mimics the input type.

There is also an overloaded method called *executeEx( )* that differs from *execute( )* in that it returns a *GXSExResponse* object, which is a lightweight wrapper object for responses from SOAP services that might involve one or more output parts and/or header parts.

The various signatures of *execute( )* and *executeEx( )* are shown below, along with a brief description of the intended usage..

```
java.lang.String execute()
```

Executes a Composer service that does not expect an input document.

```
org.w3c.dom.Document execute(org.w3c.dom.Document aInputDoc)
```

Executes the Composer service using the supplied DOM.

```
org.w3c.dom.Document execute(org.w3c.dom.Document[]
aInputDocs)
```

Executes the Composer service using the supplied mulitple DOMs.

```
java.io.Reader execute(java.io.Reader xmlIn)
```

Executes the Composer service using the supplied XML Reader.

```
java.lang.String execute(java.lang.String xmlIn)
```

Executes the Composer service using the supplied XML string.

```
java.lang.String execute(java.lang.String[] aInpDocs)
```

Executes the Composer service using the supplied XML strings.

```
GXSExResponse executeEx(java.lang.String[] aInpDocs)
```

Executes the Composer service using the supplied XML strings.

```
GXSExResponse executeEx(java.lang.String[] aInpDocs,
java.lang.String[] aInpHdrDocs)
```

Executes the Composer service using the supplied XML strings.

Service Triggers

A *service trigger*, broadly speaking, is any object responsible for obtaining a service instance and executing it. In the framework, the principal trigger objects are *GXSServiceRunner*and *GXSServiceComponentBean*. The former is a servlet; the latter is a general-purpose bean.

The *GXSServiceRunner* class inherits from *javax.servlet.http.HttpServlet* and implements the *IGXSServiceRunner* interface (as well as *java.io.Serializable*). The *GXSServiceRunnerEx* class differs from *GXSServiceRunner* in its ability to deal with one *or more* input documents.

GXSServiceComponentBean inherits from *GXSServiceComponentBase.* Both implement *IGXSServiceRunner* as well as *java.io.Serializable.* The parent class, *GXSServiceComponentBase,* has many getter and setter methods, allowing you to fine-tune its functionality dynamically. It is not limited to handling HTTP requests.

If you are implementing a trigger that handles data arriving via HTTP, a convenient starting point may be *GXSServiceRunner* or *GXSServiceRunnerEx*.

Of course, strictly speaking, it is not necessary for you to extend any of the framework's preexisting service-runner classes in order to execute a service. In fact, it's not even necessary for your custom trigger object to implement *IGXSServiceRunner*. (See "Executing the Service" for example code that neither extends nor implements framework classes.) Even so, you should understand how these classes and interfaces work.

IGXSServiceRunner

The interface that all framework service-runner objects implement is *IGXSServiceRunner*. This interface has two methods, called *getServiceProperty( )* and *getClassLoader( )*, plus numerous predefined public/static properties (Strings) that are used for parameter discovery at runtime. The *getServiceProperty( )* method takes a String as an argument; the String should match one of the static property strings defined on *IGXSServiceRunner*. The *getServiceProperty( )* method uses the String passed to it as a key to look up information about the service environment.

For example, one of the properties is called *SERVICE_NAME*. The hard-coded (final) value of *IGXSServiceRunner.SERVICE_NAME* is "servicename." If your service-runner object receives this value in a call to *getServiceProperty( )*, the method should return the name of the service that will be called.

The *getServiceProperty( )* method is called by various objects that, from time to time, might receive a reference to your service-runner and might need to look up information about the service your runner intends to run. For example, the *GXSServiceFactory* object has an overloaded method called *createService( )*. One of the *createService( )* methods takes an *IGXSServiceRunner* argument. Using the passed-in service-runner reference, the factory object can inspect properties on the caller to determine how to configure a service instance before returning it to the caller. This same mechanism is used by various data-converter objects in the framework.

As it turns out, your service runner does not need to define lookup values (nor "get" methods) for *all* of the String properties in the *IGXSServiceRunner* interface. Some of the properties are relevant only in specialized scenarios involving (for example) digitally signed XML in SOAP transactions. For most common scenarios, the only "discovery" properties you *must* make available before every call to a service factory's *createService( )* method are the *SERVICE_NAME* and *SERVICE_TYPE* properties. (The latter allows the factory or converter object to discover whether the caller is expecting an EJB, or non-EJB service.)

A bare-minimal implementation of *IGXSServiceRunner* is shown below:

```
class MyServiceRunner implements IGXSServiceRunner
{
   private String mFullServiceName;

   MyServiceRunner(String fullServiceName)
   {
      mFullServiceName = fullServiceName;
   }

   public String getServiceProperty(String aName)
   {
      if( aName == IGXSServiceRunner.SERVICE_NAME )
         return mFullServiceName;
      else if( aName == IGXSServiceRunner.SERVICE_TYPE )
         return IGXSServiceRunner.SERVICE_TYPE_NORMAL;
      else
         return null;
   }

   public ClassLoader getClassLoader()
   {
      return Thread.currentThread().getContextClassLoader();
   }
}
```

Note that if *getServiceProperty( )* is called with an argument other than *SERVICE_NAME* or *SERVICE_TYPE*, the method returns null. It is important to return null here, because the Composer runtime objects that call *getServiceProperty( )* implement default behaviors of various kinds based on a null return value being encountered. If you return a dummy value (such as “Not supported”), you will get unpredictable results.

