Novell exteNd Composer

5.2

www.novell.com

USER'S GUIDE

Novell®

Legal Notices

Novell, Inc.
404 Wyman Street, Suite 500
Waltham, MA 02451
U.S.A.

www.novell.com

exteNd Composer *User's Guide*
June 2004

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

Novell Trademarks

ConsoleOne is a registered trademark of Novell, Inc.
eDirectory is a trademark of Novell, Inc.
GroupWise is a registered 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.
iChain is a registered 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.
Novell eGuide is a 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

The Apache Software License, Version 1.1

Copyright (c) 2000 The Apache Software Foundation. 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 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 "Apache" 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.

JDOM.JAR

Copyright (C) 2000-2002 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).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following: "This product includes software developed by the JDOM Project (http://www.jdom.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.jdom.org/images/logos.

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 JDOM AUTHORS OR THE PROJECT 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.

Sun

Sun Microsystems, Inc. Sun, Sun Microsystems, the Sun Logo Sun, the Sun logo, Sun Microsystems, JavaBeans, Enterprise JavaBeans, JavaServer

Pages, Java Naming and Directory Interface, JDK, JDBC, Java, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, SunWorkShop, XView, Java WorkShop, the Java Coffee Cup logo, Visual Java, and NetBeans are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Indiana University Extreme! Lab Software License

Version 1.1.1

Copyright (c) 2002 Extreme! Lab, Indiana University. 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 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 Indiana University Extreme! Lab (http://www.extreme.indiana.edu/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "Indiana University" and "Indiana University Extreme! Lab" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact http://www.extreme.indiana.edu/. 5. Products derived from this software may not use "Indiana University" name nor may "Indiana University" appear in their name, without prior written permission of the Indiana University.

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 AUTHORS, COPYRIGHT HOLDERS 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.

Phaos

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

W3C

W3C® SOFTWARE NOTICE AND LICENSE

This work (and included software, documentation such as READMEs, or other related items) is being provided by the copyright holders under the following license. By obtaining, using and/or copying this work, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions.

Permission to copy, modify, and distribute this software and its documentation, with or without modification, for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the software and documentation or portions thereof, including modifications: 1.The full text of this NOTICE in a location viewable to users of the redistributed or derivative work. 2.Any pre-existing intellectual property disclaimers, notices, or terms and conditions. If none exist, the W3C Software Short Notice should be included (hypertext is preferred, text is permitted) within the body of any redistributed or derivative code. 3. Notice of any changes or modifications to the files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permission. Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

Contents

About This Book

Purpose

This guide describes how to use Novell exteNd Composer, a visual design environment for creating business-to-business integration applications, including Web Services. This documentation provides information on the use of Composer's design-time features. Runtime functionality is more thoroughly described in the *Composer Enterprise Server Guide*.

Audience

This guide is aimed at application designers who will be building J2EE applications (including Web Services) using exteNd Composer.

Prerequisites

You should be familiar with XML-related standards (including Schema, XSL, and XPath), the Document Object Model, and basic J2EE concepts involving file packaging (JAR/EAR/WAR files). Some knowledge of ECMAScript is also helpful, though not required, for using the product. If you are building Web Services, you should be familiar with WSDL, SOAP, and related standards.

Additional documentation

For the complete set of Novell exteNd exteNd Director documentation, see the Novell Documentation Web Site (**http://www.novell.com/documentation/**).

Organization

This guide is organized as follows:

Chapter	Description
Chapter 1, *Welcome to exteNd Composer*	Gives an overview of exteNd Composer, its capabilities, and design philosophy.
Chapter 2, *Planning your Application*	Describes the necessary preparations for designing and building an XML Integration Application.
Chapter 3, *Getting Started in exteNd Composer*	Describes launching the product and the elements of the Composer environment.
Chapter 4, *Creating and Managing your Projects*	Describes projects and their elements and explains how to create them.
Chapter 5, *XML Templates*	Describes XML templates, sample documents, DTDs, XSL stylesheets, and XML categories and when and how to use them.
Chapter 6, *Creating an XML Map Component*	Describes XML Map Components, using XML sample documents to build components, DOMs, and how and why DOMs are used.

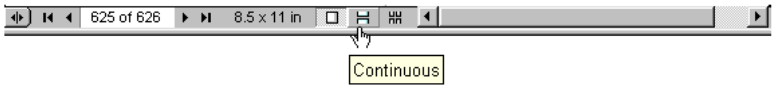
Chapter	Description
Chapter 7, *Basic Actions*	Describes the core actions that are available in all of Composer's component editors, including map actions, log actions, various flow-control actions, and so on. Step-by-step procedures are given for how to create each action.
Chapter 8, *Advanced Actions*	Describes advanced actions including Declare Group, Repeat for Group, Process XSL, Repeat While, Throw Fault, Try/On Fault, and the various data exchange actions.
Chapter 9, *Resources*	Describes the various types of Composer resources, including schema resources, WSDL, code maps, code map tables, connections, and custom scripts.
Chapter 10, *Custom Scripting and XPath Logic in exteNd Composer*	Describes custom scripting using Composer's built-in ECMAScript facility. Also, a discussion of how scripting can be used in conjunction with XPath, DOMs, and Java. An API guide to Composer's built-in ECMAScript extensions is presented, as well.
Chapter 11, *Applying Actions to Common Tasks*	Describes element and data mapping, leaf element mapping, deep copy mapping, transforming elements with code tables and functions, performing loop actions, and performing aggregate calculations.
Chapter 12, *Testing and Debugging*	Describes the animation tools and how to use them to test services and components.
Chapter 13, *Working with Services*	Describes how and what services are, how to build them or import them, what the service editor is, and how to build a service with components. This chapter also contains an end-to-end example of how to call an external Web Service based on information contained in a WSDL resource.
Chapter 14, *Working with Registries*	Tells how to use the features associated with Composer's "Registries" tab, including how to search UDDI registries, publish to UDDI registries, retrieve WSDL from registries, etc.
Chapter 15, *Deploying Your Project*	Explains basic issues relating to the deployment of Composer services to an app server.
Appendix A, *The Composer JSP Tag Library*	Describes the Composer tag library.
Appendix B, *Reserved Words*	Lists reserved words, which should not occur in user-defined variable names or labels.
Appendix C, *Glossary*	Definitions of key terms used in this Guide.

About the PDF Documentation

Various navigational features are available when viewing this document in Acrobat Reader:

- The Bookmarks frame (left side of window) lists the contents of the document, by chapter name, heading, and subheading. Every topic listed in the content tree is a clickable link. To flip open the entire subtree under any tree node, Control-click on the parent node. To toggle the visibility of the Bookmarks frame, press F5.
- Every item in the book's Index is a clickable link that will take you directly to the text discussion.
- Wherever a website address (URI) appears, you will usually find that clicking on it will take you to the site in your browser. Even if the URI is not in blue or underlined, it will generally be a hot link. You can test this by hovering the mouse over the URI. The cursor will change from an arrow to a finger cursor if the link is hot.
- Cross-references within and between chapters are also clickable.

- Use Control-N to navigate to a given page in the document. A dialog will prompt you for the page number.
- You can Copy PDF text to the clipboard in the normal way (by shift-dragging to select text, then using Control-C). Many programs will allow you to Paste (or “Paste Special”) clipboard contents as RTF (rich text format), retaining certain formatting features. To select large portions of text spanning PDF pages, first click the “Continuous Pages Mode” icon in the button bar at the bottom of the Acrobat window: Then shift-drag to select text (or Control-A to Select All) and Copy.



1 Welcome to exteNd Composer

Web Services are fundamentally changing the way enterprises exchange information and perform business transactions. But to succeed in web services development, business analysts and developers must be able to work together on sophisticated, large, distributed applications that meet strict requirements for performance, security, scalability, and reliability, in the face of increasingly stringent time-to-market demands.

The most important factor in making a successful transition to a services-based architecture that leverages modern web technologies is the *availability of powerful, easy-to-use development tools*. Such tools should be:

- Purpose-built, from the ground up, for Web Services development
- Tightly integrated by design—not a grabbag of unrelated pieces
- *Easy to learn and use*, so that a diverse team of users—from business analysts to system administrators to software engineers—can be productive immediately in a concurrent-development setting
- 100% standards-aware in terms of all important Web Services technologies: XML grammars (including SOAP), description and discovery technologies (WSDL, WSIL, UDDI), transport layers (HTTP and others), directory protocols (LDAP, DSML), and security-related standards, among others.
- Compatible with diverse deployment and runtime environments: i.e., a variety of application servers on a variety of operating systems
- If Java is the programming language, the development environment should be fully J2EE-aware—not just a 3GL IDE (integrated development environment), but a front-to-back development, testing, packaging, and deployment toolset with full awareness of JAR/WAR/EAR issues, portal/portlet architectures, etc.

Novell's exteNd product line meets all of these criteria.

The Novell exteNd Family

Novell exteNd is a family of Web Service development products for rapid development of Web Service objects on J2EE (Java) application server platforms The major pieces are available independently or as an integrated suite. The suite, in turn, comes in two flavors: Professional Edition, and Enterprise Edition.

Novell exteNd 5 Professional Edition

The Novell exteNd 5 Professional Edition Suite is the base configuration of tools that enable the application developer to develop and deploy enterprise-level Web applications. The Professional Suite contains the following tightly integrated components:

- Novell exteNd Application Server (Sun-Certified J2EE application server)
- Novell exteNd Composer with JDBC Connect and LDAP Connect.

NOTE: The Professional Edition suite does not support direct-to-app-server deployment from the Composer GUI. (Deployment must be done from within exteNd Director.) It also does not support transaction (JTA) awareness, EJB-based service triggers, nor XForms awareness.

- Fully functional eval versions of the Composer Connects for 3270, 5250, CICS RPC, Data General, EDI, HP3000, HTML, JMS, SAP, Tandem, Telnet, and Unisys (UTS and T27).
- Novell exteNd Director (*excluding* the Content Management and Workflow subsystems and the Autonomy-based search functionality). Director is a full J2EE development environment with sophisticated packaging and deployment capabilities, as well as robust subsystems for portal and portlet application development.
- MySQL 4.1 database server
- Novell exteNd LDAP Utility directory server
- Web Services SDK (a JAX-RPC implementation that includes compilers and runtime environment for supporting SOAP-based Web Services)
- The Novell exteNd Messaging Platform—support for standards such as JMS (Java Messaging Service), the Common Object Request Broker Architecture (CORBA), Java Transaction Service (JTS), and the Java Transaction API (JTA).

Novell exteNd 5 Enterprise Edition

The Novell exteNd 5 Enterprise Edition Suite contains all of the same pieces as the Professional Suite, with a few additions:

- Novell exteNd Composer Enterprise Edition, *including* the Business Process Modeler (BPM) subsystem, plus support for direct-to-app-server deployment, EJB trigger options, transaction control, and XForms integration.
- Full versions of the following Composer connectivity add-ins:
  - JDBC Connect
  - LDAP Connect
  - JMS Connect
  - HTML Connect
  - Telnet Connect
- Support for the SAP Service type (triggering of Composer services via RFC requests at an SAP gateway).
- Novell exteNd Director Enterprise Edition, *including* the Content Management and Workflow subsystems as well as the Autonomy-based search functionality.
- All of the other pieces mentioned in the previous section (MySQL, WSSDK, Messaging Platform, and so on).

The Novell exteNd Composer Product Line

Novell exteNd Composer is a development (and runtime) environment designed for rapid design and deployment of Web Services and XML integration applications—applications that can connect to diverse back-end systems and data sources.

The Composer product consists of the following pieces:

- **Novell exteNd Composer –** A visual design-time tool for creating and debugging Web Services and XML-enabled back-end integration applications.
- **Novell exteNd Composer Enterprise Server –** The runtime container layer (for use on any compatible J2EE app server) that executes and manages applications created in exteNd Composer.

- **Novell exteNd Connect Family –** Individual add-in products that augment the capabilities of exteNd Composer and Server to permit the XML-enablement of systems that rely on specialized data sources, such as EDI, CICS RPC, 3270/5250 terminals data streams, Telnet, and JMS. (The exteNd Composer JDBC Connect, which allows communication with relational databases, is bundled into the core Composer installation suite, as is the LDAP Connect. Other Connect products are available separately.)

All exteNd Composer products are certified to run under the Novell exteNd Application Server, Apache Tomcat, IBM's WebSphere, and BEA WebLogic, with support for operating systems ranging from Windows NT and Windows 2000 to Linux, Solaris, AIX, and HP-UX.

NOTE: Novell exteNd Composer Enterprise Server and exteNd Composer Enterprise Connect products each have their own documentation. This Guide covers only the core development environment (which we refer to herein as Composer). See the separate JDBC Connect and LDAP Connect guides for information on those component editors.

What Is Composer?

Composer offers a powerful, intuitive, point-and-click GUI (graphical user interface) for rapid application development, giving the business analyst or application developer a powerful tool for creating robust XML integration applications in minimum time.

Composer offers, among other features:

- An XML editor with code-completion features for WSDL, WSIL, and other "specialty grammars"
- A drag-and-drop-enabled XML mapping engine, with support for schemas, DTDs, XSL, XPath, and DOM Level 2
- An intuitive, visual editing environment for implementing standard control-flow constructs, error trapping, logging, etc., without the need for extensive Java programming expertise
- Realtime step-into/step-over debugging and animation, so that applications can be tested in real time without leaving the development environment
- Support for "watch variables" at debug time
- Support for To-Do lists
- A multi-document interface (MDI), allowing you to work in more than one document or component at one time
- Realtime registry browsing, with support for WSDL publishing/retrieval using UDDI registries
- Autogeneration of WSDL with SOAP bindings
- Built-in ECMAScript support (including a custom script editor and live console) for users who need fine control over business logic or data manipulation
- Back-end system connectivity via exteNd Composer Connect add-ins for 3270, 5250, Telnet, JMS, JDBC, CICS RPC, EDI, etc.
- Deployment facilities for direct deployment of projects to the app server (with context-driven customizations for Novell, WebSphere, or WebLogic app servers)
- Integration with exteNd Director (switch to the Director environment at any time and import Composer projects into any WAR file or EAR project)

Composer also has XForm and JSP code generation features, and many other capabilities that aren't listed here for space reasons. This is just a partial list of major features.

Who Can Use Composer?

Composer is targeted at business analysts, IT managers, Java developers, and other stakeholders in the Web Services development process.

Composer is designed to be accessible to users of various skill levels. (It is *not* a Java-programming IDE.) Business analysts with little or no programming background can quickly master complex data transformations using Composer's drag-and-drop XML mapping features. Webmasters can use Composer's JSP, XForms, and UDDI browse/publish capabilities to assemble sophisticated web apps with no need for additional tools. Java developers can use Composer to develop sophisticated, reusable XML components that might rely heavily on ECMAScript, SQL, LDAP, custom Java classes, and/or specialized packagings (WAR/EAR/JAR files).

Because Composer's GUI is rich with wizards, picklists, and drag-and-drop-enabled features, users of all skill levels (regardless of domain expertise) can become productive quickly. Sophisticated web applications that might take *months* to develop using a "grabbag of tools" approach often can be rolled out in weeks or days using Composer.

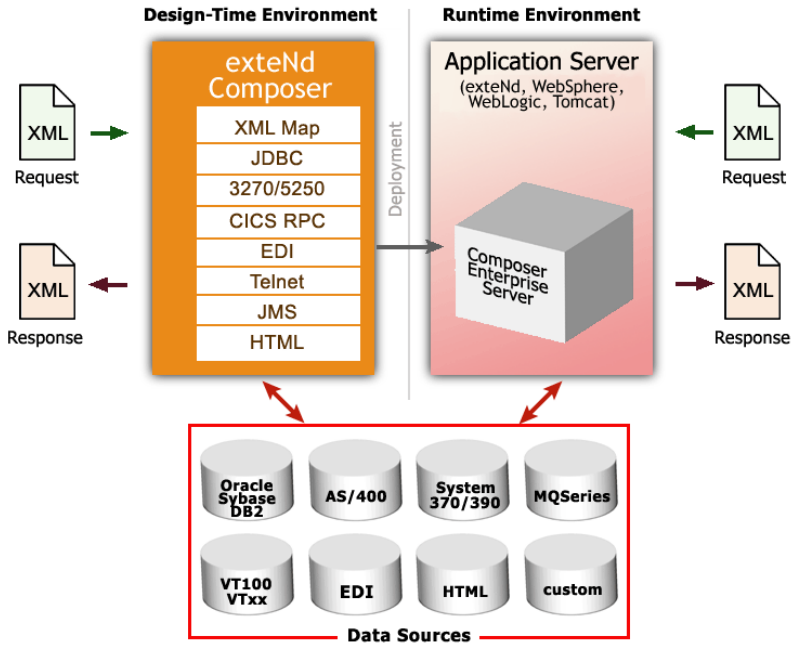
Components and Services

Composer application design is based on an *Action Model* architecture that includes two main processing constructs: *components* and *services.*

Components are executable units of work that encapsulate business logic, usually in the context of connectivity requirements.

For example, a typical JDBC component validates an incoming XML request document, maps the document's key pieces of data to an SQL inquiry, and maps the SQL result set to an XML response document. All of the business logic and data retrieval functionality of this type of operation can (and should) be encapsulated at the component level.

Services, on the other hand, typically oversee the execution of components and coordinate the flow of data between them. A typical service might wrapper a series of components that receive an input XML document, perform sophisticated document mappings/transformations, collect information from back-end data sources, execute transactions on mainframes and AS/400s, process error conditions, send context-sensitive e-mail or JMS notifications, and/or return one or more XML response documents to the original requestor(s). By breaking up a service's tasks into discrete components, important benefits—in terms of testing, debugging, code maintenance, encapsulation, and code reuse—can be realized.



You will typically use Composer to create components and services that perform B2B integration tasks involving data retrieval and transformation through XML technologies, including (optionally) SOAP and Web Services technologies. You'll deploy these components and services into a J2EE application server environment, where their execution is mediated by exteNd Composer Enterprise Server (the runtime half of Composer).

What Kinds of Applications Can You Build with Composer?

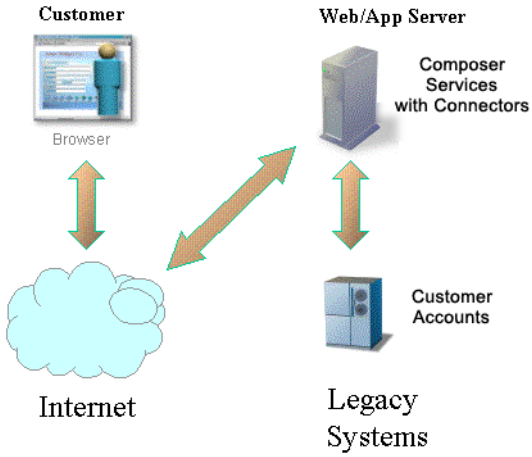
You can build many types of applications with Composer, but typically you will create XML integration applications triggered by servlets, EJBs, custom Java objects, or incoming messages on a JMS message queue. Your applications might, in some cases, simply be used locally on the app server to provide services across local processes, with no exposure to the outside world. In other cases, your applications will be fully web-enabled. The interface(s) to your web-facing applications might or might not involve SOAP or WSDL.