In addition to *getServiceProperty( )*, your service runner needs to provide an implementation of *getClassLoader( )* for use by factory objects. The implementation shown in above is appropriate for most cases.

GXSServiceRunner and GXSServiceRunnerEx

If your code will be handling HTTP requests, you might want to extend *GXSServiceRunnerEx*. This is the framework’s all-purpose servlet for triggering Composer services.

The following code shows how to extend *GXSServiceRunnerEx*. It implements a custom Java class called *MyComposerServiceRunner*.

```
package com.composer;

import javax.servlet.*;
```

```
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import com.sssw.b2b.xs.*;
import com.sssw.b2b.xs.service.GXSServiceRunnerEx;

public class MyComposerServiceRunner extends GXSServiceRunnerEx
{
    static final String CONTENT_TYPE = "text/html";

   // Overload the following method if you want to
   // override the default converter class architecture

   protected String[] processRequestEx(HttpServletRequest aReq)
throws ServletException
   {
   return super.processRequestEx( aReq );
   }

   // Overload the following method if you want to
   // override default response architecture

   public void processResponse( String xmlOut,
                                HttpServletRequest req,
                                HttpServletResponse res )
                                        throws GXSException
   {
      super.processResponse( xmlOut, req, res );
   }
}
```

The *processRequestEx( )* and *processResponse( )* methods offer convenient hooks for implementing your own special data pre- and post-processing logic. The above code merely delegates execution to the parent's default implementations of these methods. Remove the "super" calls and insert your own code to take over control of pre- and post-processing.

The example class shown above inherits from *GXSServiceRunner*, which in turn inherits from *HttpServlet*. Normal servlet control flow applies. In *GXSServiceRunner*, the following flow occurs:

- *doGet( )*, if invoked, calls *doPost( )*
- The *doPost( )* method calls *initService( )*, which obtains the desired service via *GXSServiceFactory.createService(this)*.
- Also within *doPost( )*, a method named *performProcessRequest( )* is called.

- *performProcessRequest( )* calls *processRequest( )*, which in turn obtains the input data for the service. (To obtain the data, a *GXSInputConverterBean* is instantiated. The bean, in turn, inspects the CONVERTER_CLASS_NAME property of the service runner to determine which converter class to use.) The service's *execute( )* method is then called.
- When *processRequest( )* returns, the method *processResponse( )* executes. This is where data post-processing can be performed. It is also where any OutputStreams that are opened from the *HttpServletResponse* should be closed.

  NOTE: The default implementation of *processResponse( )* contains code for converting XML to HTML (using server-side XSL transformation), based on the value of the HTML_INDICATOR property set by the service runner. Study the source code for *GXSServiceRunner* if you want to see how this kind of data post-processing can be done.

Initialization Parameters

It's important to understand that the default implementation of *GXSServiceRunner* depends on framework methods (specifically, methods belonging to *GXSServiceFactory* and *GXSInputConverterBean*) in which the service runner itself is an argument to the method. When a reference to the service runner is passed this way, it's because the factory object needs access to the caller's properties. The properties in question usually involve configuration parameters of some kind.

For example, when *GXSServiceRunner* calls the *GXSServiceFactory* method *createService( )*, passing 'this' as an argument, the factory uses the servlet reference to find out the *name* of the service to obtain and the *type* of service (EJB or non-EJB). These pieces of information ultimately come from the servlet's initialization parameters (in particular, the params called "servicename" and "xcs_servicetype"). The initialization parameters, in turn, are specified in the **web.xml** file in the servlet's WAR module.

The following listing shows what the **web.xml** servlet entry for the *MyComposerServiceRunner* class might look like. This example assumes that the target Composer service is called *HelloWorld* and that the framework-supplied *GXSInputFromHttpParams* converter class will be used to obtain data from the HTTP request.

```
<servlet>
  <servlet-name>
    MyComposerServiceRunner
  </servlet-name>
  <display-name />
  <servlet-class>
   com.composer.MyComposerServiceRunner
  </servlet-class>
```

```
  <init-param>
  <param-name>servicename</param-name>
  <param-value>com.composer.HelloWorld</param-value>
  </init-param>
<init-param>
  <param-name>xcs_servicetype</param-name>
  <param-value>NORMAL</param-value>
  </init-param>
<init-param>
  <param-name>transform_into_html</param-name>
  <param-value>false</param-value>
  </init-param>
<init-param>
  <param-name>rootname</param-name>
  <param-value>greeting</param-value>
  </init-param>
<init-param>
  <param-name>converterclassname</param-name>
  <param-value>
    com.sssw.b2b.xs.service.conversion.GXSInputFromHttpParams
  </param-value>
  </init-param>
</servlet>
```

Note that the “servicename” init param specifies the complete (full-context) name of the target service, in this case **com.composer.HelloWorld**.

Other parameters are supplied as well, such as “rootname” (to specify the root element name of the XML document that the converter class will create as input to the service), a “transform_into_html” flag to indicate to *GXSServiceRunner* whether to attempt XSL transformation of the output data in *processResponse( )*, and so on.

The important point to note is that if you intend to extend *GXSServiceRunner* or *GXSServiceRunnerEx*, you should ensure that the **web.xml** file for your servlet class specifies, at a minimum, the init params “servicename”, “xcs_servicetype”, and “converterclassname” (and valid values for them), as shown above. The other initialization parameters are optional.

The framework factories “understand” a large number of possible init parameter types: see the properties defined on the *IGXSServiceRunner* interface for a full list. Some of the more commonly used params are shown in the following table. (Required params are in bold.)

IGXSServiceRunner Property Name	Description	Initialization Parameter	GXSServiceRunner method
SERVICE_NAME	The name of the Composer service component.	**servicename**	getServiceName()
ROOT_NAME	The root node that is expected as the input document.	rootname	getRootName()
JNDI_NAME	The JNDI name of the EJB home interface, for the Composer service component.	jndiname	getJndiServiceName()
CONVERTER_ CLASS_NAME	The class that should be used to convert the HTTP request into an XML document.	**converterclassname**	getConverterClassName()
PARAM_NAME	The name of the parameter that contains the input XML document.	xcs_paramname	getxcsParamName()
SERVICE_TYPE	Whether the service component reference is an EJB or NORMAL.	**xcs_servicetype**	getServiceType()
PROVIDER_PARAM	The JNDI provider URI.	providerURI	
CONTEXT_FACTORY	The JNDI context factory.	contextfactory	
HTML_INDICATOR	An indicator used to specify whether the output document will be rendered as HTML.	transform_into_html	getOutputHTMLIndicator()

OUTPUT_XSL	If the output document is being transformed into HTML, this will give the URI of the style sheet. This is **only** necessary if the XSL processing instruction has not been embedded in the output XML document.	output_xsl_URI	getOutputXSL()

If your servlet class will be used in an environment where the **web.xml** init-param mechanism can't be relied upon, you should provide custom implementations of the following methods:

- `getServiceName()` to bind the servlet to the Composer service component (mandatory in all cases)
- `getRootName()` to return the name of the root element to be used if the converter class will be *GXSInputFromHttpParams* (otherwise "root" will be used by default)
- `getServiceType()` should return a string value of "NORMAL" or "EJB", indicating the type of service component that will be invoked (mandatory in all cases)
- `getConverterClassName()` should return the name of a class that implements the `IGXSInputConversion` interface (not mandatory in every case, but recommended as a best practice)
- `getOutputHTMLIndicator()` should return true if the output of the service will be transformed into HTML using the default implementation of *processResponse( )*; false if it will be XML. (Again, (not mandatory in every case, but recommended as a best practice.)

IGXSInputConversion and IGXSExInputConversion

The framework's servlet-based service runner class (*GXSServiceRunnerEx*) makes use of so-called *converter classes* to obtain data arriving via HTTP. These classes are intended to provide a clean separation of "data marshalling and unmarshalling" logic from service invocation logic.

The converter classes in the framework implement the *IGXSExInputConversion* interface (which in turn extends *IGXSInputConversion*). This interface has only one method:

```
String[] processMultipleRequests(HttpServletRequest aReq)
```

As you can see, this method essentially converts a servlet request to an array of XML strings.

NOTE: Since you cannot implement the *IGXSExInputConversion* interface if your custom service runner class does not use (or cannot supply) a *HttpServletRequest* object, this discussion applies only if you are extending *GXSServiceRunnerEx* (or if you are implementing a custom servlet that will eventually get passed to factory objects). If your trigger class does not inherit from *HttpServlet*, you can implement your own scheme for fetching data, and simply pass the data to the service's *execute( )* method.

Most of the framework's converter classes implement a constructor that takes a *IGXSServiceRunner* argument so that the converter can obtain initialization parameters (or other information) from the caller. Study the source code for the framework converter classes if you want to see examples of this technique in use.

Framework-Supplied Converter Classes

The framework contains a number of predefined converter classes (that is, classes that implement the *IGXSExInputConversion* interface). The names of these classes can be specified in servlet init-params, or supplied to the *setConverterClassName( )* method of *GXSServiceComponentBase*.

Converter classes available in the framework include:

- **GXSInputFromHttpContent** — Obtains XML directly from the request's InputStream
- **GXSInputFromHttpMultiPartRequest** — Obtains XML from multipart form data
- **GXSConvertHttpMPReqNonBuff** — Same as above, but uses a non-buffered MultipartRequest. (Note: The MultipartRequest class is defined in the framework. See the relevant Javadoc and/or source code for details.)
- **GXSInputFromHttpParams** — Obtains XML by parsing query parameters off the tail end of the URL in an HTTP GET. Those params are assembled into an XML document on the fly.
- **GXSInputFromHttpSpecificParam** — Assumes that a form has been POSTed, with a field called 'xmlfile' that contains XML.