In general, with Composer, you can implement any kind of application where data inputs and outputs involve XML.

If you are using the Enterprise Edition of Composer (which comes with the JMS Connect), you can also build services that use *messages* for inputs and outputs.

NOTE: Messaging (involving Message Oriented Middleware, such as IBM's MQSeries) is a powerful data-sharing metaphor in its own right, allowing the use of payloads other than XML. With Composer and JMS Connect, you can build applications that use messages for input and XML for output; XML on the input side and messages for output; or messaging within an XML-in/XML-out application; plus other variations.

In the simple example depicted below, a buyer and a supplier connect their respective business systems across the Internet using XML and Composer.



Your organization might want to build one or more of the following types of applications using Composer:

- **Internal Application Integration Services**. You may have many applications between which you want to exchange data from diverse sources. For example, you may want to connect an Oracle financial application, an SAP manufacturing application, and an in-house-developed order processing application together. Composer will help you achieve this.
- **External Web Service Applications**. You may have a need to expose a service to trading partners (or other users) via the Web. SOAP services and WSDL-based Web Services can be constructed quickly and easily using Composer. Once you've designed a service in Composer, Composer will actually autogenerate WSDL for the service (and even publish it to a UDDI registry, if you want).
- **Data Warehousing Applications**. Composer works well with data mining and warehousing technologies, since Composer's key function is to map data from disparate sources.

Automated Business Process Management (Workflow)

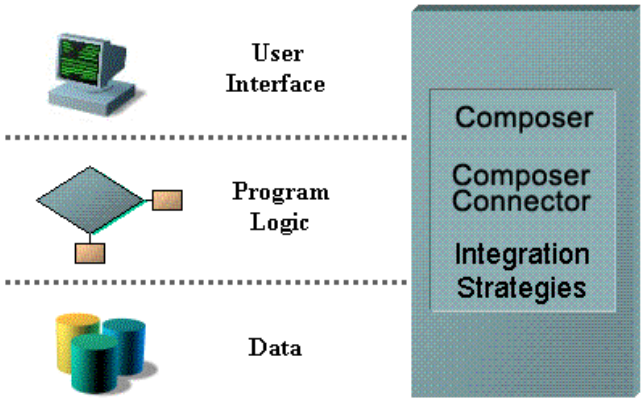
Packaging business applications as Web Services opens up new opportunities for automating workflows. The Web Services Flow Language (one of several emerging standards for workflow automation) provides a standard to which next-generation BPM software will build. The basis for this next-generation technology is workflow built on *Web Services*. SOAP and XML will be key technological underpinnings of future workflow systems.

Composer plays directly to emerging standards and technologies involving automated workflow. Next-generation workflow engines will "hook up" Web Services (external or internal) to allow sophisticated long-running applications to be built, relying on BPM concepts such as timeouts/retries, conditional links between services, control flow between individual services involving parallel execution, etc. Complex choreographies involving Web Services will be attainable. (Some of the possible choreographies are described in RosettaNet Partner Interface Processes.) Composer will be a valuable tool in creating WSFL-ready applications.

About the Composer Enterprise Connect Product Line

Composer is built upon a simple hub and spoke architecture. The hub is a robust XML transformation engine that accepts XML documents, processes the documents, and returns an XML document. The spokes, or Connect products, are plug-in modules that "XML enable" sources of data that are not natively XML-aware, bringing their data into the hub for processing as XML. These data sources can be anything from legacy COBOL / VSAM managed information to Message Queues to HTML pages.

The various Connect products can be categorized by the integration strategy each one employs to XML enable an information source. The integration strategies are a reflection of the major divisions used in modern systems designs for Internet-based computing architectures. Depending on your B2Bi needs, exteNd can integrate your business systems at the User Interface, Program Logic, or Data levels.



In addition to JDBC and LDAP (which are core Connects, included with all versions of Composer), there are additional Connect products:

- **JMS**—Java-based messaging using the Java Message Service standard. This Connect product provides connectivity between Composer applications and any JMS-aware messaging system.
- **3270 and 5250**—Seamless connectivity with two of the most common terminal data stream types.
- **CICS RPC**—Transparent ability to interact with COBOL systems via remote procedure calls through CICS.
- **Data General**—Connect with Data General hosts using DG emulation.
- **EDI**—Create XML integration applications that are EDI-aware.
- **HP3000**—Connect with HP3000 systems.
- **HTML**—Screen-scrape web pages and/or remap HTML data to XML data.
- **Tandem**—Connect with Tandem-based systems.
- **Telnet**—Screen-scrape and interact with any Telnet data stream.
- **SAP**—Launch Composer-built services when a BAPI-enabled SAP function launches.
- **T27**—Connect with Unisys T27 systems.
- **UTS**—Connect with Unisys UTS systems.

NOTE: This guide describes the basic functionality of exteNd Composer. The addition of each Connect increases the features available to you in Composer. These additional features are described in separate user guides that accompany each Connect.

Once you install a Connect product, the Composer GUI will become updated with:

- New xObject categories corresponding to the Connection Resource types and Component types specific to the product in question
- Specialized component editors and realtime interactive Native Environment Panels (emulation screens) appropriate to the target system
- New action types
- New menu choices and associated dialogs and wizards

These customizations and additions are automatic with the installation of the Connect and integrate seamlessly into the existing Composer design-time environment.

Updating Your License(s)

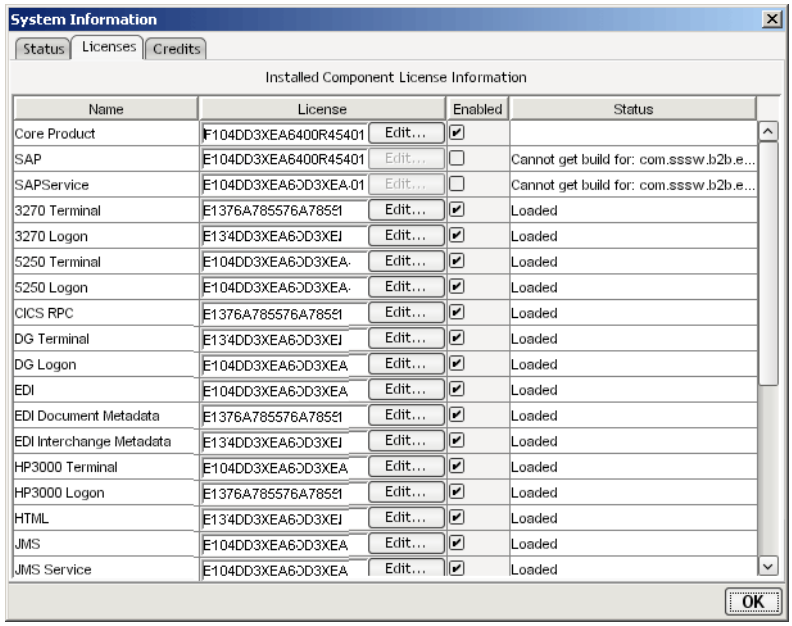
Should the need arise to update the license string(s) associated with Composer or a Connect product, you can do so at any time, using an intuitive point-and-click UI.

IMPORTANT: When changing a license string, remember that it is necessary to change the string for the *design-time* environment as well as the *server* environment. Both procedures are shown below.

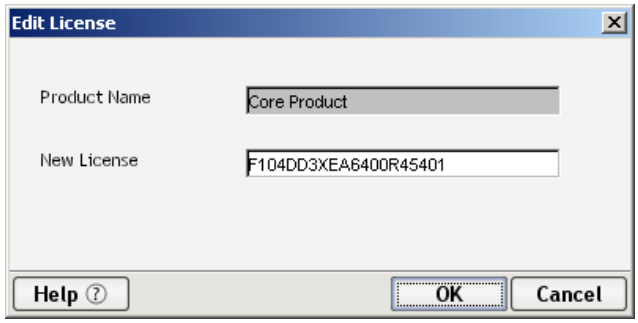
Updating Design-Time License String(s)

➢ **To update a Composer product license string on the design-time machine:**

1. Launch Composer.
2. Under the **Help** menu, select **About Composer**. A dialog appears.
3. At the bottom of the "About" dialog, click the **System** button. A new dialog appears:



4. At the top of this dialog, click the **Licenses** tab.
5. The columns of the table shown on this tab give useful information about the name and status of each Composer product, including those that for some reason didn't load properly. To edit a license string, click the **Edit...** button next to the appropriate string under the License column. A new dialog will appear.



6. Enter a new string in the **New License** text field.
7. Click **OK** to dismiss the dialog.

NOTE: If the string you enter is not correct, you will get an alert dialog at this point. Doublecheck the string and reenter it. If problems persist, Cancel out of all dialogs to return to Composer, then contact Customer Support.

8 In the System Information dialog, check the **Enabled** checkbox next to the field you edited, if it is not already checked.

9 Click **OK** (or use the Enter key on your keyboard) twice to return to Composer.

10 Restart Composer to make your changes take effect.

11 If you have not already updated the same license string on the app server, continue now to the following procedure.

Updating Runtime License String(s)

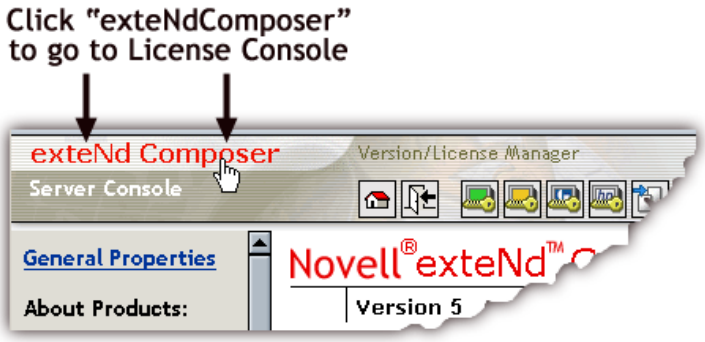
When changing license strings in the design-time environment, it is critical that you make corresponding changes in the app-server environment so that Composer Enterprise Server will treat the corresponding product(s) as enabled at runtime.

➢ **To update a Composer product license string on the app server:**

1 Launch the app server, if it is not already running. This should also launch Composer Enterprise Server, if it was previously installed.

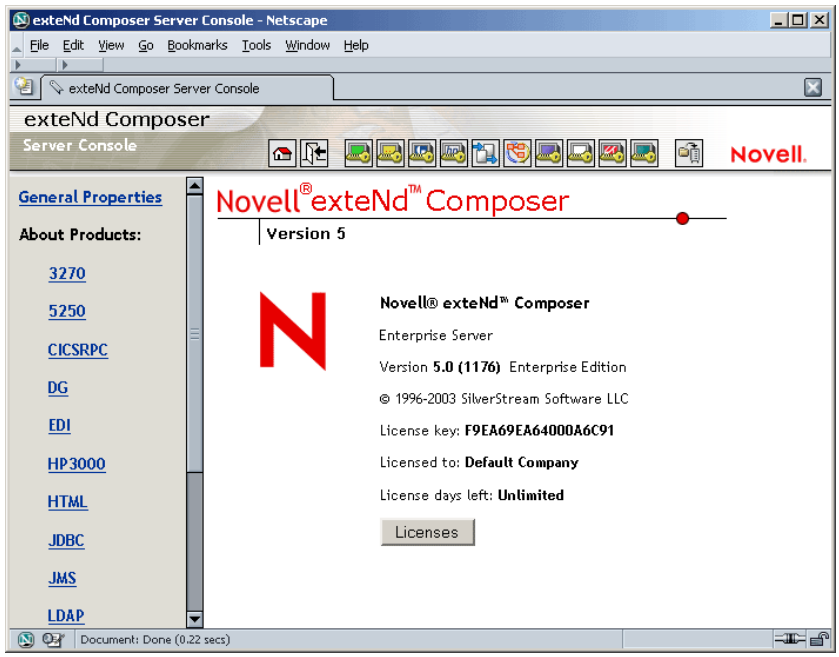
2 Go to Composer's default Administrative Console page, which is typically at **http://localhost/exteNdComposer/Console**.

3 In the upper left corner of the main console page, hover the mouse over the words "exteNd Composer." (See illustration below.)

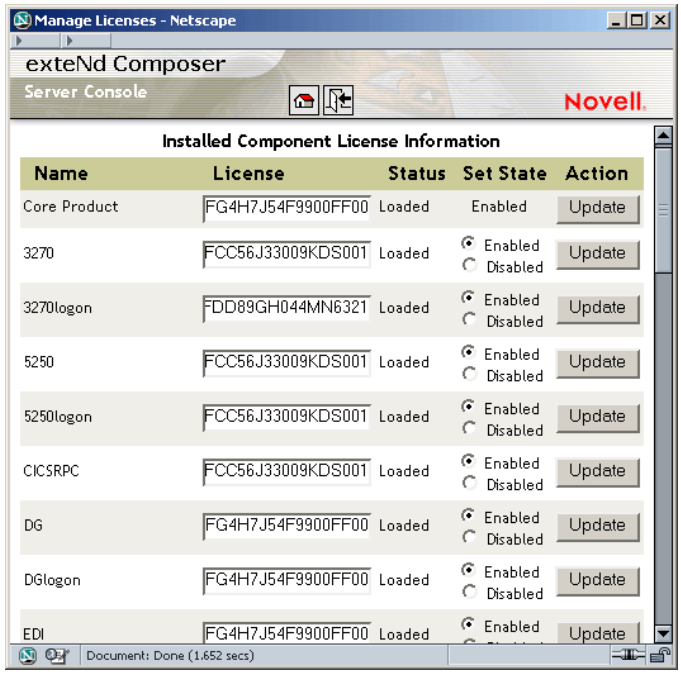


When you hover the mouse over the words "exteNd Composer," the words change color (to red) and new text, "Version/License Manager," appears off to the side.

4 **Click** once on “exteNd Composer.” A new page appears:



5 This is the main License Manager screen. In the center of the screen, you will see version and license information for Composer Enterprise Server. To see additional information (and edit license strings), click the **Licenses** button. A new window appears:



6 This page shows name and status information for all installed components. Each text field in the **License** column is editable. Enter a new string as appropriate, then ensure that the correct **Enabled/Disabled** radio button is active next to the text field in question.

7 Click the **Update** button next to the field in question.

8 Restart the server.

Where To Go for More Help

Perhaps the best way to understand Composer is to see it in action. The Composer installation includes a fully functional project, in the Tutorial Solution, that you can step through to see how the application handles a practical business operation. See the **tutorial** folder, under the main exteNd Composer installation folder.

For the most up-to-date documentation and tutorials (plus other resources), be sure to consult **http://developer.novell.com/extend/composer/**.

2 Planning Your Application

Novell exteNd Composer allows you to build robust XML integration applications that can be deployed as Web Services. The applications you build with Composer can tie together diverse back-end systems, effectively XML-enabling data sources that are heterogeneous with respect to communications protocol, file formats, and/or operating systems.

A Composer service can include components that map and transform XML content, as well as other operations (such as sending email), while carrying out any kind of business logic that can be handled with Java or J2EE technologies.

The number of different types of back-end systems you can reach with Composer applications depends on how many Composer Connect products you have installed. Composer's core installation includes the JDBC Connect for reaching into database systems. Other Connect products allow you to exchange data with 3270/5250 systems, take advantage of CICS RPC operations, use JMS messaging, establish Telnet sessions, etc. You can also use EDI data and/or execute SAP functions.

Composer offers an intuitive visual interface for creating integration applications and testing them at design time via a powerful, interactive "step-through" debugging facility. Using simple drag-and-drop operations, you can build extremely sophisticated XML integration applications in minutes, without writing a single line of Java code. When you're done, your application can be deployed quickly to a J2EE application server (whether Novell's exteNd app server, or another J2EE-compliant server).

How Do I Design and Build an Application in Composer?

Your approach to using Composer begins the same way you begin any project: by capturing the requirements and by understanding the building blocks available to you to meet your requirements.

The building blocks that you'll use in Composer are:

- Services
- Components
- Resources
- XML Templates

You can think of Components as implementing the smaller units of work that will be collected into a Service. Resources are things like XML schemas, custom script libraries, and connection profiles that one or more components might need at execution time in order to do their work. Templates are typically XML stub documents needed by components and services.

Care should be taken when designing and building your components so that you can achieve the greatest amount of reuse. For example, you can create a component that uses a common XML document to access information from a legacy data source and call that component for each request. The component can be designed to preprocess incoming requests to particular format needs so that other components won't have to do the same thing on a component-by-component basis.

What is an xObject?

You'll often see the word "xObject" used throughout this Guide. An xObject is nothing more than a metadata definition of a Service, Component, Resource, or Template created by Composer. All of the data and instructions used in a Service, Component, or Resource are persisted to disk in XML form. Composer creates the corresponding runtime object(s) via the persisted metadata. The object that gets created is an xObject.

You won't need to worry about the low-level internals of xObjects *per se*. Composer handles that for you. From a terminology standpoint, you can think of xObjects as the XML-storable objects that make up a Composer application: namely, Components, Services, Resources, and Templates.

NOTE: If you're curious to see what an xObject looks like inside, open any of the XML files under your project's **\composer** directory structure, using your favorite XML editor. For example, to inspect a Connection Resource xObject, open any file under **\composer\connection**.

What is a Service?

A Composer *service* is an xObject that calls one or more *components* designed to perform a logical unit of work. A service accepts one XML document as input, uses components to operate on the XML data, and then produces one output XML document. Services map, transform, or transfer data between data sources on an XML document level. Services are the runtime deployable units that integrate into an enterprise scalable application server environment (For additional information on deployment strategies, see the *exteNd Server Guide*). A service can execute other services or components. Examples of services that you can build include:

- Sending status information to a trading partner based on an XML request
- Retrieving data from legacy data sources in response to a Web browser request
- Exchanging information between internal data sources

What is a Component?

A Composer *component* is a set of instructions or actions for processing XML document elements and/or communicating with non-XML data sources. Components accept one or more XML documents as input, performs activities on an element level, and then produce one or more XML documents. You can build simple or complex components of different types and link them together to carry out complete business operations. They map, transform, or transfer data between XML documents on an XML element level. They can also move data between XML documents and external data sources such as live 3270 transactions and SQL databases. Components can execute other components or services.