- **GXSInputFromJavaObject** — This is actually a convenience object for use by Composer JSP taglib methods. It is constructed using a reference to a *GXSServiceComponentBean*. The bean needs to be able to point to a XML String whose variable name is located in an init param called ‘xmldoc’. See source code for details.
- **GXSInputFromSoapContent** — Obtains XML strings from elements under the BODY element of a SOAP request. Every element is accumulated into a String[ ].

You should study the source code for these classes to see how they work before implementing any but the most trivial of custom converter classes. Depending on what kinds of data conversion you need to do, you may be able to extend an existing converter class (and save yourself a lot of coding).

A custom converter class will look something like this.

```
public class MyConverterClass implements IGXSInputConversion
{
   // Attribute that holds the service
// runner for querying parameters.
   IGXSServiceRunner mRunner = null;

   // Constructor to take the IGXSServiceRunner
   // so that the class can retrieve params
   public MyConverterClass( IGXSServiceRunner aRunner )
   {
      mRunner = aRunner;
   }

   // the processRequest method should take
   // an HttpServletRequest as
   // a parameter and return an XML doc as a String:
   public String processRequest( HttpServletRequest aReq )
      throws GXSConversionException
   {
      String lsExpandedDoc = null;
   // ( create or obtain XML . . . )
      return lsExpandedDoc;
   }
}
```

EJB-Deployed Services

The Enterprise Java Bean (EJB) API implements a container architecture designed to facilitate clean separation of logic, data-access, and presentation layers while also providing connection pooling, transaction management, persistence, access control (via Roles), “naming services” (JNDI), and remote invocation mechanisms, so as to free applications from having to implement or manage such features individually.

Composer services can be deployed as EJBs. In Composer Enterprise Edition, a simple drag-and-drop UI exists for designating EJB associations at design time (see the separate *Composer User’s Guide*), such that when you choose a service to deploy as an EJB session bean, you can specify whether it is to be Stateful or Stateless, the transaction participation level (Mandatory, Never, Supports, etc.), and the JNDI name of the service.

Since EJBs cannot be instantiated directly by use of constructors, you must use the *GXSServiceFactory*’s static *createService( )* method to obtain a reference to a service. The signature of the method in question is:

```
public static IGXSServiceComponent
createService(javax.naming.InitialContext aContext,
              java.lang.String aJNDIName)
                                          throws GXSException
```

The returned service object is of type *IGXSServiceComponent*, which means it supports all of the various *execute( )* overloaded signatures discussed previously.

An alternative to using the *GXSServiceFactory* is to utilize the *GXSServiceComponentBean* class, which can act as a kind of “proxy object” for interacting with Composer services. Example code for using this JavaBean was given earlier (under “Delegating Service Calls Through GXSServiceComponentBean”). To use this bean as an EJB-service accessor, follow the procedure discussed before, but specify “EJB” in *setServiceType( )*, and in addition to calling *setServiceName( )* with the name of the deployed service, use *setJndiServiceName( )* to specify the JNDI name that you supplied for the service at deployment time. If you are implementing the *IGXSServiceRunner* interface yourself, you should provide an implementation of *getJndiServiceName( )* in your service runner and vector to it from *getServiceProperty( )* when the latter gets a request for JNDI_NAME.

Getting the EJB Home and Remote Interfaces

The EJB remote interface, called *IGXSEJBServiceComponent*, is located in the **com.sssw.b2b.xs.ejb** package. When deploying an Composer service as an EJB, you will assign a JNDI name to the EJB. It is this name rather than the qualified Composer service name that will be used to get a reference to the EJBs home interface. The name of the EJB home interface for creating the *IGXSEJBServiceComponent* is **IGXSEJBServiceHome**.

When specifying the JNDI name for an EJBs home interface remember that, for the Novell exteNd Application Server, the string "sssw://host/RMI/" needs to be prepended. For example, if you were deploying an EJB into an Application Server called **main.server**, and the JNDI name for the EJB happens to be **com/acme/inventory/ProductInquiry**, then the fully qualified JNDI name would be **sssw://main.server/RMI/com/acme/inventory/ProductInquiry.**

Once the home interface has been retrieved, much like the *GXSServiceFactory's* `createService()` method, a method called `create()` can be invoked which will return the remote interface of the EJB. (This is the closest thing to "instantiating" an EJB that exists in the EJB world.) The remote interface contains several *execute* methods, as described below:

```
java.lang.String execute()
```

Method to execute a Composer service that does not expect an input document.

```
java.lang.String execute(java.lang.String inXML)
```

Method to execute the Composer service against a single XML document.

```
java.lang.String execute(java.lang.String[] asInputStrs)
```

Method to execute the Composer service component using multiple input documents.

```
GXSExResponse executeEx(java.lang.String[] asInputStrs)
```

Executes the Composer service using the supplied XML strings.

```
GXSExResponse executeEx(java.lang.String[] aInpDocs,
java.lang.String[] aInpHdrDocs)
```

Executes the composer service component using the supplied XML strings as inputs and headers.

You will notice that the Reader or Document versions of *execute( )* available in the `IGXSServiceComponent` are not available in the EJB remote interface. This is because neither Reader nor Document is *serializable* and thus neither one is able to appear in a remote method.