Components should be designed to perform discrete processes so that these common processes can be shared between services. Examples of components include:

- Mapping an input request to a common standard
- Accessing a relational database based on the common standard
- Transforming XML documents from one standard to another

What is a Resource?

As you will see later, components and services contain Action Models that execute the mapping, transformation, and transfer of data within XML documents. However, there are instances when the operations required are more specialized and complex than the Action Model's capabilities. This is where resources are used. Resources do not contain Action Models, nor do they contain input or output XML documents. Resources work like utilities to help components and services carry out their tasks.

Composer's resources include:

- **Code Tables**—Code Tables store commonly used business code tables (e.g., State and Region tables)
- **Code Table Maps**—Code Table Maps transform one set of codes from a Code Table into another set of codes (e.g., State to Region mapping)
- **Connections**—Connections establish communications with specific sources of data in Connect transaction environments (e.g., JDBC connections).
- **Custom Scripts**—Custom Scripts represent a library of user-developed functions using ECMAScript or Java language (e.g., String manipulations, accessing Java Business Objects)

What Is an XML Template?

An XML template contains the sample documents, definitions, and stylesheets that assist you in designing and testing the inputs and outputs to a component. In Composer, you use XML templates as the inputs and outputs for the components you build. It is important to note that XML templates are only used during design time; exteNd Server uses live documents during the actual execution of a service.

Basic Steps for Developing a Composer Service

Your application development process should take into account the following basic steps.

- Plan your service(s) before using Composer and gather the sample XML documents, definitions, and stylesheets you need
- Build and test the service(s) in Composer
- Deploy your service(s) to the server

Part One: Plan the Service (Before Using Composer)

Your Composer application is based on the processing of XML documents. In planning your application you will want to write and analyze the requirements before designing the services. You will:

- Determine input/output requirements. Where is the data coming from and where is it going to and in what format(s)?
- Collect any existing XML documents including, if available, any standard XML documents from industry groups and business partners
- Create input and output XML documents if required

Write the Requirements

In writing your requirements, the following questions will be useful to answer:

What does the input document look like? Does it conform to an industry standard? Do I need to define my own?

What does the output document need to look like? Where can I get samples? Are DTDs or XSL stylesheets required?

What processing components are required? Will the application need an XML Map component to transform XML data? Will the application need a JDBC component to connect to one or more databases? Can I reuse any components or resources that have already been built?

What additional resources does the application need? Are customized functions required?

Analyze the Requirements

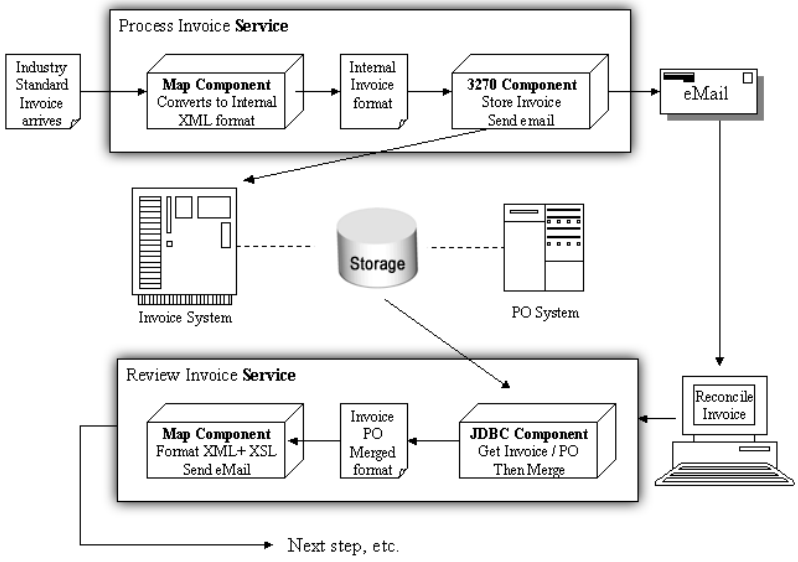
There are many aspects of your project that you must consider when in the design stages.

You'll need to know the data sources to which you need to connect. You must know what data you'll need, where it will come from, and what the transmission mode will be. Among the other details you to consider are:

- **Authentication**—Does the data source you plan on connecting to require authentication information, such as user IDs and passwords? Will you need authorization from an IT group? Will you need to coordinate with other departments?
- **Security**—Are there security issues? Firewalls?
- **Personnel**—Will you need special help connecting to data sources? Is there someone in your organization with the skills necessary to understand, help create, and troubleshoot or debug the necessary external data source connections?
- **Legacy Applications**—Will you need to contend with terminal data streams? Relational databases? Message queues?
- **Availability**—Are the data sources with which you want to connect going to be available whenever you want to connect with them? Can you connect as often as you wish?
- **Transaction Control**—Will your application need to incorporate rollback/commit logic?
- **XML Documents**—Are there existing XML documents or schemas you need to obtain from industry groups, standards organizations, or business partners?
- **Logging and Notifications**—Does your application have special progress-tracking or error-monitoring needs? Do notifications (via e-mail or JMS messaging) need to be sent when certain conditions arise? Does your service need to adhere to well-defined escalation procedures in case of problems involving credit limits, dollar amounts, supply chain difficulties, etc.?
- **Trading-Partner Requirements**—Is your service going to be used by trading partners? Do they have their own security, audit-trail, timeout/retry, and/or other requirements that may put constraints on your application's design?

Design the Service

Once you've analyzed the requirements, it's time to design the service. Your design may now begin to take into account Composer's building blocks, as the illustration shows.



As described earlier, a service is comprised of *components* and is the unit of deployment in exteNd Server. You should have a good idea at this point how many components you will need to build as part of your service. For example, if you need to map data from one XML document to another and perform a code table conversion, you will need a component to perform that task. If you need to make a connection to a JDBC database and extract data, you'll need a component to accomplish that work too.

Part Two: Build the Service

In building the service, you will:

- Create XML templates
- Create needed resources for the service, such as Schema and WSDL Resources, plus others
- Create executable building blocks (called Components) for the service, encapsulating the various stages of data retrieval and XML transformation unique to your service
- Create the service using the building blocks
- Test the service
- Document the service (if desired)

Part Three: Deploy the Service

You've created and tested your service. Now it's time to deploy it to an app server and put it into action. Composer Enterprise Edition includes a Deployment Object facility with an associated Deployment Wizard to step you through the process of packaging your design time service into a deployable archive and then deploying it. Also, exteNd Director contains facilities for packaging Composer runtime objects into archives as subprojects of a larger J2EE project.

NOTE: Basic (but essential) deployment considerations are discussed in Chapter 15 of this guide. A more detailed discussion of deployment-related issues and procedures will be found in the *Composer Enterprise Server User's Guide.*

How is Data Handled When a Service Executes?

Services and components pass information to one another during runtime processing by way of XML input and output messages and message parts. The messages and parts are parsed into DOM form (Document Object Model, used here to mean the in-memory object representation of a document). The XML output for one component or service is often the XML input for another component or service; however, services and components don't actually pass these XML documents as disk files, but rather pass "in memory" DOM images of the files. This is an important distinction, as these DOMs can be destroyed, changed, and recreated during processing to achieve your data integration goals without ever being written to disk or actually changing any disk files. Once an XML document is parsed and loaded into memory, it can be manipulated by the various mapping, transformation, and transfer features of the component editor and transaction environment each one accesses.

SOAP Messages

SOAP (Simple Object Access Protocol) is an industry-standard XML messaging methodology in which XML and/or non-XML payloads and attachments are sent (typically) over HTTP. The protocol easily accommodates, although it does not actually specify, many common conversational modalities.

A SOAP transmission consists of an XML document structured as a header section and a *body* section, both of which are wrappered inside an *envelope*. The envelope and its contents are referred to as a *SOAP message*. The SOAP message may simply convey data, or it may contain the information necessary to invoke a remote service (Remote Procedure Call).

SOAP is a convenient mechanism for encapsulating data and meta-information about the data. Its advantages over unstructured, non-standard exchange of XML data (such as sending arbitrary XML via HTTP POST) include the following:

- SOAP is *lightweight*, which means it is simple, easy to implement, and adds little to the payload's size or handling requirements
- It's XML
- It is well suited to simple text-based transports layers (such as HTTP)
- It is extensible
- It accommodates security layers unto itself (such as XML Signature and XML Encryption), independent of the transport layer

Composer includes a number of SOAP-enablement features, including the ability to send and receive SOAP messages from a service and the ability to control custom header information, apply (or decode) digital signatures, and use arbitrary attachments to SOAP messages.

For more information on SOAP-related functionality in Composer, see the sections on "Planning your Deployment" and "The Web Service (WS) Interchange Action" as well as Index entries for this book under "SOAP."

XML Signatures

Composer provides support for XML security via mechanisms defined in the XML Signature standard (see **http://www.w3.org/TR/xmldsig-core/**).

The XML Signature specification addresses business requirements for:

- Data integrity (detection of content modification)
- Non-repudiation (irrefutable proof that an order was placed, or a transaction begun, by a specific party)
- Certificate-based authentication (positive identification of transaction participants)

Composer lets you build XML integration applications that support digitally signed input as well as signed output, using the SOAP-header mechanisms spelled out in the XML Signature specification. (See above URL.) You can specify, for example, that a given Web Service *must* receive input that is digitally signed.

Refer to the discussion of the Web Service Interchange action (later in this guide) or the discussion of Composer-service deployment options for more information.

3 Getting Started with exteNd Composer

Novell exteNd Composer is a powerful design environment for creating, testing, debugging, and packaging J2EE-based integration applications. It can run in standalone mode *or* as a subprocess of exteNd Director. The features and techniques described below are applicable in both running modes.

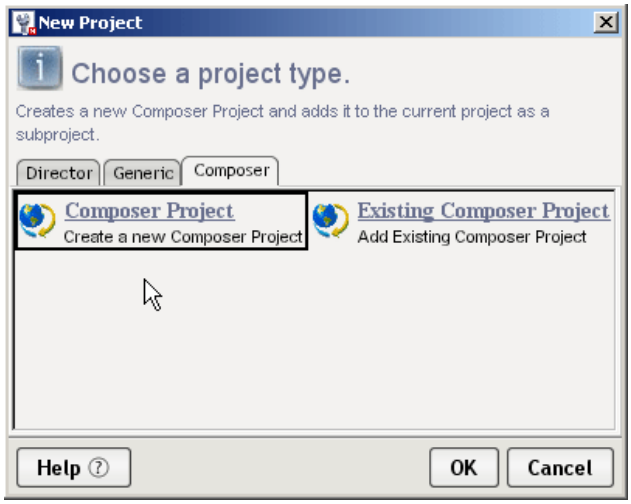
Launching exteNd Composer

You can start exteNd Composer in one of the following ways:

- Doubleclick the **exteNd Composer** icon on your desktop
- From the Windows **Start** menu, click **Programs** then **Novell exteNd**, then **Composer.**
- Open Windows Explorer, navigate to the **\exteNd\Composer\bin** directory, and doubleclick **XC.exe**.
- Start Composer from exteNd Director using the procedure described below.

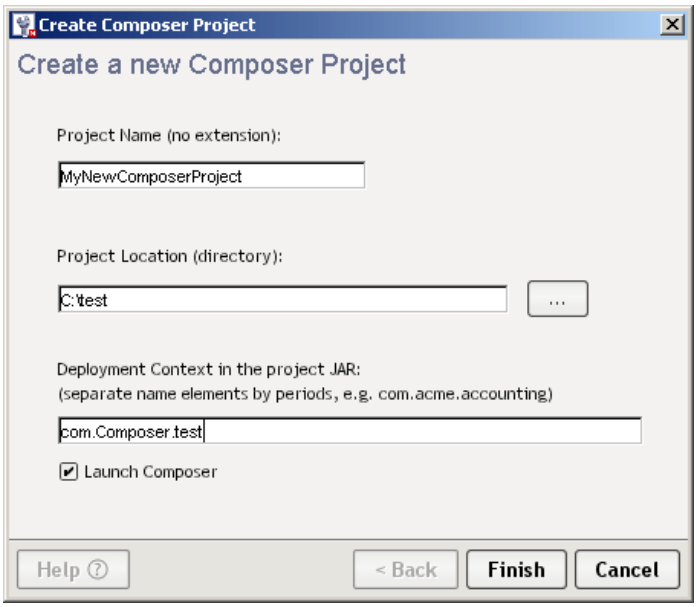
➢ **To launch Composer from Director:**

1. Open an EAR, WAR, or EJB-JAR project in Director, if you have not already done so.
2. Use Director's **File** menu to select **New > Project**. A dialog appears.



3. Choose the **Composer** tab.
4. Select (single-click) **Composer Project** if you are creating a new Composer project from scratch. (It will become a subproject of the currently open Director EAR/WAR/EJB-JAR project.) Otherwise, select **Existing Composer Project** if you are interested in opening an existing Composer project. Again, the project you open will be added, automatically, to your current Director project.

5 Click **OK**. A new dialog appears.



6 Enter a name for the new project under **Project Name**.

7 Specify a **Project Location**. (Use the Browse button, if necessary.)

8 Specify a **Deployment Context** for your project. This can be any series of alphanumeric strings separated by periods.

NOTE: Novell strongly recommends, as a best practice, that you include the word **Composer** in the context string, as shown in the illustration above, so as to provide clear, recognizable namespace separation of Composer deployment artifacts from other artifacts created in other programs.

9 Check the **Launch Composer** checkbox. This will ensure that Composer launches when the dialog is closed.

10 Click **Finish**. The dialog goes away, while Composer launches and comes to the front.

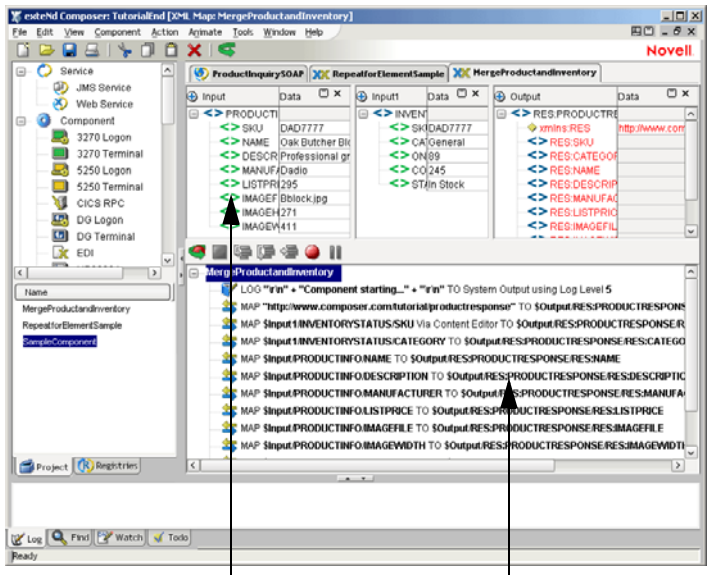
Exiting Composer

Exit out of Composer by selecting **File > Exit** from the main menu, or by typing **Alt+F4**.

Understanding the exteNd Composer Environment

Composer offers a rich design-time environment for creating XML-based B2B integration services. The services you build are deployed to a Java application server (either Novell's, or another J2EE server) and are executed by exteNd Composer Enterprise Server. Composer lets you create, organize, and collect together all of the resources needed (metadata, code, JARs, JSPs, and/or other items) to deploy a web application.

Within Composer, you'll find resource editors (e.g., component editors pertinent to the type of resource in question), a custom script editor, and component editors for creating *action models*. See the illustration below.



You use Composer component editors to create different types of components that can access various data source and map or transform XML structures and data.

How to Get Started

Whether you are using Composer to build a relatively simple XML integration service, or a sophisticated web service, your approach to the building process will most likely follow these basic steps:

1 Become familiar with the **Composer environment** (explained in this chapter).

2 Create a **project**. A project holds all objects for the application you're building. It is stored with a a file extension of "spf."

NOTE: A single .spf file or project can contain many components and resources of many different types, as well as services that use these resources and components. You can deploy your projects straight from Composer or use the exteNd Director to import them into EAR/WAR files.

3 Create **XML Categories** that represent the way you want to organize the sample documents you'll use to build and test your application.

4 Create **XML Templates** that contain the sample documents.

5 Create **Resources** (e.g., Connections, Custom Scripts, XSD or WSDL resources, etc.) that you need for the project.

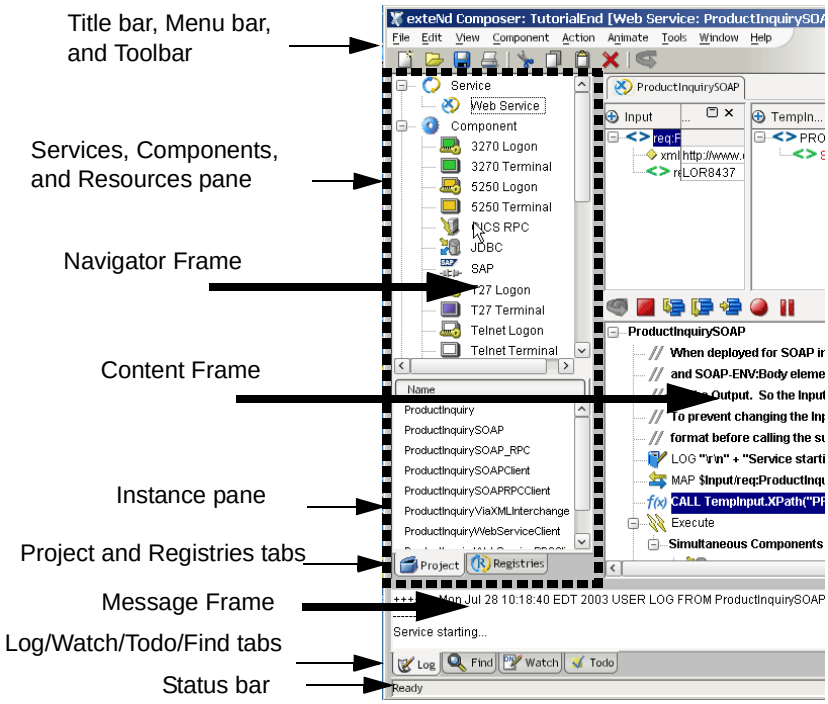
6 Create **Components** that use the templates and resources.

7 Create a **Service** that executes your components.

8 Prepare the project for **deployment**.

About the Composer Environment

You use the Composer main window to create and organize objects. The individual parts of the window are shown below. (The Navigator Frame has been specially highlighted with a dotted-line box.)