Factory to Obtain EJB Home Interfaces

If you want low-level control over EJB access, you will want to know about a factory class called *GXSEJBAccessor*, located in the **com.sssw.b2b.xs.sssw** package. It contains two methods to obtain an EJB's home interface from a Novell exteNd Application Server.

One method can be used within a server that does not require authentication; the second provides two extra parameters for username and password.

When using the factory, there is no need to fully qualify the JNDI name assigned to the EJB. The factory creates the fully qualified hostname with the supplied parameters. In the following example, the JNDI name of the EJB is **com/acme/inventory/ProductInquiry**, the Novell exteNd Application Server name is **main.server** and the ports are at their installation default of 80 for HTTP and 54890 for RMI.

```
import com.sssw.b2b.xs.ejb;
import com.sssw.b2b.xs.sssw.GXSEJBAccessor;

public void doSomeEJBStuff() throws java.rmi.RemoteException
{
    IGXSEJBServiceHome srvcHome = GXSEJBAccessor.getHomeBean(
        “com.sssw.b2b.xs.ejb.IGXSEJBServiceHome”,
        “com/acme/inventory/ProductInquiry”, “main.server”,
        80, 54890 );
    IGXSEJBServiceComponent ejbSrvc = srvcHome.create();
    // Do something with the service component
}
```

5 Transaction Management

Composer applications that perform transactions require special planning and deployment. Runtime and deployment issues associated with transaction management are covered in this chapter. For a discussion of design-time issues, such as how to use the Transaction Action, see the chapter on Advanced Actions in the *Composer User's Guide*.

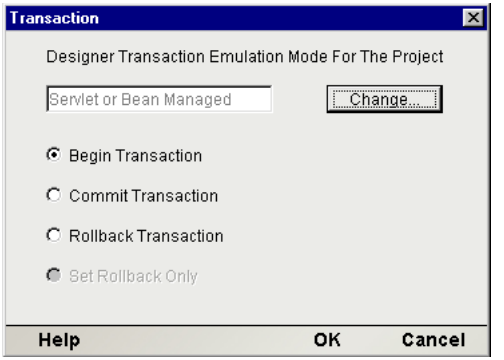
NOTE: JTA and XA -resource transaction features (including Composer's *Transaction Action* as described below) are not supported in the version of Composer that comes with exteNd 5 Suite Professional Edition. Full transaction support is available in the Enterprise Edition version of Composer.

Transaction Control in exteNd Composer

In Composer Enterprise Edition, the *Transaction action* can call any of the defined Java Transaction API (JTA) server-side transaction commands. For example:

- The *begin*, *commit*, and *rollback* commands are available for use in projects that will be deployed as servlets or EJBs with bean-managed transaction behavior.
- The *Set Rollback Only* command is available for use in projects that will be deployed as container-managed EJBs.

These choices, appropriately enabled/disabled, are available from the Transaction dialog (see below), which appears when you create a new Transaction Action in Composer.



Transaction Deployment Considerations for the Novell exteNd Application Server

As described in an earlier chapter, Composer services can be front-ended by servlets, EJBs, or arbitrary Java classes. Each mechanism has important implications for transaction control, as a result of the way transactions are defined in the Java Transaction API (JTA).

Servlet Deployment Considerations

Servlet deployment using JDBC connection pools are recommended when complex transactional behavior is not required, such as inquiry-only services. The primary limitations of Servlet deployments are:

- Declarative transaction control is not allowed. If this is a requirement, use an EJB deployment instead.
- JDBC connections from the connection pool destined for a Servlet are by default set with auto-commit turned on. This means that after each Update, Delete, or Insert statement, the transaction is automatically committed to the database. Subsequent rollbacks will have no effect. There are two ways to change this behavior:

  1. Issue a Begin Transaction command (using a Transaction action), and utilize a subsequent Commit or Rollback command as appropriate.

  2. Check the “Allow SQL Transactions” checkbox for the connection. See “JDBC Transaction Control: Allowing User Transactions” on page 74 for further details.

  NOTE: Nested transactions are not allowed, but sequential ones are.

EJB Deployment

Deploying Composer services as EJBs gives the maximum transaction management flexibility. EJBs are the recommended deployment choice if your application requires a distributed transaction environment where data has to be updated in a number of back end systems. Before examining exteNd specifics for EJB Deployment, it is worthwhile to review the deployment options regarding transactions as indicated in the EJB specification. The following definitions are helpful:

Application is the user of the transaction services, normally the EJB.

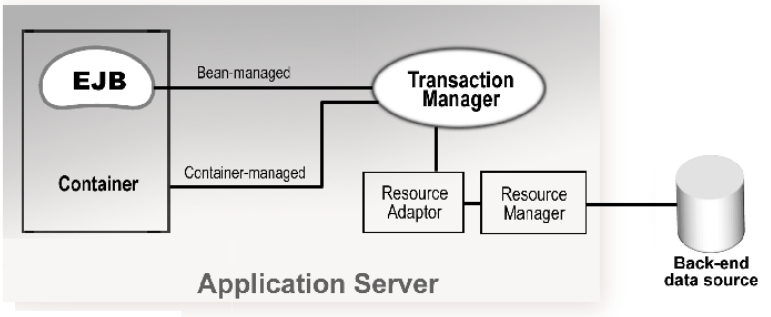
Container is the application server provided context in which an EJB is deployed to and executes.