Navigation, Message, and Content Frames

The Composer window, by default, exposes three frames: a Navigation frame (on the left), a Message frame (at bottom), and a Content frame (at right). The size of each frame can be adjusted relative to the others by dragging the separator bars that separate the frames. You can also adjust the size of the main window in the usual ways (maximize, minimize, iconize, and stretch). For maximum flexibility in managing “screen real estate,” you can also hide the Navigator and Message frames individually. The Navigator frame’s visibility can be toggled using the left/right arrows between the Navigator frame and the Content frame or by pressing **Control-Shift-N**. The Message frame’s visibility can be toggled using the up/down arrows between the Content frame and the Message frame or by pressing **Control-Shift-O**.

Navigation Frame

The Navigator frame has two tabs at the bottom: a Project tab, and a Registries tab. The Project tab allows you to view Composer objects in Category (top pane) and Instance (bottom pane) views, as shown above. The Registries tab allows you to search for and display registry entries in UDDI-type registries. For more information on this feature, see Chapter 14.

Message Frame

The Message Frame, at the bottom of the Composer window, has four tabs: a Log tab, a Watch tab, a ToDo tab and a Find tab.

Log - This tab allows you to see error messages and Log Action output in real time during your Composer session, eliminating the need to open a log file manually or check system console messages.

Watch -This tab holds the watch list so that users can examine the data values of their variables during the execution or animation of a Composer Service or Component. Watch is a debugging tool which is explained in more detail in “Adding a Watch Variable” on page 119.

ToDo - This tab contains a tree list of ToDo Action items in your open component or service. To find out more about ToDo actions, refer to the section entitled “The Todo Action” on page 153.

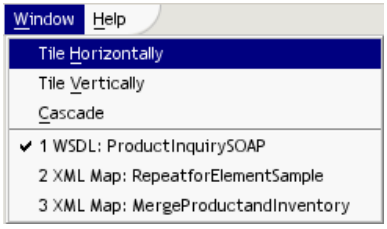
Find - This tab allows you to view search results. See “Searching for xObjects or Text” on page 68 for more information about using the Find command.

Content Frame

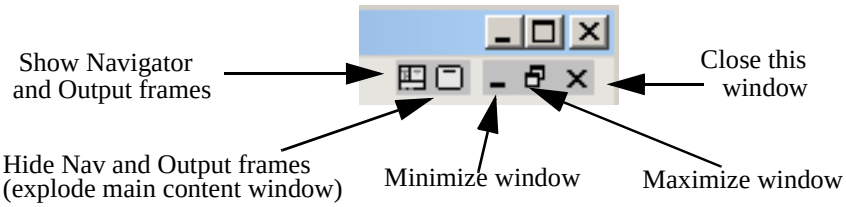
The Content Frame (upper right) displays component-editor content, including DOM trees, action model, and Native Environment Pane.

Manipulating Composer’s MDI Windowing Environment

Composer features a *multi-document interface* (MDI) in which you can have multiple editor windows open (and visible) simultaneously. As shown in the preceding illustrations, by default multiple open windows are shown as a tabbed interface. However, individual windows can be minimized, maximized, and closed, like any other windows. In addition, they can be tiled, cascaded, or arranged arbitrarily by clicking and dragging. You can use the Window commands in the Composer main menubar to control the arrangement of multiple open windows:



In addition, you can hide non-editor panes (such as Composer’s Navigator and Message panes) by clicking on the appropriate icons in the upper right corner of the main screen. This is useful when you are working on an Action Model and you have no need to see the Log pane, navigator tree, etc.



NOTE: The hide/show all panes icons will be visible only when the current editor window has been maximized. But you can always hide or show Navigator and Message panes individually using the commands under the View menu; see discussion below.

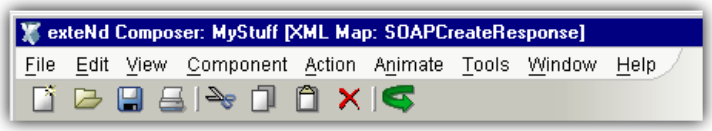
Hiding all non-editor panes explodes the current edit window to take up the whole Composer window (except for the toolbar and menus). This is often useful when an action model contains multiple subpanes containing numerous DOMs, or when you are working in the custom script editor and need more “real estate.”

The View menu offers additional commands that adjust your Composer main window configuration:

- **Navigator Tabs** is the same as typing **Control-Shift-N**. This toggles the visibility of the Navigation Frame of the main Composer window.
- **Output Tabs** is the same as typing **Control-Shift-O**. It toggles the visibility of the Message Frame at the bottom of the window.
- **Document Tabs** is the same as typing **Control-Shift-D**. It toggles the visibility of the Document Tabs. The documents themselves will still be visible, but the tabs separating them will appear or disappear as you toggle.

Using Title Bar, Menus, Toolbars, and Status Bar

You can manipulate objects in Composer using standard Windows menus and toolbars. The following illustration shows the title bar, main menu, and toolbar that appears when you first open Composer.



Title bar

The title bar displays the name of the current project you have open. A *project* is a collection of exteNd Composer services that are developed, maintained, and deployed together.

Menus

The following menu options are available.

Composer Menu Command	Description
File Menu	
New	Used to create new xObjects and Projects. xObjects include: services, components, resources, XML templates, and XML categories. Resources include code tables, code table maps, connections, and custom scripts. See "Creating an xObject" on page 63. Clicking on **New** followed by **xObject** brings up a dialog from which you will select the kind of object you wish to create.
Open	Opens an xObject in the Detail pane. You can also open an object by doubleclicking on it, or by pressing **Ctrl-O**. See "Opening an xObject" on page 65.
Delete	Removes an object from the Composer window and deletes all associated files on disk. You can also delete an object by highlighting it and pressing **Delete**.
Open Project	Opens an existing project.
Save Project As	Saves a project under a different file name to a location you specify.
Delete Project	Deletes selected project from disk.
Deploy Project	Begins the deployment process.
Import xObject	Adds an xObject to your project. See "Importing an xObject" on page 66.
Properties	Displays the properties of the highlighted object. Properties include the object's header information (name and description) along with information particular to the object type. See "Displaying an xObject's Properties" on page 66.
Print	Prints the details of the highlighted object. You can also print an xObject by pressing **Ctrl-P**. See "Printing an xObject's Properties" on page 67.
Recent	Displays a list of recently opened xObjects and projects from which you can select to open.
Exit	Exits the Composer application. If any components are open and have not been saved, you are prompted to save them or ignore the changes. Composer can also be closed by pressing **Alt-F4**.

Composer Menu Command	Description
Edit Menu	
Undo	Deletes the last operation, returning the opened object to the state it was in prior to the operation. The Undo option is only available in a component editor's Action Model pane. See "Creating an XML Map Component" on page 97.
Cut	Deletes the highlighted object(s) or action(s) from the Composer window and puts them onto the Windows Clipboard. (You can also use **Ctrl-X** to cut.)
Copy	Puts a copy of the highlighted object(s) or action(s) onto the Windows Clipboard. (**Ctrl-C** also copies.)
Paste	Copies the contents of the Windows Clipboard into the Composer window. (**Ctrl-P** will also paste.)
Delete	Removes the highlighted object(s) or action(s) from the Composer window and deletes associated files for objects. (Pressing **Delete** with an object highlighted will also work.)
Find	Finds the first instance of a string in an object. The Find option is available whenever you have an xObject open. See also, **Find** on the **Tools** menu.
Find Next	Finds the next instance of the string you entered in the Find Text dialog box. The Find Next option (**F3**) is available whenever you have an xObject open.
Replace	Replaces a string with a new string you enter. The Replace option is only available in a component editor's Action Model pane. See "Creating an XML Map Component" on page 97.
View Menu	
Navigator Tabs	Toggles the display of the Navigator Frame on the left side of the main Composer window.
Output Tabs	Toggles the display of the Message Frame at the bottom of the main Composer window.
Document Tabs	Toggles the display of the tabs at the tops of the component editor pane
XML Documents	Allows you to modify the display of your XML documents. Sub-headings include: Show/Hide, Collapse All, Expand All, View as Tree/Text/Stylized.
Windows Layout	Gives you the ability to select the orientation of the various panels used in the component editor
Tools Menu	The options in this menu change depending upon the object type you select.
Find	Finds xObjects in the project by name, a string it contains, any XML templates it uses, or where a component is used.
Next Occurence	Find the next occurence of the last searched for string. (**F4** will also search for the next occurence.)
Previous Occurence	Find the previous occurence of the last searched for string. (You can also use **Shift-F4**.)
Preferences	Allows you to customize General, Display, Editing and Designer Settings such as establishing an XML editor and Web browser, setting log file details, and entering proxy server settings.
Project Settings	Allows you to set project global variables and manage subprojects

Composer Menu Command	Description
Profiles	Allows you to create, edit and delete Registry Profiles for UDDI, WSIL and ebXML registry types.
Window Menu	Displays all open windows.
Help Menu	
Help Topics	Displays online help for Composer.
Novell on the Web	Displays a submenu with links to help on the World Wide Web.
My Project	Displays an HTML help file you create for a project.
About exteNd Composer	Displays program and version information about Composer.

Toolbar

In addition to the menu options, the toolbar contains the following buttons:

Button	Description
	New. A dialog box allows you to select the component type you want to create.
	Open. A dialog box allows you to select the component type and name you want to open.
	Cut. Clicking this button removes an object from the Composer window and puts in onto the Windows Clipboard.
	Copy. Clicking this button puts a copy of the highlighted object onto the Windows Clipboard.
	Paste. Clicking this button puts the contents of the Windows Clipboard into the highlighted object.
	Delete. Clicking this button removes the highlighted object from the Composer window and deletes its associated files.

Status Bar

In addition to the menu and toolbar, the Composer window has a status bar, at the bottom of the window frame, that displays the state of the currently selected object. When the status bar indicates READY, you can perform an operation on the object.

Understanding Composer Icons

Composer uses icons to represent the different object types. The list below shows the icons and their types.

Icon	Object Type
	The Service group
	Web Service

Icon	Object Type
	The Component category
	XML Map component
	The Resource group
	Code Table
	Code Table Map
	Connection
	Custom Script
	The XML Template Category
	XML Template Group
	XML Template

Navigator Frame

The main Composer window has a Navigator Frame on the left, which in turn can be used in two different modes depending on which tab you've selected at the bottom. The two tabs that control the modes are labelled Project and Registries.

The Project Tab

When the Project tab is selected, the navigation frame contains a "Services, Components, and Resources" pane (top portion) and an "Instance" pane (lower portion).

NOTE: You can adjust the relative sizes of the two panes by dragging the small horizontal divider (between them) up and down.

The contents of the lower pane will change as you select different items in the upper pane. For example, if you click on the Web Service item in the upper pane, the lower pane will be populated with the names of any existing web services in your current project.

Services, Components, and Resources Pane

The Services, Components, and Resources pane contains the four main categories of objects (also known as xObjects) that you'll create with Composer: Services, Components, Resources, and Templates.

Services

Services represent the high level units of work or business partner transactions that occur on the application server after you have deployed a project to your production system. They are used to combine various components you build to create a logical unit of work within the application server environment. Services are the primary objects within a project that are actually executed by exteNd Composer Enterprise Server. Services are primarily concerned with deployment related issues and can be differentiated by the input they receive (URL parameters or XML documents), the type of object that triggers their execution (Servlets or EJBs) and the output they return (XML or HTML documents).

Components

A component is an object that accepts one or more XML documents as inputs, uses a collection of actions to operate on the inputs and returns an XML document as output. A component is usually called by a service and can contain calls to actions or other components. Components are differentiated by their ability to XML enable external data sources. The basic XML Map component can enable XML aware applications. The JDBC component can XML enable relational database systems via JDBC; the 3270 Terminal component (installed separately by the 3270 Connect) can XML enable mainframe transactions through the 3270 terminal data stream; etc.

Resources

Resources are xObjects that perform specialized operations. They are used by services and components to help perform their tasks. Resource types include Code Tables, Code Table Maps, Connections, and Custom Scripts.

Templates

An XML template contains the sample documents, definitions, and stylesheets that assist you in designing and testing a component. You'll create XML categories to contain similar XML templates. Next you'll create XML templates, that will be used as the inputs and outputs for the components you build.

Working with xObjects

You can add an object to one of the four xObject categories using the **New** option on the **File** menu. You can remove an object from a category by using the **Delete** option on the context menu (described below). You cannot remove a main category or add to the main categories in the xObject pane.

Each category has a plus or minus sign. The sign indicates the state of the icon in the tree. If a plus sign appears, you can click it to expand the category to show all child nodes under the category in question. Likewise, if a minus sign appears next to the icon, you can click it to collapse the category, hiding all child nodes.

Using the Context Menu

The top pane has its own context menu, shown next, that can be accessed by clicking the right mouse button inside the pane.

Using the context menu, you can create a new xObject, import an xObject, and paste an xObject that has been copied to the Windows Clipboard. (These topics are addressed separately in other sections.)

About the Instance Pane

The Instance pane lists all user-created objects that belong to a given xObject category. When you click on an icon in the upper pane, its instance objects appear in the lower (Instance) pane.

To change the view of the Detail pane:

1 Highlight an icon in the Category pane to display its contents.

2 From the View menu, select a view option. The options are Icons and List. See "View Menu" on page 41.

Using the Instance Pane Context Menu

xObjects in the Detail pane have a context menu, shown below.

Menu items with functionality above and beyond the standard Windows-based functionality (Cut, Copy, Paste, Delete, Print) are explained in the table below:

Instance Pane Context Menu Command	Description
Open	Makes the highlighted object visible in the content frame
Rename	Changes the name of the highlighted object.
Properties	Displays the properites of the highlighted object. Depending on the type of object highlighted, this could open a dialog with a tabbed interface containing several panels.
Find Where Used	For components and XML templates, this choice opens the Find dialog and automatically searches the project for other objects that reference the selected object.
XML Templates have the following additional menu items	
Edit Sample	Displays a list of XML documents that are included in the selected XML template and allows you to edit them in an XML editor. This option is available only when an XML template is selected.
Edit DTD	Displays a list of Document Type Definition (DTD) files and allows you to edit them. This option is available only when an XML template is selected.

The Registries Tab

When you select the Registries tab at the bottom of the Navigator Frame, the frame assumes this appearance:

There are two panes, labelled Organization and Service (with an adjustable divider between them). These panes are used for searching and retrieving information contained in UDDI registries. For more information, see Chapter 14.

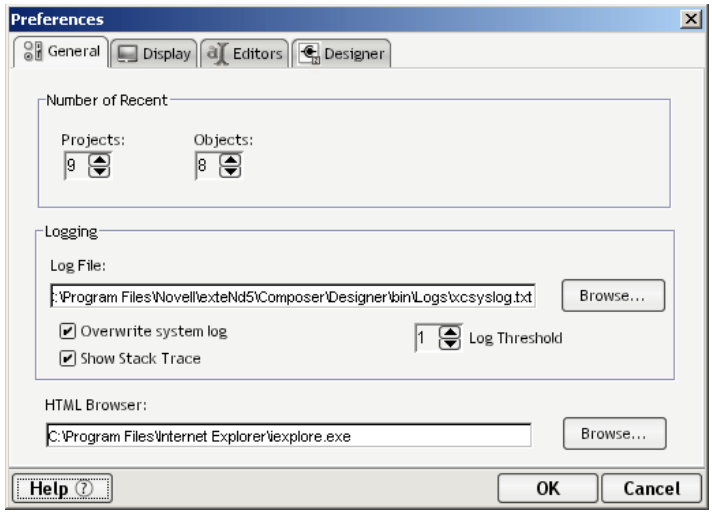
Configuring Composer's Environment

You can configure Composer in a variety of ways to meet your design-time requirements. The Preferences and Project Settings tabs located under the Tools menu are meant to assist you in customizing your user experience.

Setting Preferences

The Preferences dialog (available under the Tools menu) has four tabs: General, Display, Editors and Designer. The function of each tab is described below.

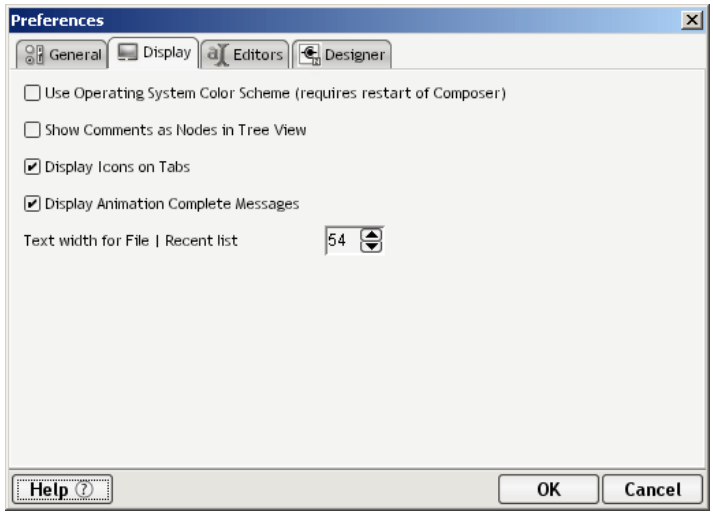
General Preferences



- Using the up/down (spin) control, set the Number of Recent Projects to display under File>Recent
- Similarly, set the Number of Recent Objects to display under File>Recent
- Select a name and location for the System Log File by typing one in or clicking on Browse.
- Check **Overwrite system log** if you want the system log cleared each time you start Composer.
- Check **Show Stack Trace** to turn on the functionality to log the stack traces to the log file.
- Set the **Log Threshold** to a value from 1 to 10. This value is a threshold value that controls which Log actions execute inside a component *and which system messages get written out.* Only Log actions with a priority setting equal to or greater than this number will execute. (See the“Log Priority Levels” section in the “Basic Actions” chapter, page 137.)

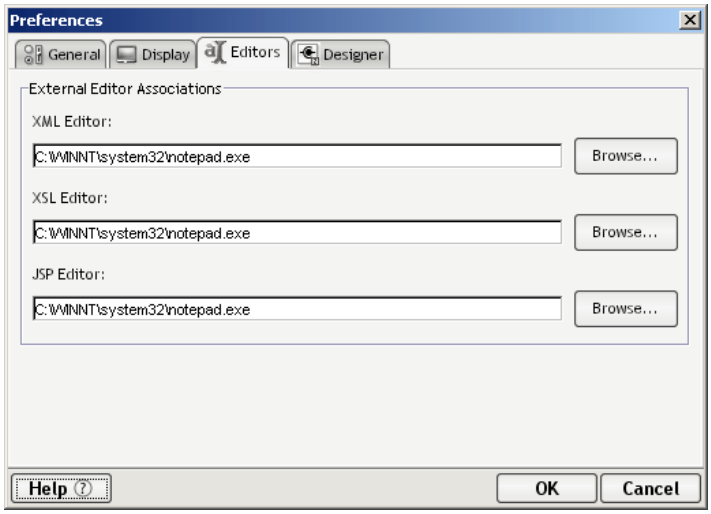
  NOTE: Set this value to 10 if you wish to see all system messages (error messages); set it to a lesser value if you want to see only minimal system messages.
- Select the executable you wish to use as your HTML browser. This will default to Internet Explorer.

Display Preferences



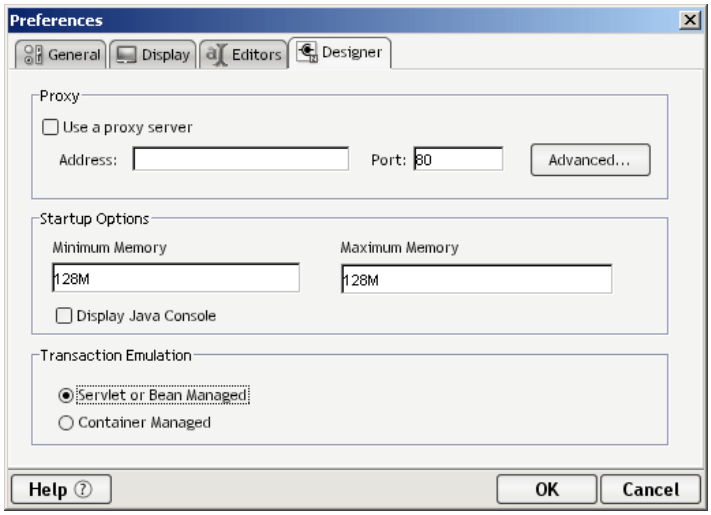
- Check **Use System Settings** if you would like Composer to reflect your default Windows look and feel for colors, menu font sizes, etc. Uncheck to use the standard Composer look and feel.
- Check **Show Comments in Tree View** if you want the be able to see XML comments. Uncheck to hide comments when viewing as a tree.
- Check **Display Icons on Tabs** if you wish to see icons on the Project and Registry tabs. Uncheck to hide the icons.
- By default, **Display Animation Complete Messages** is checked, indicating that "Animation Complete" messages will be displayed. Uncheck if you do not wish to see these messages.
- Using the up/down (spin) control, set a number for the **Directory Display Length for most Recent.** This is used for both recent projects and objects.

Editing Preferences



- Type the fully qualified path of the selected **XML Editor** or click **Browse** to locate the application on your disk or network.
- Specify an **XSL Editor** as indicated above.
- Specify a **JSP Editor** as indicated above.

Designer Preferences



- The **Use a proxy server** check box pertains to the Data Exchange actions during component execution in Composer. If the URL referenced in the action goes through a proxy server, click the **Use a proxy server** check box, then type in the **Address** and **Port** of the proxy server.
- Click **Advanced** to set up Proxy Settings. (These are described in ".Entering Advanced Proxy Settings" below.)
- Under Startup Options, you can set a value for **Minimum Memory** and **Maximum Memory** to be used on startup.
- You can also check the **Display Java Console** box
- If you have installed the Novell exteNd Enterprise Edition version of Composer, you will see a group of controls relating to Transaction Emulation. (These controls are not available in the Professional Edition version.) Select the type of Transaction Emulation to use for design-time testing purposes. (This is a design-time setting only. You control runtime transaction behavior via individual actions.) Your choices are:
  - **Servlet or Bean-Managed**—Means that transacted action models or code blocks will roll back or commit in accordance with explicit calls made via Transaction Actions.
  - **Container-Managed**—Means that transaction scope is managed by the EJB container. If Begin, Commit, or Rollback commands are issued within a component, an IllegalStateException is thrown.

  NOTE: See your *Composer Enterprise Server Guide* for a more detailed explanation of transaction control as it applies to deployed services

.Entering Advanced Proxy Settings

If you check **Use a proxy server** in the **Tools>Preferences>Designer** dialog box, you can enter advanced proxy settings. These settings establish the connections to HTTP and FTP servers, and allow you to exclude certain addresses from using the proxy server.

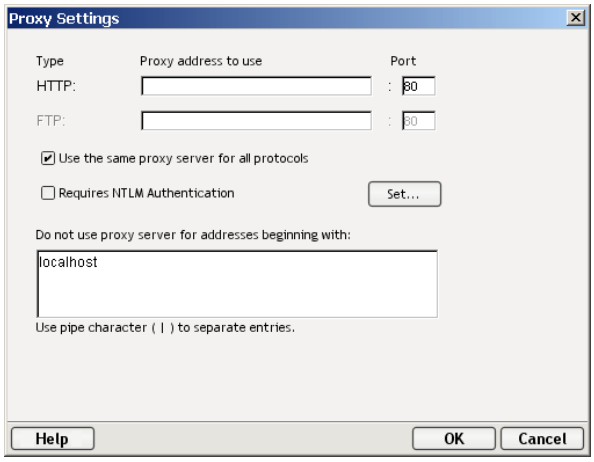
➢ **To enter advanced proxy settings:**

1 From the Tools menu, select **Preferences**.

2 Select the **Designer** tab.

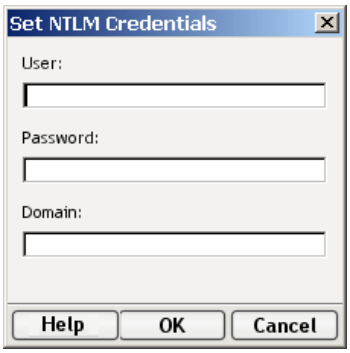
3 Make sure **Use a proxy server** is checked.

4 Click **Advanced**. The Proxy Settings dialog box displays.



5 Type an **Address** and **Port** for the HTTP and FTP servers. If both are the same, fill them in for one server and check **Use the same proxy server for all protocols**.

6 If you will be going to a site that requires NTLM authentication, check the **Requires NTLM Authentication** checkbox. Then click the **Set** button. A new dialog will appear:



Enter the appropriate information for UserID, Password, and Domain, then dismiss the dialog by clicking **OK**.

7 Type the addresses you do not want using the proxy server(s). Separate the addresses with a pipe character (|).

8 Click **OK** to return to the Preferences dialog box.

Project Settings

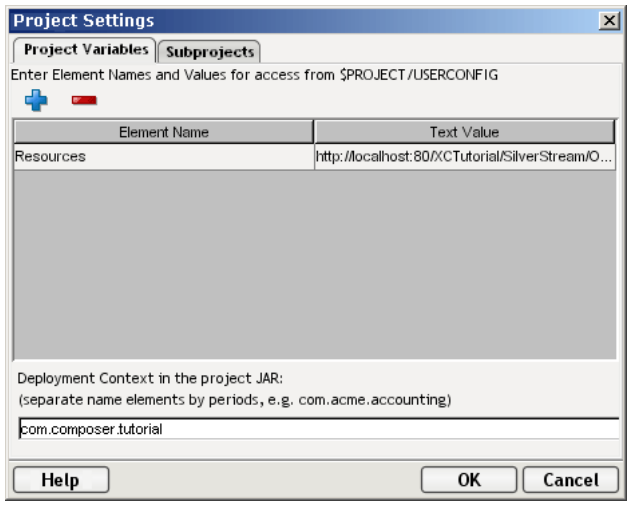
The Project Settings dialog (available under the Tools menu) has two tabs: Project Variables and Subprojects. The function of both tabs is described below.

Project Variables

You can think of Project Variables as being global variables with project scope. They are stored in their own XML document, which gets deployed with other project resources at deploy time. This tab allows you to specify the names and initial values of any global variables you want to use, with intra-project scope. (These variables will apply in the deployed project as well as at design time.) The variables are actually stored in an XML document at an XPath of $PROJECT/USERCONFIG.

➢ **To create project variables:**

1 From the Tools menu, select **Project Settings**. The Project Variables tab is selected by default.



2 Click the plus-sign icon in the dialog's mini-toolbar to add a variable. Click minus to remove a selected variable.

3 After adding a new variable, enter its name under **Element Name** and an initial value under **Text Value**. (Project variables must have String values.)

4 In the text field at the bottom of the dialog window, enter a deployment context for your variables. This can be any number of labels separated by periods. (See illustration above.)

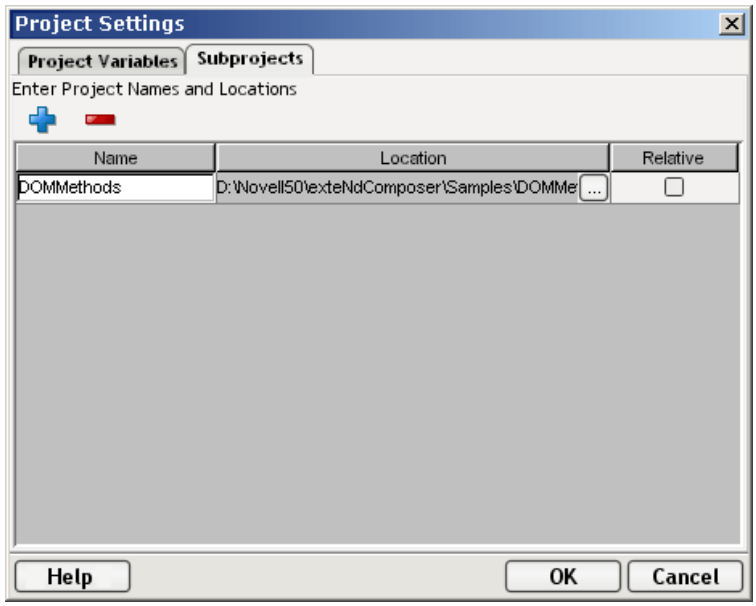
NOTE: Do not use Java keywords such as *protected, default, int, new, try*, etc., in your context string. For the complete list of reserved words, see "Reserved Words" in the appendix.

5 Click **OK** to dismiss the dialog.

NOTE: For further information on Project Variables, see "Creating Project Variables" on page 70.

Subprojects

The Subprojects tab of the Project Settings dialog is where you can add or delete subprojects (other Composer-created **.spf** projects) to your current project.



The advantage to importing projects in this way is that it can be done without actually making new copies of all the necessary files. This subject is discussed in detail in Chapter 4, "Creating and Managing Your Projects". Refer to that section for instructions on using this tab of the Project Settings dialog.

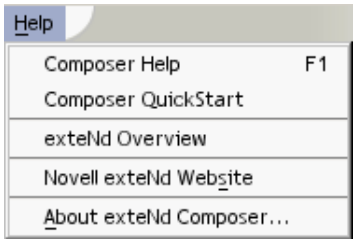
The xconfig.xml and xuserpref.xml files

Your modifications to all the Preferences and Project Settings outlined in the foregoing pages are actually stored in two XML files called **xconfig.xml** and **xuserpref.xml**, located in Composer's **\bin** directory. These files can be edited directly, if you want, but in most cases the quickest, most convenient way to make changes to them is to use the Preferences and Project Settings dialogs as described above.

Composer Online Help

Composer has several forms of online help to assist you when using the program.

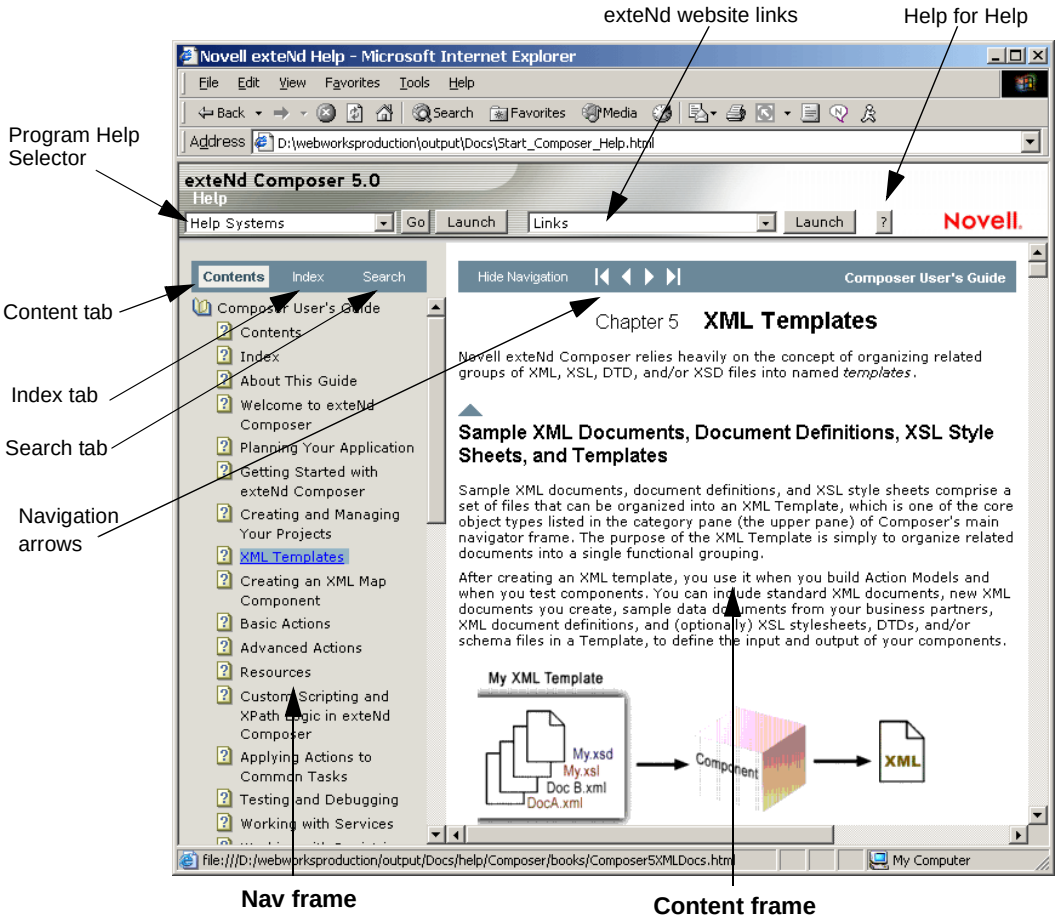
You can access context-based help at any time by using the **F1** key on your keyboard, or by using the **Composer Help** command under the main **Help** menu.



The commands available under the Help menu include those shown below. (Note that the exact makeup of this menu is subject to change.)

Help Type	Description
Composer Help (F1)	Launches the browser-based help system, giving you help for the task on which you are currently working. You can access Composer Help one of three ways. Select **Composer Help** from the **Help** menu to display a table of contents for the online help. From there, you can select a topic to view. You can also press **F1** at any time to display help for the dialog box that is currently selected. In addition, you can get context-specific help from within dialogs by clicking the Help button in the lower left corner of the dialog window (or by hitting F1 while viewing the dialog).
Composer QuickStart	Provides a QuickStart information path to get you started on the right track using Composer to build and deploy web-based applications.
exteNd Overview	Links to an overview of exteNd, the visual integrated services environment for enterprise information systems that allows you to quickly deliver highly interactive solutions that integrate existing business systems.
Novell exteNd Website	Displays the Novell exteNd URL: http://www.novell.com/products/extend/ for information on all the exteNd products, including documentation for the products.
About exteNd Composer	Displays the version number. Clicking on **System** will open a tabbed interface, which displays further information about Status, Licenses and Credits. Clicking **OK** dismisses these informational tabs.

NOTE: The first time you call up the help system, you may notice a brief delay while the system is being loaded and cached into memory. This delay will not occur on subsequent accesses of the help system.



Using Online Help

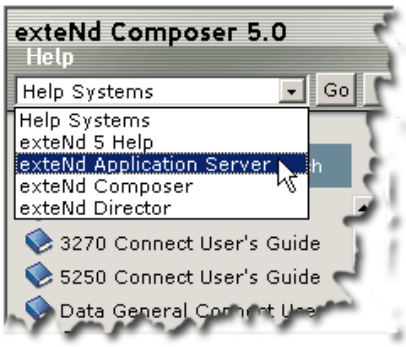
You can call for online help at any time by clicking **F1**. A new free-floating, non-modal window like the one above will appear.

NOTE: If you were in a modal dialog at the time you press F1, you will normally see *context-sensitive help* in the content pane of the help window.

The content of Composer's online help system comes from the product documentation and basically duplicates the PDF documentation, only in HTML form. The HTML files are organized in your installation directory (**Program files\Novell\extend5**). To find the HTML files for a particular product's help, go to **...Docs\help\Composer\books**, where you will see all the .html files for each Composer Product. You can view these HTML files with your favorite web browser, if you want. You do not have to use Composer's viewer.

Note that Composer's online help viewer gives you access to all help topics, covering all installed exteNd products (including Composer Connectors), in one consolidated helpset. Aggregated help for the entire installed product is always available, regardless of which exteNd application, which Composer component editor you might be in, which wizard panel you might be using, etc.

If, for example, you were viewing help for the Composer HTML connect and a question arose in your mind about the Application Server, you could simply change your Help System to exteNd Application Server, as shown below:



Navigating Online Help

Composer's help system offers three navigational options, represented by a Contents tab, an Index tab, and a Search tab at the top of the left-hand (navigation) frame. (See graphic, above.)

Content Browsing

Select the Contents tab to see a complete listing of all help topics covering all installed Composer products.

The "book" icons represent folders. Click any book icon to expand the tree under that folder level. (The content pane will not show useful content when a book icon is selected. Select a topic beneath the book for detailed content.)

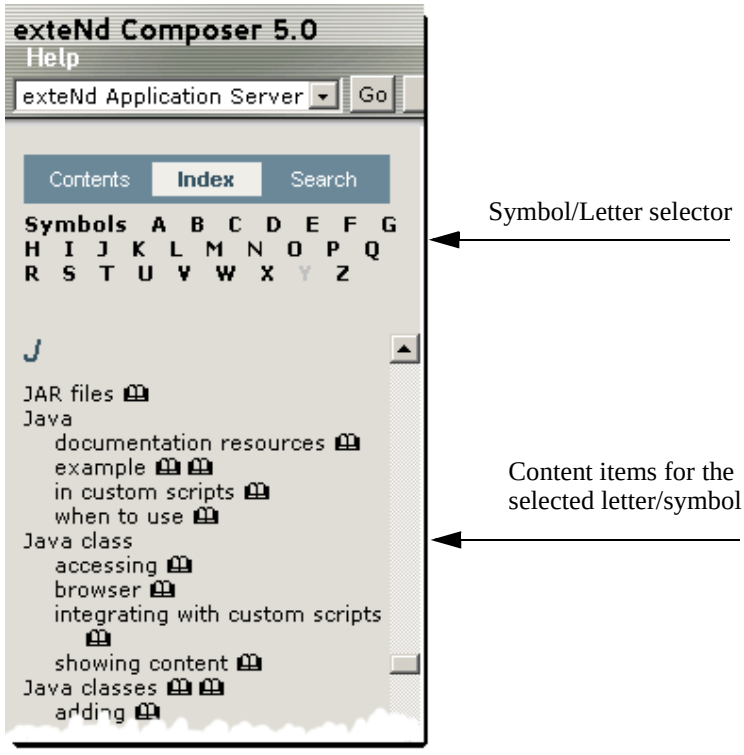
"Page" icons (containing a question mark) represent individual topics for which detailed help is available. Single-click any page icon to see related content in the content frame of the viewer.