Resource Manager is the interface to the back end system, such as a database or a message queue.

Resource Adapter is the interface to the Resource manager, such as a JDBC driver.

Transaction Manager is an application server provided object that controls the flow of the transaction, setting up the transaction between all players. This normally involves mapping the high level calls to low level transaction calls to the standard X/Open XA protocol.

See the accompanying illustration.



All container-managed transactions are on a method call basis, while stateful Bean-managed transactions may span method calls. The EJB literature sometimes implies that with EJBs, all transaction management is done behind the scenes and is of no concern to the application developer. Although complex two-phase commit logic is, in fact, performed automatically (and rollbacks will automatically occur if exceptions are thrown), developers still need to have an understanding of how EJB transactions are managed in order to ensure desired application results.

Bean Managed Transaction Demarcation

When an EJB is deployed as a Bean-managed transaction, it is expected to communicate with the transaction manager indirectly via a simplified transaction interface called *UserTransaction*. UserTransaction provides transactional commands such as begin, commit, and rollback. These commands are only available to the Bean if it is deployed as Bean-managed. If they are issued when the EJB is deployed as Container-managed, an IllegalStateException is thrown. Consequently, developers need to know in advance how the Bean is going to be deployed.

Container-Managed Transaction Demarcation

Container-managed Transaction demarcation, also known as declarative transaction support, is a powerful and flexible means for transaction support. The application assembler is free to determine the EJB's transactional behavior post construction. Container-managed transactions are most useful in cases where EJBs utilize other EJBs to get work done. The classic example of this case is a stateless Session Bean calling several Entity Beans to update various tables in a database. Linking their transaction via declarative transaction management greatly reduces the complexity of the code, as any failure in any of the components can automatically roll back the transaction.

EJBs support six different Container-managed transaction types. The most important differentiator among the six is the notion of *transaction propagation*. If one EJB with an ongoing transaction calls another, the transaction may or may not be passed along to the second EJB. If it is passed, and the transaction is subsequently rolled back, then all work done in *all* EJBs within the scope of that transaction are rolled back.

Container-managed transaction types include:

Table 5-2

Transaction type	Behavior
Not Supported	No transaction support is available.

Required	If called with a transaction, it will join, else it will create one.
Supports	If called with a transaction, it will use, otherwise it will run without one.
Requires New	Always creates a new transaction. The callers transaction is suspended until this one completes.
Mandatory	If called with a transaction, it will use, otherwise it throws an exception.
Never	If called with a transaction, it will throw an exception.

In Container-managed transactions, there is no way to call any type of commit. The user can initiate a rollback by calling the `setRollbackOnly()` method on the EJB context. This call is only appropriate in certain situations, however. If the application is deployed as a Bean-managed EJB, or a Container-managed EJB without Transaction support, a call to `setRollbackOnly()` will result in a `java.lang.IllegalStateException`.

Container-managed transactions are a very powerful mechanism to perform complex transaction management in a heterogeneous environment. Such a complex distributed environment requires support from the back end resource manager, the middleware drivers, and the application server.

NOTE: At this time, the Novell exteNd Application Server supports distributed transaction management across connections from a *single* connection pool. Check with the appropriate vendor's documentation if you are using a server other than Novell exteNd Application Server.

XA-Aware Database Drivers

Check that you are using an XA-enabled database driver before using transactions involving database access. Most vendors provide XA and non-XA version of their drivers. If you are not able to use an XA-aware driver, you may still be able to enlist JDBC components in transactions, but you should commence the transaction *before* opening the database connection (i.e., before calling the JDBC component). You should test this scenario, obviously, before relying on it.

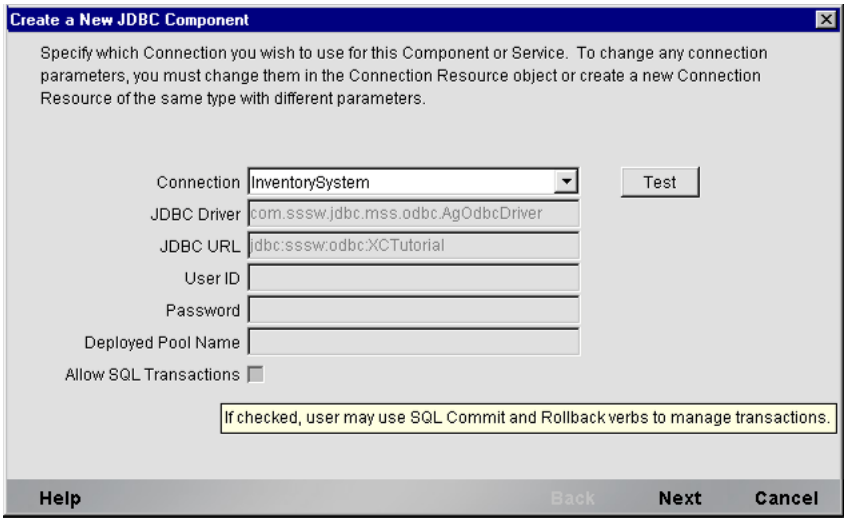
EJB Deployment Considerations

EJB deployment is recommended in situations where complex transactional behavior is required. By default, the Deployment Wizard bases the deployment-mode choice on the current Transaction Emulation Mode (as set in **Tools > Preferences**, using the Designer tab). If the emulation mode you've chosen indicates a bean-managed EJB deployment, the Deployment Wizard will create this type of deployment. Otherwise, it will default to a Container-managed, "Transaction Not Supported" deployment. (One can easily change from Not Supported to Mandatory, Supports, Requires New, or any of the other valid choices for bean or Container-managed transactions using the pull-down menu in the Transaction Attribute field of the EJB-Based Service Triggers Panel of the Deployment Wizard.)