NOTE: When focus is in the nav frame, you can traverse topics quickly by using the up-arrow and down-arrow keys. You can also expand a folder (book icon) by hitting the Return key.

Index

Select the Index tab to populate the nav frame of the help window with an alphabetized index of topics. Single-click any topic in the list to see its content.

Select any symbol or letter from the listing at the top of the frame to see the matching topic references in the index listing. This is usually faster than manually scrolling through each symbol and letter.

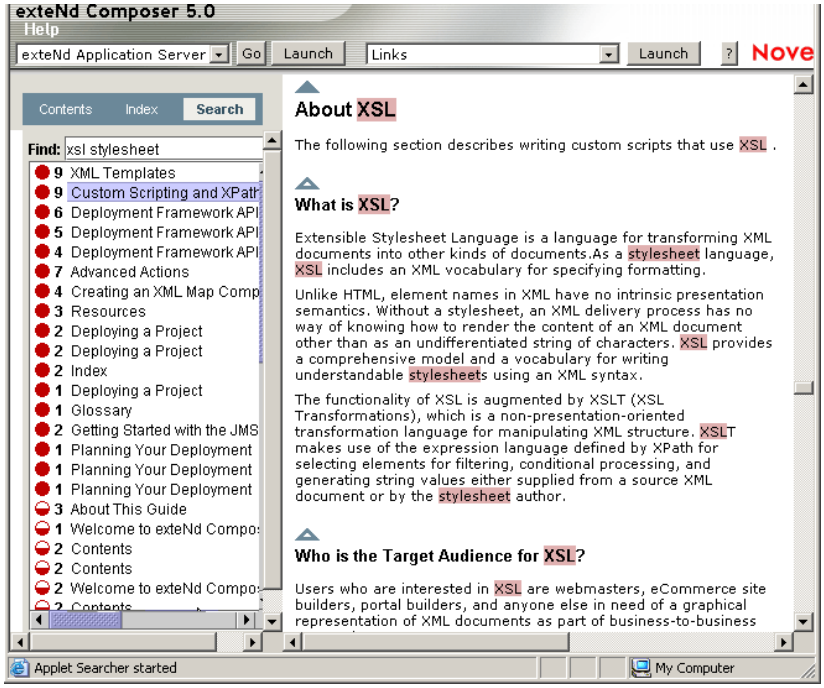


Keyword Search

The fulltext search engine uses a natural language search technology where matches returned from Searches are ranked for relevancy using “relaxation rules.”

- The red circle in the first column of search indicates the relevancy rank, with a completely filled in circle indicating the most relevance and less filled in circles indicating less relevance.
- The number next to the circle indicates the number of matches the search engine found for the topic listed in the third column.
- The third column displays the names of the topics that contained matches. (See graphic, below.) The ranking and relevance ratings improve when search queries are more complex and contain more information.

Single-click any “hit” in the nav frame to see related content in the content frame. The viewer will automatically scroll any relevant section(s) of text into view; and you will see that “hit words” are highlighted in mauve. See below.



The search engine also uses a word morphing technology to find words with common roots. For example, when the term “build” is included in a search string, matches that contain “built”, “builder”, “building”, and “builds” are returned.

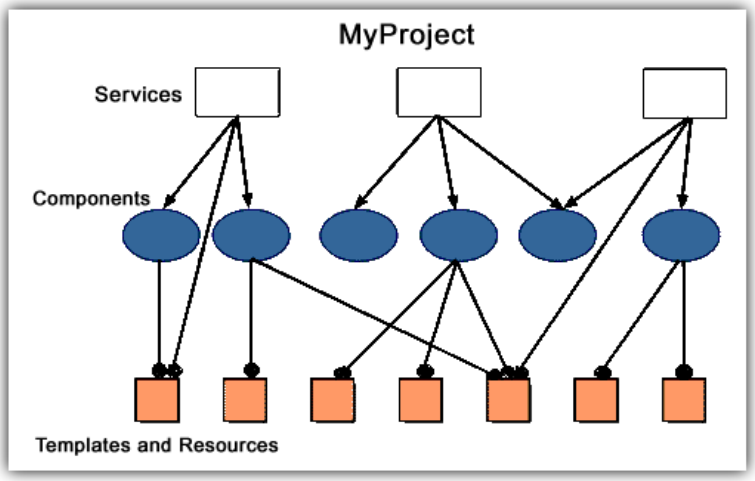
Help likewise performs partial text searches. For example, entering the letter “x” in the Index Find box will locate: *examples*, *execution errors* and *XML Integration*. Help will also find close matches to a whole word so that searching for *execute* finds “execute” and “executes.”

4 Creating and Managing Your Projects

What is a Project?

A *project* is a collection of Composer objects designed to perform XML based B2B integration services. A project holds all the objects for the application you're building, and is the unit that is deployed to the exteNd Server. You may deploy as many projects to an application server as you wish.

The illustration shows the parts of a project.



About Services

A service is used to combine the various components you build to create a logical unit of deployment for exteNd Server. Services are the objects that are actually executed within exteNd Server. A Web Service is started by a Service Trigger object and accepts XML document(s) as input(s), returning XML documents as output. A JMS Service accepts messages as input and is triggered by the arrival of a message on a queue. For more information, see "Creating a New Service" on page 314.

About Components

A *component* is an xObject that accepts one or more XML documents as inputs, uses a collection of actions to operate on the inputs, and returns an XML document as output.

A component is usually called within a *service* and can contain calls to actions or other components. (Services are basically collections of components.)

For more information about how components work, how they are created, and underlying design principles, see "Creating an XML Map Component" on page 97.

About Resources

Resources are xObjects that perform specialized operations. They are used by services and components to help preform their tasks. Resource types include Code Tables, Code Table Maps, Connections, and Custom Scripts.

About XML Templates

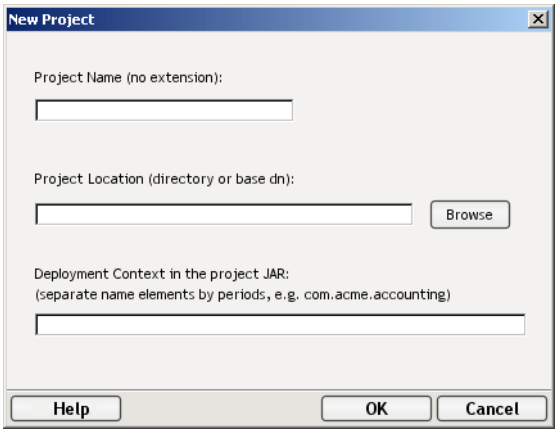
An XML template contains the sample documents, schemas, and stylesheets that assist you in designing and testing a component. You'll create XML categories to contain similar XML templates. Next you'll create XML templates, then use the templates as the inputs and outputs for the components you build. For more information, see "About XML Templates" on page 79.

Creating a New Project

When you first start Composer, a sample project, called Tutorial, is loaded. When you begin your own application, you should start by creating a new, empty project.

➢ **To create a new project:**

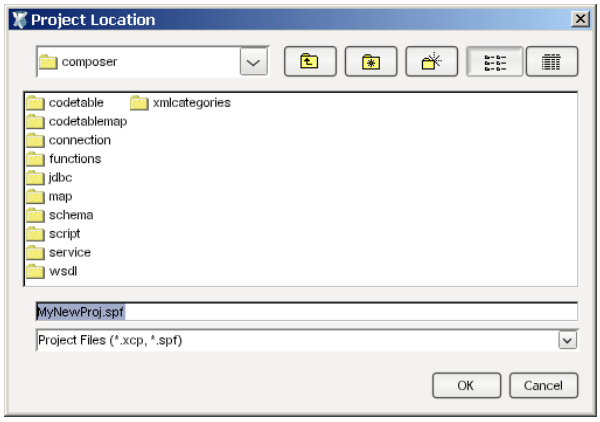
1 Select **File**, then **New**, then **Project**. The New Project dialog box appears.



2 Type in a **Project Name**. This is a required field. Composer adds the project name extension, which is.spf.

3 Select **Browse** to locate the folder where you want your project to reside. The Project Location dialog appears.

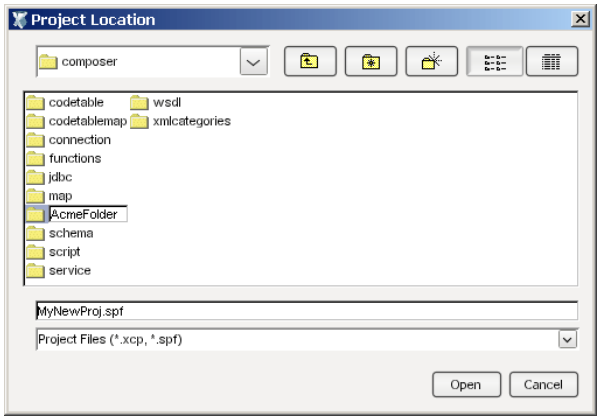
NOTE: If you have a project open, the Project Location dialog defaults to the folder where the open project resides.



4 Navigate to the folder where you want your project to reside.

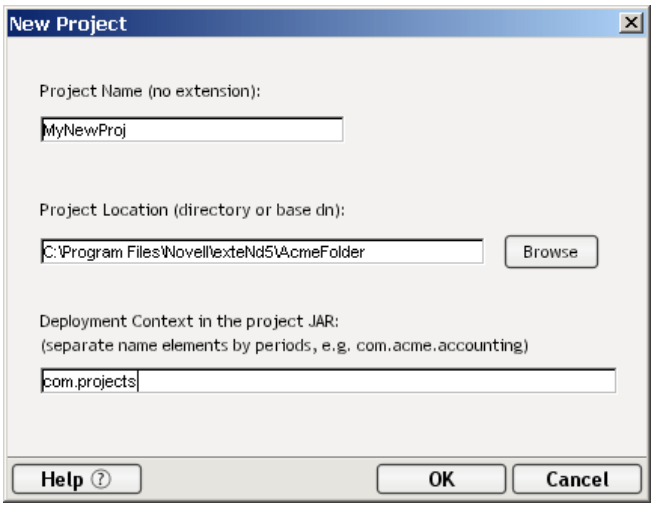
5 To create a new folder in which to save your new project, click the **New Folder** icon. A folder called “New Folder” will appear in the list of folders in the current directory. Click this new entry a single time to highlight it, then click again to rename it to an appropriate **Folder Name**.

6 Click **OK**. The File Location dialog appears with the newly-created folder in the **Look In:** field.



NOTE: The **File Name** in the Project Location dialog defaults to the project name you designated in step 2.

7 Click **OK**. The New Project dialog appears with your newly-created **Project Name** and **Project Location** displayed.



8 Enter a *deployment context* string in the bottommost text field of the dialog. The string should contain labels (no spaces) separated by periods, as in "com.server.apps."

NOTE: The context string should not contain Java-language keywords, such as *try, catch, finally, int, for,* etc. For a complete list of Java keywords, see the "Reserved Words" appendix.

9 Click **OK**. The Composer window appears with the name of the project you just created in the title bar.

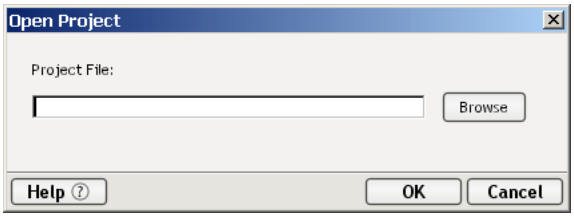
Opening Projects

You can open a project in the following ways.

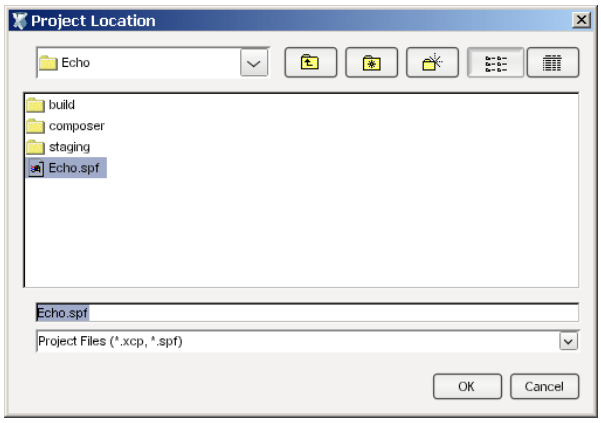
Opening a Project from within Composer

➢ **To open an existing project:**

1 Select **File** then **Open Project**. The Open Project dialog appears.



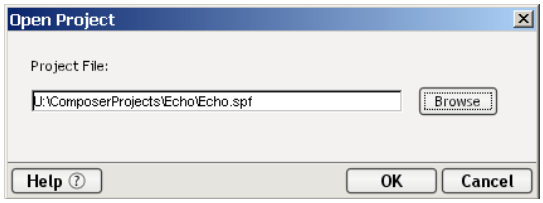
2 Click the **Browse** button. Optionally, you can type in the path of the project you wish to open. The Project Location dialog appears.



3 Navigate to the directory where the project you'd like to open resides.

4 Select the project.

5 Click **OK**. The Open Project dialog appears with the path of the project you just chose in the **Project File** field.



6 Click **OK**. The Composer window appears with the name of project you just opened in the title bar.

NOTE: You can also open a project by selecting a project from the **Recent Projects** list on the File menu.

Opening a Specific Project When Starting Composer from the Command Line

As a startup option, you can launch Composer by running **XC.exe** in command-line mode, and you can specify a project name parameter, such as:

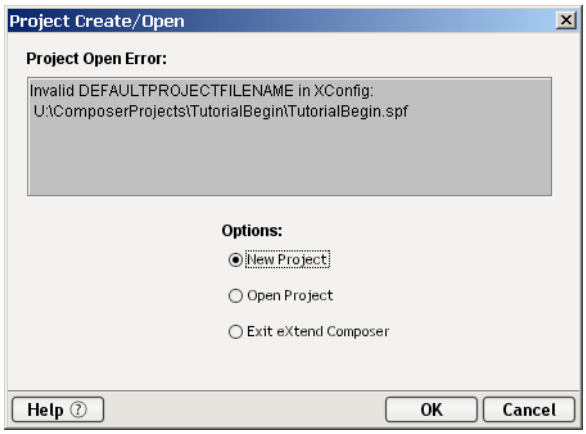
C:\exteNd\Composer\Bin\xc myproject

In this example, **XC.exe** is run with a project file named *myproject.spf*.

Opening a Project when the Recent Project is not Found

When you launch Composer, the last project you worked on is automatically loaded. If you moved the project files, or if you are trying to access another user's project that is inaccessible, Composer may not be able to find your project.

At startup, Composer uses the first command line parameter for the name of the project file to open. If the command line parameter is omitted (or is invalid), Composer uses DEFAULTPROJECTFILENAME from xconfig.xml as the project name to open. If both the command line option is omitted or invalid and the DEFAULTPROJECTFILENAME is omitted or invalid, then Composer displays the Project Create/Open dialog.



The Project Create/Open dialog allows you to create a new project, open an existing project, or exit Composer altogether, as well as providing relevant error information regarding the failure to open an initial project.

➢ **To locate a project at startup:**

1 On the Project Create/Open dialog, do one of the following:

- Select the **New Project** radio button to display the New Project dialog.
- Select the **Open Project** radio button to locate your project.
- Select the **Exit exteNd Composer** radio button to search for the project. For more information about where project files are stored see "Understanding Where Project Files are Stored" on page 69.

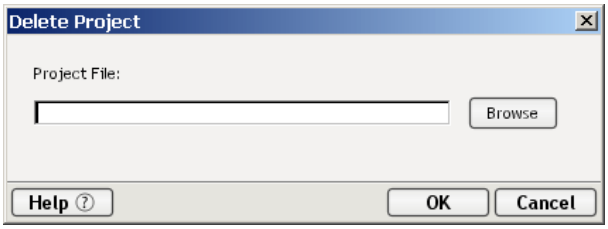
2 Click **OK**.

Deleting a Project

You can use exteNd Composer to delete an entire project and all its objects from your disk drive, or since projects are stored in normal directory structures, you can use standard windows delete functions. In either case, if you do so, the project will be permanently destroyed and unrecoverable by exteNd.

➢ **To delete a project:**

1 Select **File** then **Delete Project**. The Delete Project dialog appears.



2 In the Delete Project dialog do one of the following:

- Provide the full path and project.spf file of the project to be deleted.
- Select the **Browse** radio button to locate the project you want to delete.

3 Click **OK**. The Confirm Project Delete dialog appears.

4 Select **Yes**.

Managing xObjects

xObjects are the building blocks of all XML integration services. In order to build project components, you can either:

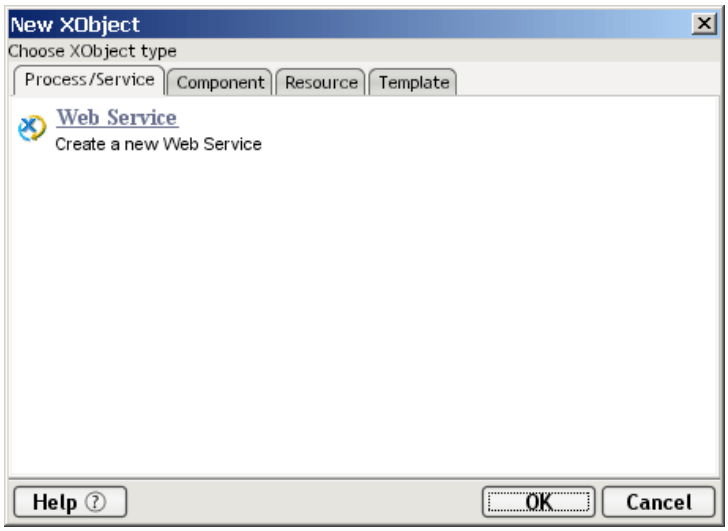
- Create xObjects
- Open existing xObjects
- Import xObjects from other projects
- Do a combination of the three above

Creating an xObject

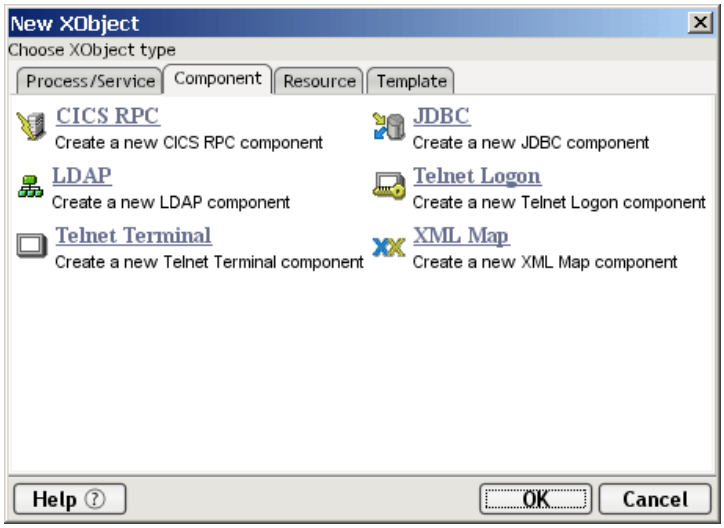
You can create xObjects from the menubar and use them in components.

➢ **To create an xObject:**

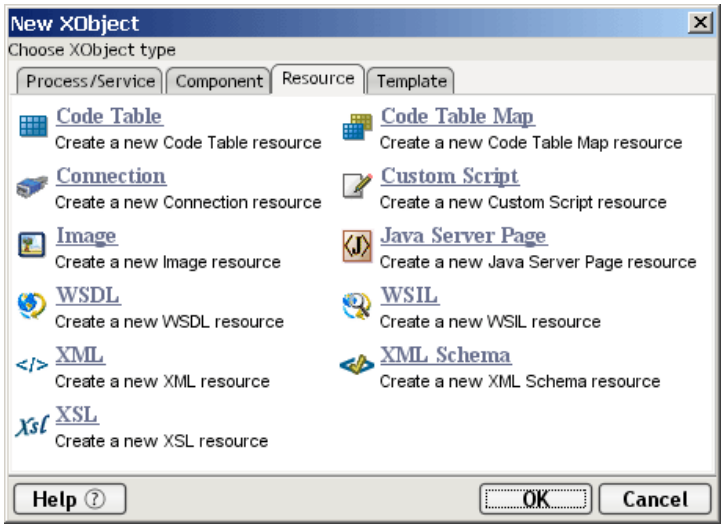
1. From the **File** menu, select **New**, then select **xObject**. Alternatively, you can click with your right mouse button on the type of xObject you wish to create in the navigator pane and select **New**.
2. Tab to select the type of xObject to create. The choices are **Process/Service**, **Component**, **Resource**, or **Template**.
3. If you select **Process/Service**, you can create a new Web Service. You may have additional choices depending on what services you have installed.



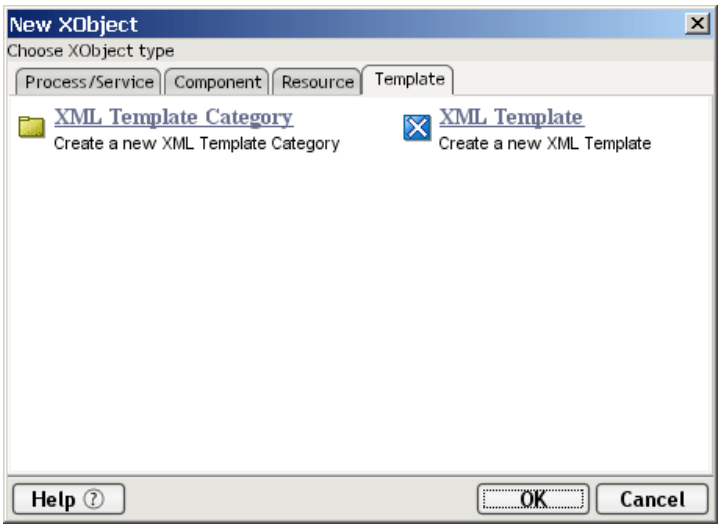
4 If you select **Component**, select a component type. The visible choices depend upon which Connect products you have installed.



5 If you select **Resource**, select a resource type. There are several choices available by default and again, the visible choices will depend on what Connect products you have installed.



6 If you select Template, you have the choice to create a new Template Category, or a new Template.



7 In any of the four cases, once your selection is complete, type a name for the xObject.

8 Complete the rest of the xObject definition screens. Click **Finish** to complete and save the xObject.

The xObject is placed under the category in Composer appropriate for its type.

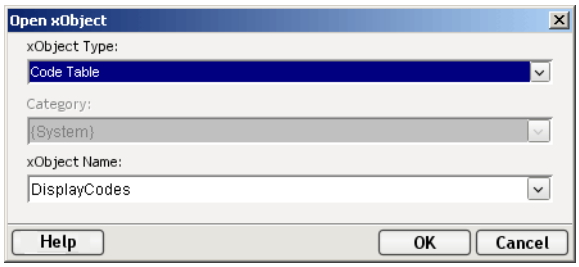
NOTE: The xObjects you create are themselves stored as XML files on your hard disk in a directory of the same name as the category they're in.

Opening an xObject

If there are existing xObjects in the current project, you can open them from the main Composer window to edit or view the contents.

➢ **To open an xObject:**

1 To open an existing xObject, from the **File** menu, select **Open**. Alternatively, highlight any existing xObject in the Instance Pane and press **Ctrl+O**. Either of these methods will display the Open xObject dialog box.



2 Select the xObject type you want to open.

3 Select the xObject.

4 Click **OK**.

The xObject is opened in its own window.

NOTE: To open an xObject directly, doubleclick on it in the Instance Pane, or highlight it in the Instance Pane, click with your right mouse button and select **Open**.

Importing an xObject

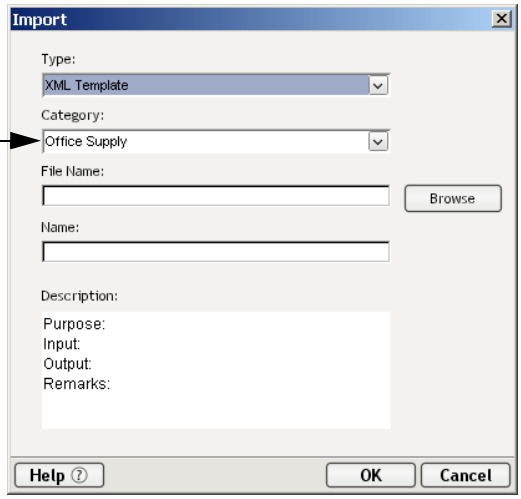
Besides opening xObjects, you can import them from another project or location.

NOTE: The xObjects you create are themselves stored as XML files on your hard disk in a directory of the same name as the category they're in. To import an xObject, select the xObject's XML file as detailed below.

➢ **To import an xObject:**

1 From the **File** menu, select **Import xObject**. The Import xObject dialog box appears.

Category is enabled only when you select XML Template



2 Select the xObject type you want to import.

3 If you selected **XML Template**, select a category.

4 Type the path and filename of the xObject, or click **Browse** and search for the xObject. You may also read in a file from a URL by explicitly preceding your filename with "http://," "https://" or "ftp."

5 Type a name for the xObject or keep the original name.

6 Optionally, type any descriptive text, or keep the original text.

7 Click **OK**.

The xObject is placed under the category in Composer appropriate for its type.

Displaying an xObject's Properties

All xObjects have properties associated with them. Properties include their name, descriptions (header information), and other information specific to the xObject type.

➢ **To display the properties of an xObject:**

1 Highlight an xObject in the Detail pane.

2 From the **File** menu, select **Properties**.

3 Click the tabs in the Properties dialog box to view the Header and XML properties.

NOTE: You can also click the right mouse button and select Properties from the context menu.

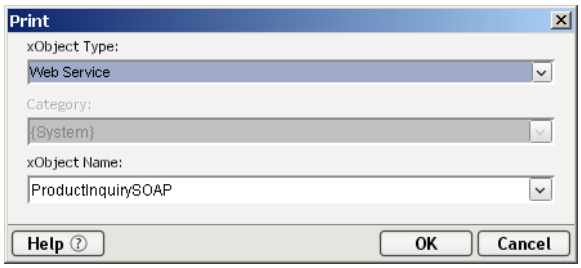
Printing an xObject's Properties

In addition to viewing the properties of an xObject, you can also print the properties. When you print an xObject, the time and date of the print, along with the name of the xObject and all Header and other information specific to the xObject type is included.

If you print the properties of a component, all data concerning the component's DOM structures (See "What is a DOM?" on page 98) as well as its Action Model (See "About the Action Model Pane" on page 112) are printed.

➢ **To print the properties of an xObject:**

1 From the **File** menu, select **Print**. Alternatively, highlight any existing xObject in the Instance Pane and press **Ctrl+P**. Either of these methods will display the Print dialog box.



2 Select an xObject type.

3 Select an xObject.

4 Click **OK**.

5 Set any printer options and click **OK**.

You can also select an xObject first and print it.

➢ **To print a selected xObject:**

1 Highlight an xObject in the Detail pane.

2 Click the right mouse button and select **Print** from the context menu.

3 Set any printer options and click **OK**.

Renaming an xObject

To rename an xObject, right-click the xObject and select **Rename**. Type a new name into the text field.

NOTE: Changing the name of an xObject on its Properties page causes a **Save As** operation, preserving the original and creating a duplicate with a new name. To change just the name of an xObject, use the **Rename** option on the context menu.

Deleting an xObject

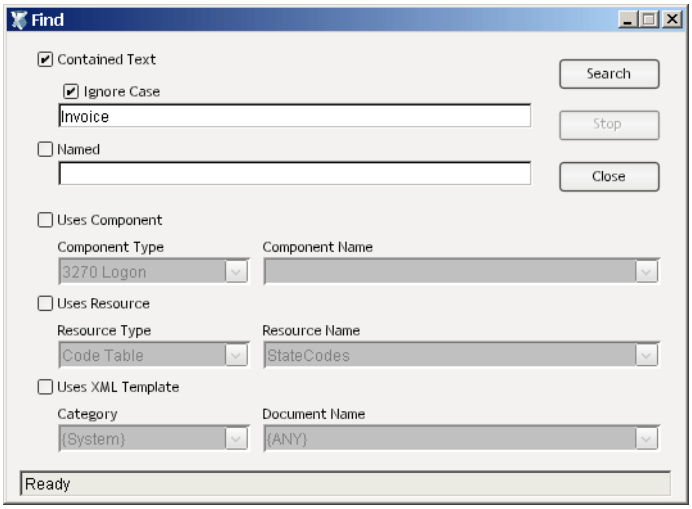
To delete an xObject, right-mouse click it, and select **Delete**. Confirm that you want to delete the xObject.

Searching for xObjects or Text

The bigger your project gets, the more services, components, resources, and XML Templates it will contain. You may find it difficult, at times, to locate information with such an overwhelming number of objects at your fingertips. The Find tool is designed to help you locate xObjects, text within objects, or objects that reference a given XML Template or Component.

➢ **To find an xObject in your project:**

1 Select **Tools** then **Find**. The Find dialog appears.



2 There are several different ways to select and specify the methods you wish to search by.

- **Contained Text** allows you to type a string to search for. This type of search will inspect all xObjects registered with your project. The entire text of the object will be searched, not just its name. **Ignore Case** can be toggled on or off.
- **Named** allows you to inspect all the xObjects registered with a project by name. An asterisk (*) can be used as a multi-position wildcard.
- **Uses Component** allows you to search by component type and, if desired, by name within component type. If you search within a Web Service component, the search will inspect other components that have actions containing calls to the Web Service you have selected. Similarly, if you have the Process Manager installed, searching a Process component will also inspect all sub-process activities that use the target process.
- Uses Resource allows you to search among resource type objects, including Code Tables and Code Table Maps, Connections, JSPs, XML Schemas, Custom Scripts, Service Providers, Service Provider Types, WSDL and WSIL
- **Uses XML Template** allows you to search among your XML templates.
- Any combination of the above search methods can be used.

3 Click **Search**.

When located, the target xObject(s) are shown in list form in the Find tab of the Composer main window. Any component or object in the search-results list can be opened by doubleclicking it. As in the Category pane of the navigation frame, the icon next to the xObject indicates its type (component, service, resource, or XML Template).

Selecting **Tools>Next Occurrence** or pressing **F4** will find the next result for the specified search. **Selecting Tools>Previous Occurrence** or pressing **Shift-F4** will find the previous result for the search.

Viewing System Messages

During the execution of a component, certain messages (e.g., internal system messages from Composer, or text specified by Log actions) are written to a log file, **xcsyslog.txt**. You can specify the location of this file by altering the contents of the **xconfig.xml** file (which is in the **\bin** directory of your Composer design-time installation). Look for the `<LOGFILE>` element in **xconfig.xml** and change its contents to the desired pathname.

NOTE: The easy way to change the log file path is to enter a new path in the **General** tab of the User Preferences dialog. Use the **Preferences** command under the **Tools** menu to bring up this dialog. See "Configuring Composer's Environment" on page 46.

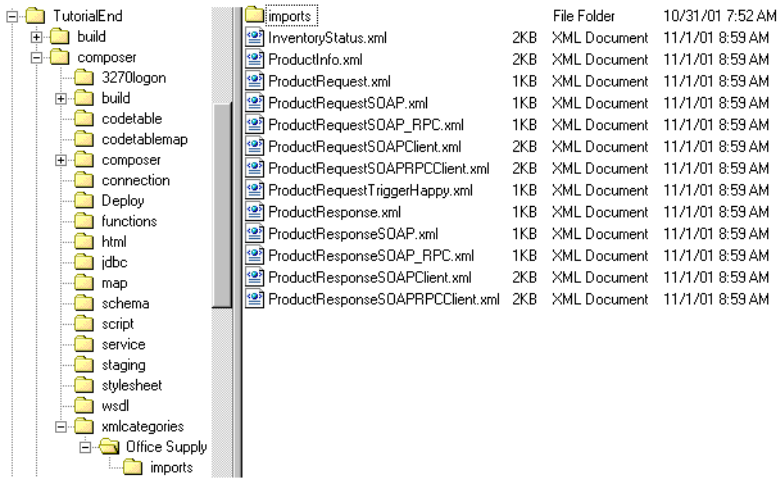
At animation time or when executing a component in Composer, system messages (and Log Action output) will appear in real time in the Message pane at the bottom of the main Composer window. Select the Log tab at the bottom of the window. (If the Message pane is not visible, choose **View > Output Tabs** from the main menu.)

Understanding Where Project Files are Stored

All exteNd objects (projects, XML categories, XML templates, components, resources, services, etc.) that you create in Composer are stored in folders with names that match the object type.

When you create a project, the project file (e.g., myproject.spf) is stored in a folder named after the project (assuming you manually created a new folder for the project). As you build your application by creating XML templates, resources, and components, the created objects are stores as XML files within sub folders of your project folder. So creating a service named "AcceptInvoice" creates an XML file named "AcceptInvoice.xml" that contains all the actions performed by that service. All XML template categories are stored under "XMLCategories" by name of category. All XML Templates are stored under category by name of template. All XML samples for a category (i.e., there could be more than one template) are stored under "Imports" by name of sample document.

The illustration below shows an example of where files are stored.



About Design Time and Deployed Project Files

All XML documents and support files are part of a project. After you deploy your project, the project files are stored in a Java Archive file (a JAR file).

The following table shows what files constitute a project:

Project File Name	Description
[projectname].spf	exteNd Composer project file. Stores startup information for your project. This file is created when you create a new project.
PROJECT.xml	This is an optional file that Composer creates. It contains project variables that you define. See "Creating Project Variables" on page 70 for more information.
[projectname].jar	A Java Archive that is created during deployment. For more information, see the *exteNd Server Guide.*
*.xml, *.xsl	All XML samples, definitions, and stylesheets you use in designing your application are stored in folders under the project folder.

Creating Project Variables

A *project variable* allows you to designate a value for an element and use the specified element globally in all components and functions you create. Unlike ECMAScript "globals," which are scoped to the component in which they are used, project variables are scoped to a service's session, which means they can be used by any number of components running inside a Composer service.

Project variables are implemented as values stored in an in-memory DOM called $PROJECT. This DOM is in turn derived from a file that Composer creates for every project called **PROJECT.xml** (which later gets deployed to the server).

NOTE: Changes to project variables that occur at runtime are *not* persisted across service invocations. In a production environment, **PROJECT.xml** is read-only. (To create *persistent* globals, you would need to read and write your own scratch file using XML Interchange actions.)

Because they are global in scope, project variables can perform important functions during both design-time development of your project and runtime maintenance of your project after deployment.

At design time, project variables provide a convenient means of centralizing project-wide values that might need to be used in multiple places in a project. At deployment time, the project variable file (**PROJECT.xml**) provides a convenient way of updating a project's static variables. After deployment, you can conveniently change the behavior of multiple deployed components and services by updating just the deployed **PROJECT.xml** file on the application server.

Examples of items that might best be stored as project variables include:

- Any URL referenced within components for items such as:
  - Log file paths
  - DTD and Schema paths
  - XSL stylesheets
  - XML Interchange URLs
- Send Mail—Mail Server Identification
- Authentication information needed for establishing connections with databases or back-end systems
- Message queue names
- Versioning info applicable to your services and components

The process of updating the **PROJECT.xml** file is described in detail in *exteNd Server Guide*.

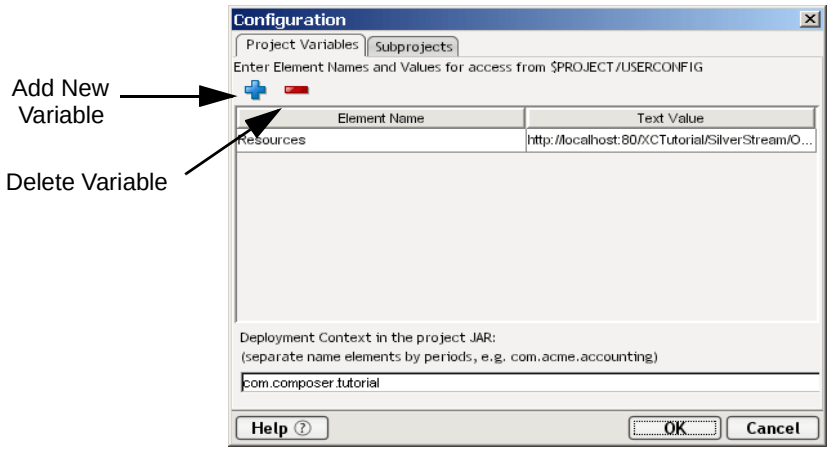
Adding a Project Variable to a Project

You create the names of project variables that map to specific values. By making the reference of certain information indirect through a project variable, you can change the data in one place and be assured that all places where it is used will get the same new values.