JDBC Transaction Control: Allowing User Transactions

Manual control of transactions is sometimes required. For such situations, exteNd Composer has a special checkbox on the JDBC connection component that allows user-controlled SQL transactions.

NOTE: This is an advanced option, and should only be used if you are comfortable with the details of SQL programming.



Checking the Allow SQL Transactions box does the following:

- It turns auto-commit *off* for the JDBC driver

- It translates all SQL commit and rollback commands to the equivalent JDBC connection calls
- It causes exteNd Composer Enterprise Server to perform a rollback on the JDBC connection *if the last **Execute SQL Action** in the JDBC component was not a commit or a rollback.*

  NOTE: This behavior is important if connection pools are used. When you return a connection to the pool, the pool manager expects to be handed a "clean" connection. If you return a dirty connection (a connection with uncommitted changes on it), undesirable results, such as table locking and transaction scope mismatches, can occur. To prevent this, Composer detects a dirty connection, and attempts to clean it by issuing a rollback, unless the user has explicitly commanded a commit. Bottom line: It is vitally important that you explicitly issue a *commit* (with a Transaction Action) at the end of the JDBC component action model, after all database operations have completed, if your transactionable logic executed without error.
- It restores the state of the autocommit flag at the end of the transaction immediately before returning the connection back to the pool

If you check the Allow SQL Transactions box, Novell recommends that you deploy your Composer service either as a conventional servlet-triggered service, or as an EJB in the Container-managed, "Not Supported" transaction mode. In addition, we strongly recommend that you issue a *commit* or a *rollback* as the last SQL statement in your JDBC component. A "best practice" would be to wrap the entire JDBC component action model in a Try/On Fault block to catch any exceptions.

NOTE: As database drivers may react differently, be sure to test your application in a deployed state to verify the desired transactional behavior.

References

EJB home page: http://java.sun.com/products/ejb

JTA home page: http://java.sun.com/products/jta

A exteNd Application Server Dependencies

Connections

Using Novell exteNd Connection Pools

When specifying the connection pool name in the exteNd JDBC connection panel, make sure that it is specified using the Novell exteNd naming convention. Any database that has been added to the server is available for use with a connection pool. The naming convention for a database pool is **Databases/appDBName/DataSource** where **appDBName** is the name of the exteNd Application Server database that will be used for connection pooling.

For example, if a exteNd Application Server had a database attached called **ProductionDB,** the correct qualified name for the pool would be

Databases/ProductionDB/DataSource

B Contents of Deployment Objects

If you look in your staging directory after deploying, you will see a number of files. This appendix describes those files.

Deployment EAR

This is the final packaging of your project into a deployable object: It is what's deployed, ultimately, to the app server. This file, like WAR and JAR archives, can be opened with any .zip-file viewer. If you open it, you will see a project JAR, a WAR file, and optionally an EJB JAR and application.xml file.

Project JAR

Deploying a project results in the creation of a JAR file (the "Project JAR") that contains all the xObjects (as well as other XML files, such as schemas and WSDL) used in your project's deployed services. The xObjects are encoded as metadata in individual XML files (one per xObject). The xObject XML files have a context associated with them in the JAR. The context follows a naming convention that consists of a two-part path prefixed to the name of the xObject file.

The first path part, which you create, is a unique name called the *deployment context.* This can be any name of your choosing. (You specify this value in the first panel of the Composer deployment wizard.) The deployment context is used to distinguish two Composer services from each other that are named the same in different Composer projects residing in the same application server database. (In other words, the deployment context provides namespace separation.)

The second path part, which exteNd creates automatically, mirrors the same directory structure as the original Composer project on the hard disk. The directory structure for a Composer project consists of a root directory whose name is the name for the project, with subdirectories for each xObject type created (i.e., JDBC, Map, Connections, Functions, Script, Service, Code Tables etc.). Consider a Composer project called **Tutorial**, with a JDBC component named **LookupInventory**. The disk directory / file structure would contain the following:

{parent directory of project}\Tutorial\JDBC\LookupInventory.XML.

The final part of the path is the name of the xObject.

Example:

com.yourcompany.project.jdbc.LookupInventory

Where:

- **com.yourcompany.project** is the deployment context
- **jdbc** is the object type (and directory name)
- **LookupInventory** is the xObject

WAR

The WAR file inside your deployment EAR contains a manifest as well as a web.xml file. The manifest file tells the app server about the JARs in your project. The web.xml file contains servlet/URL/classfile associations and related information, so that at runtime the app server knows how to invoke the trigger servlets that (in turn) invoke your services.

Servlets

For each Servlet that the Deployment Wizard generates, an entry is made in the **web.xml** file of the WAR file. The WAR file, in turn, is stored inside the deployment EAR.

EJBs

For each EJB that the Deployment Wizard generates, an entry is created in the manifest file for the EJB deployment JAR, called **meta-inf/ejb-jar.xml**, which contains the type of EJB (i.e., session or entity), environment settings for each EJB, and the classes that make up the EJB. An entry is also made in the **BuildEJBs.XML** descriptor file that specifies the EJBs to build and their JNDI names.

The EJB Jar is named using the JAR filename that was specified in the Deployment Wizard. The name of the deployment EJB JAR file and the remote interface EJB JAR, both of which are built on the Novell exteNd Application Server during the deployment, are also based upon the project JAR filename. The naming conventions for the three JAR filenames are:

- EJB JAR – **EJB-xxxx**
- EJB deployment JAR – **EJBDeployxxxx**
- EJB Remote interface JAR – **EJBStub-xxxx**

(where **xxxx** is the name of the Project JAR file)

In the following example the Project JAR filename is **Production.jar**

- EJB JAR: **EJB-Production.jar**
- EJB Deployment/built JAR: **EJBDeployProduction.jar**
- EJB Remote interface JAR: **EJBStub-Production.jar**

ImportObjects.bat

This batch utility contains all of the SilverCmd calls to import and deploy the various artifacts (deployment files) that were created by the Deployment Wizard. See the Novell exteNd Application Server documentation for detailed information about this utility.

C Reserved Words

Avoid using Java-language keywords in your deployment-context strings. The following table lists Java keywords.

Java Keywords

abstract	boolean	break
byte	case	catch
char	class	const
continue	default	do
double	else	extends
final	finally	float
for	goto	if
implements	import	instanceof
int	interface	long
native	new	package
private	protected	public
return	short	static
strictfp	super	switch
synchronized	this	throw
throws	transient	try
void	volatile	while

D Server Glossary

Bean Managed Transactions

An Enterprise Java Bean that demarcates its own transaction boundaries is said to exercise *bean-managed* transaction control. (The alternative is Container-managed transactions.) The bean-managed model allows the programmer to exert low-level control over transaction logic, but at the expense of extra code and program complexity.

Connection Pool

A group of database connections that can be shared among processes, under the control of a management process (typically the application server). Since opening and closing database connections can becostly from a performance standpoint, it makes sense for a server to cache connections.

Container-Managed Transactions

Also known as declarative transaction control, the Container-managed transaction model shifts transaction management responsibilities out of the EJB and into its Container. EJBs that use this transaction model need not be “transaction aware” at the internal code level. Instead, the bean’s transaction attributes can be set in a descriptor, and the Container will ensure that appropriate control is exercised over transactions in which the bean may play a part. The Container-managed model can greatly reduce code complexity while increasing reliability.

Deployment Context

The deployment context is a name string (whose elements are separated by periods) that can be used to prevent namespace collisions between services with like-named components.

JNDI

Java Naming and Directory Interface. A standard extension to the Java platform, providing a unified interface to multiple naming and directory schemes that might exist across file systems and server domains.

JTA

Java Transactions API. A standard Java interface between the transaction manager and parties involved in a distributed transaction system. Bean-managed transactions rely on this API.

Params (URL/Form)

One of the four canonical Composer service trigger types. This Servlet type builds an in-memory XML document using HTTP URI form parameters as the names of nodes and their values as text. Multiple values for a parameter can be handled, but multiple input documents are not created.

Service Triggers

A Service Trigger is a Java Servlet or Enterprise Java Bean created when deploying a project from Composer. It submits a Service to exteNd.Server for execution. A Service Trigger is also associated with an URI and converts inbound data into XML documents as input to the service it triggers.

SOAP (Simple Object Access Protocol)

A platform-independent protocol for remote invocation of objects using HTTP as the transport layer and XML to represent the payload.

XML (HTML form field)

One of the four canonical Composer service trigger types. This Servlet type extracts a service's input document from a POSTed form's field. The Servlet expects the field name containing the XML file to be called 'xmlfile' and it uses the first occurrence of this parameter for the extraction.

XML (HTTP POST)

One of the four canonical Composer service trigger types. This type of trigger Servlet extracts an XML document sent via an HTTP POST method. This differs from HTML Form POSTs that contain *parameter name* | *value* pairs. The payload of this kind of HTTP transmission is, in fact, the raw XML document. It is a convenient method for exchanging XML documents with trading partners.

XML Metadata

All exteNd objects created in Composer are themselves stored as XML files. The object data and processing instructions in these files are referred to as XML metadata. The exteNd runtime engine processes this metadata to perform XML Integration services.

XML (MIME multipart)

Another of the four canonical Composer service trigger types. This Servlet type extracts a service's input document from a multipart encoded form containing a field with an input type of file. The Servlet expects the field name containing the XML file to be called 'xmlfile' and it uses the first occurrence of this parameter for the extraction.

Index

S

T

U

X