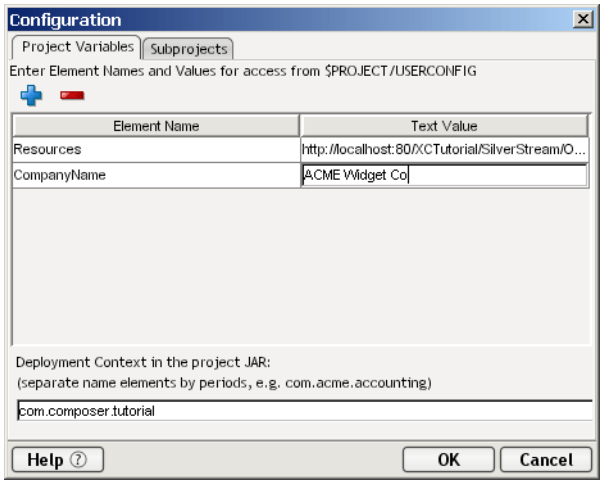
➢ **To add a project variable:**

1 Select **Tools** then **Project Settings** from the Composer window. The Project Settings dialog appears.

2 By default, the **Project Variables** tab will be displayed.



3 Click the **Add New Variable** button. A blank field appears in the Project Variable window.



4 Click in the blank **Element Name** field and type an element name. For example, type the element "CompanyName." Do not use spaces.

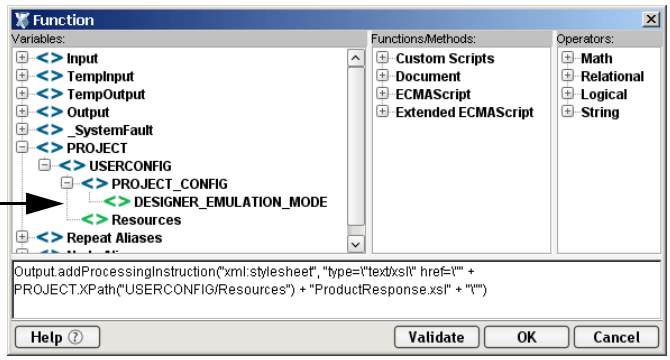
5 Click in the blank **Text Value** field and type a text value. For example, "ACME Widget Co."

6 Click **OK**.

The Element Name and Text Value that you just created are now stored in a project XML file called PROJECT.xml. This file can be manually edited after you deploy your project, if you need to change the variable value.

The Element Name and Value are automatically added to Composer dialog boxes for your use in building components. For example, the variable is available for use in functions.

A project variable is available for use in building components.



Creating Project Variables Dynamically

In addition to creating permanent (static) project variables, you can also create project variables dynamically within a component or a service.

The $PROJECT DOM is always present in the DOM lists (dropdown menus) that display in the Map action dialog. It's a very simple matter to create a project variable and assign it a value, because you can map to the $PROJECT DOM the same way you would map elements and element values to any other DOM (including via drag-and-drop).

NOTE: You can view the contents of the $PROJECT DOM in tree, text, or stylized form at any time by choosing **Window Layout** from Composer's **View** menu and making the $PROJECT DOM visible. See the discussion under "Using Window Layout and Show/Hide in the Component Editor" on page 104 of this guide.

If you look at the structure of the **PROJECT.xml** file, you'll see that the root element is called USERCONFIG. User-defined variables are attached to this node as child elements. The string values of the child elements are the values of the project variables corresponding to the element names.

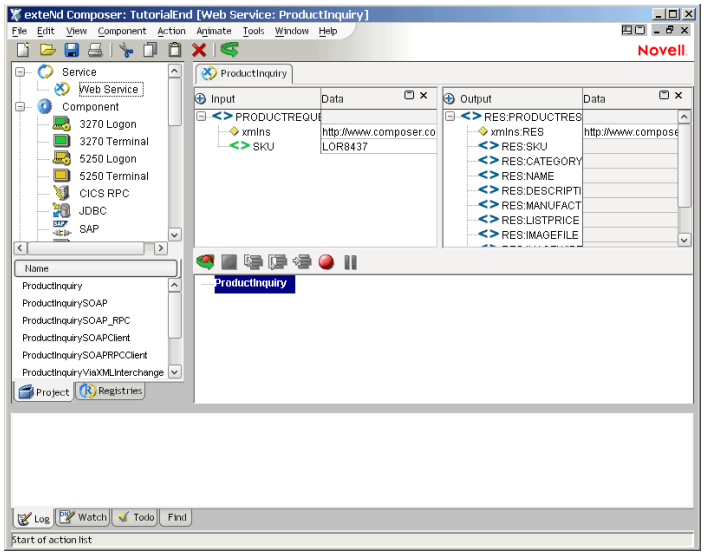
In addition to user-defined project variable names, you will also see Composer-defined elements under USERCONFIG, because Composer uses the **PROJECT.xml** file to persist certain project preference values.

Dynamically created project variables are, of course, volatile. You can use dynamic project variables for the lifetime of the executing service (which may in turn call many components that use it). When the service finishes executing, the dynamic variables are destroyed, since they were created in memory.

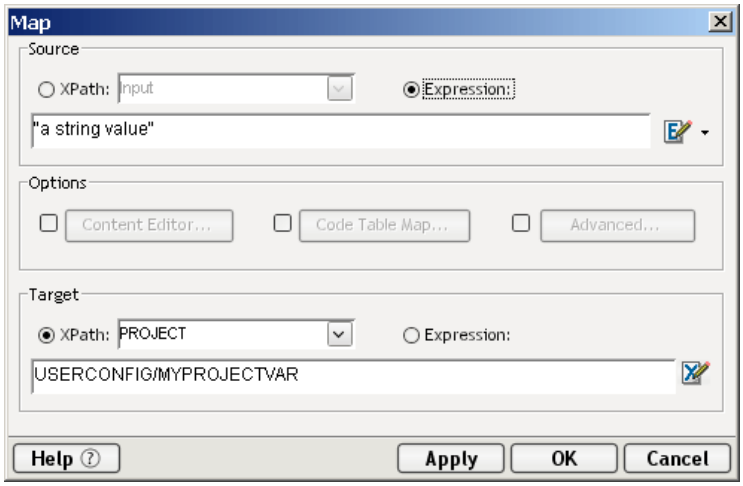
You can reassign values to a project variable as many times as needed, by mapping to its node in the $PROJECT DOM. An example of this follows.

➤ **To create a dynamic project variable and map a value to it:**

1 Doubleclick a service in the instance pane. The Service editor window appears.



2 In the main menubar, select **Action,** then **Map**. The Map dialog appears.



3 Click the **Expression** radio button in the **Source** section of the Map dialog.

4 Type in a value for your project variable in the **Source** field. If the value is a string, don't forget to enclose it in double quotes.

5 There are two ways to enter the target expression:

- Click the **Expression** radio button in the **Target** section of the dialog and type

  `PROJECT.createXPath("USERCONFIG/MYPROJECTVAR")`

  in the **Target** field, where *MYPROJECTVAR* is the name of the project variable you wish to create—**or:**

- Click the **XPath** radio button under Target, select PROJECT from the dropdown menu, and in the field underneath type:

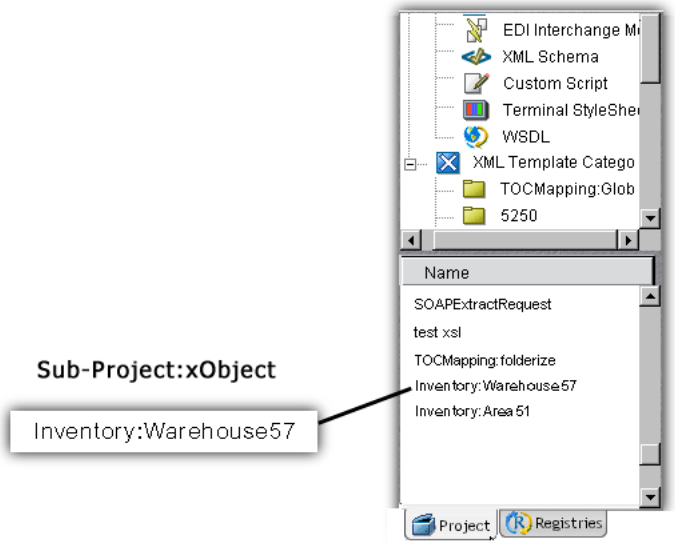
  `USERCONFIG/MYPROJECTVAR`

  in the target field, where *MYPROJECTVAR* is the name of the project variable you wish to create. (See above illustration for the completed dialog's appearance.)

6 Click **OK**. The dynamic variable you just created now appears in the Action pane of the Service editor window.

Subprojects within Projects

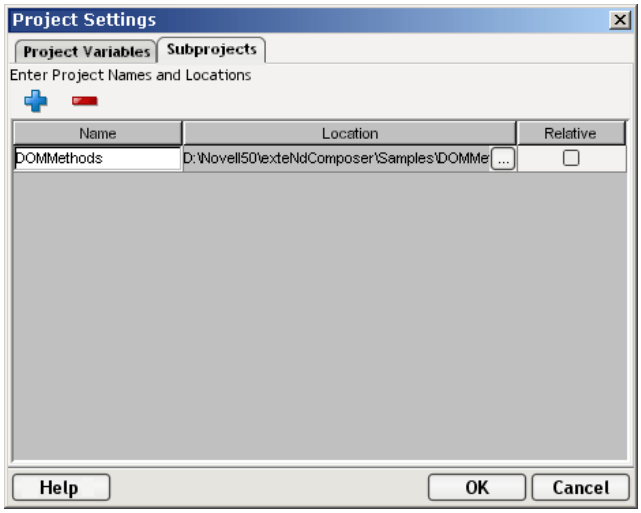
You can include other Composer projects within your current project—a feature designed to foster rapid application development via reuse of existing xObjects. When external projects are reused in this fashion, they are called *subprojects*. A subproject's xObjects are exposed in the current project's Category and Instance panes, in the usual way, except that a project prefix appears before the name of each object, to identify the object as coming from a named subproject. An example is shown below.



➢ **To include a Subproject in a Composer Project:**

1 Under the **Tools** menu on Composer's main menubar, choose **Project Settings**.

2 Select the **Subprojects** tab of the Project Settings dialog, as shown below.



3 Click the **plus-sign** icon in the upper left part of the dialog to add a subproject. A dialog will appear, allowing you to browse your file system. Choose any **.spf** file that was created by Composer; it will appear in the list of subprojects.

NOTE: If the **.spf** you choose already contains a subproject of its own, you will get an error dialog advising you that you cannot add subprojects containing subprojects.

4 Click the **Relative** checkbox if you want to change the location of the subproject to a relative path (to the main project.spf). If the project is on a separate drive than the main project, then the Relative checkbox is disabled.

5 To remove a subproject, select it and then click the **minus-sign** icon.

6 Add as many subprojects as you like, by repeating Step 3.

7 Dismiss the dialog by clicking **OK**. Your subproject's xObjects will appear in the detail pane of Composer's nav frame. They can be distinguished by the appearance of a namespace/colon prefix on each xObject name.

Imported xObjects versus Subprojects

To achieve object reuse, you can import xObjects directly, one-by-one, into a given project, rather than take advantage of subprojects. (See "Importing an xObject" elsewhere in this chapter.) But the disadvantage of *importing* an xObject is that it results in the original object's underlying XML files being *copied* into the current project. This can pose code maintenance problems, in that alterations or updates of the original xObject will need to be made, also, in any *copies* of that object that might exist in projects that imported the object. *This is not true for subprojects.* When you include an external project within your current project, no additional copies of the subproject's source files are made. All "source code" stays in one place, simplifying maintenance.

Nesting of Subprojects

Nesting of subproject beyond one level s is not supported. A given project can have any number of subprojects, but they must all be at the same level (one level deep). This also means that a project containing one or more subprojects cannot serve as a subproject for another project. For example, consider the case where Project A contains a subproject named Project B. A third project called Project C could *not* use Project A as a subproject, although it could use Project B.

If you attempt to add a subproject to your current project, and that subproject contains its own subprojects, you will get a warning as follows:

Scope and Visibility of xObjects and Variables in Subprojects

The sharing of xObjects and variables among projects and subprojects is limited by certain scoping rules that you should be aware of.

1. **xObjects:** A project can access a subproject's components (and other xObjects), but the subproject cannot access the parent project's objects. For example, if Project A contains Project B as a subproject, the components in Project B (the "child" project) cannot address components or resources in Project A (the parent).
2. **Project variables:** Variables derived from the $PROJECT DOM (see "Adding a Project Variable to a Project" earlier in this chapter) belong to *the project in which they were created*. Components and services in Project A cannot "see" project variables belonging to Project B, nor vice versa.
3. **ECMAScript variables and functions:** The lifetime of script variables is always scoped to the component. When a component goes out of scope, any ECMAScript variables it may have used also go out of scope. Custom Script resources in a subproject, on the other hand, are accessible to the main project, via a built-in Composer ECMAScript extension called the `Projects` object. For example, suppose that the current project, Project A, contains a subproject, *MyOtherProject*; and suppose MyOtherProject contains a Custom Script resource in which there is a function called `salesTax()`. A component in Project A can use the salesTax() function by calling:

```
Projects.MyOtherProject.salesTax()
```

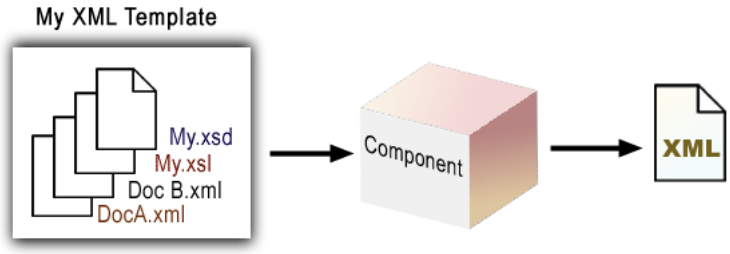
5 XML Templates

Novell exteNd Composer relies heavily on the concept of organizing related groups of XML, XSL, DTD, and/or XSD files into named *templates*.

Sample XML Documents, Document Definitions, XSL Stylesheets, and Templates

In order to simplify working with XML data at design time, Composer lets you defined XML Templates. The purpose of the XML Template is simply to organize related documents into a single functional grouping. For example, it's not unusual, when designing an XML integration application, to have one or more sample input documents that represent hypothetical "incoming data." These input documents might or might not conform to a particular schema (.xsd) or DTD. They might or might not be associated with XSL stylesheets. You may or may not also want to associate various kinds of *fault documents* with the service. In Composer, you would typically organize *input* documents into one XML Template and create a different XML Template (with any or all of the above-mentioned ancillary items) to hold sample *output* documents. The "XML Template" wrapper identifies a group of related documents: sample XML docs that go with particular stylesheets, schema docs, and/or DTD files, and/or fault docs that need to be used in association with each other.

At design time, the XML sample documents in your templates serve as exemplars, or "hints," to enable Composer to display proper document tree views in the various GUI pieces that need to show your service's inputs and outputs. In this way, it becomes possible for Composer to translate simple UI gestures (like drag-and-drop) into XPath and ECMAScript expressions that can be used to carry out mappings and transformations at runtime. (Composer does the "hard work" of generating XPath and DOM methods so that you don't have to.)



You supply both the input and output XML Templates

About Sample XML Documents

A sample XML document is nothing more than a *representative model* of the data your component or service will process: it contains the same elements, attributes, and structures. For example, if your application will process Company ABC's invoices, you might use a sample invoice when building the application. The sample (if it's truly representative) will have exactly the same XML structure as the invoices that will be processed.

One of the most important parts of planning and designing an XML integration application is determining all of the possible kinds of sample documents your components might need before you begin development.

The types of sample documents you may need are:

- Sample input documents. These could include XML documents provided by a standards organization (e.g., cXML, OAG, and OFX) containing the elements and structure for the particular kind of data you want to process.
- Sample output documents.
- Sample intermediary ("temporary" or scratch-pad) documents.
- Sample fault documents.
- XSD (schema) or DTD documents. (These can be stored in a project as separate resources; they are merely *referenced* in XML Templates.)

An important concept to note is that sample documents used in designing a component are not used on the server at runtime. The samples in an XML Template are really only design-time hints. They cannot be used as sources of instance data. (For that, you'd probably want to use XML Resource documents. See "About XML Resources" in the chapter on Resources.)

NOTE: If you need to initialize any data elements with hard-coded values, you can do it programmatically in the action model, by mapping an ECMAScript string or number to XPath locations, as needed. You can also load an XML Resource and create mappings from it to (say) an input document using drag-and-drop gestures.

The sample document is a design aid that allows you to visualize the data manipulations that need to happen at runtime. At animation time (during testing or debugging) you can watch element data in the sample change locations or values, or show up in output, etc., in real time, in response to XML Map actions, ECMAScript operations, and so on. After watching the data change in real time during step-through/step-over debugging, it's easy to forget that the data values are just design-time values—placeholders, if you will. At runtime, Composer merely executes the map actions, XPath and ECMAScript operations, etc. that you specified in development.

About XML Validation Documents (DTDs and Schemas)

Document Type Definition and XML Schema Definition files (DTDs and XSDs, respectively) can be used to define and validate XML documents. Schemas and DTDs define the grammatical rules of the document, such as which elements must be present and what the structural relationships are between the elements.

Recall that a schema differs from a DTD in several ways, including:

- The XSD file is a true XML file which itself conforms to a schema defined by W3C. DTDs, by contrast, are not true XML files.
- A schema can enforce *data typing*, so that if an element requires (for example) data that takes the form of a date in CCYY-DD-MM format, such a requirement can be specified (and strictly enforced).
- A schema allows namespace declarations, so that elements can be uniquely identified as belonging to a given document vocabulary.
- Schemas are designed to be granular, providing for maximum reusability.

- Schemas are flexible in terms of allowing an author to specify strict enforcement of some grammar rules but lazy enforcement of other rules, within the same document.
- Schemas are extensible in that they allow authors to define all-new custom data types.

For these and other reasons, schemas (XSD files) are gradually displacing DTD files for definition and validation of XML documents.

Runtime Validation versus Design-Time Validation

Schema and DTD validation are enforced by Composer only at design time. At runtime, no validations (other than a well-formedness check) are performed on incoming or outgoing data. Nevertheless, you can force runtime validation to occur by means of ECMAScript (used either in a Function action, or wherever ECMAScript is permitted in Composer). For example, suppose you want your service to validate the Input document. You would execute this expression:

```
result = Input.validate();
if (result == true)
// do something
else
// throw fault
```

If a schema is associated with Input in the XML Template for Input, that schema will be used for validation when the above code executes. If no schema is referenced anywhere, the `validate()` function simply performs a well-formedness check and returns a boolean result.

NOTE: The validate() function will not use DTDs.

About XSL Stylesheets

As part of the set of files you use in a component, you can include an XSL stylesheet. An XSL stylesheet defines the display properties of an XML document. You create or obtain the stylesheet external to Composer. The stylesheet may be useful for a component of your application that is creating a page to be displayed in Web browser.

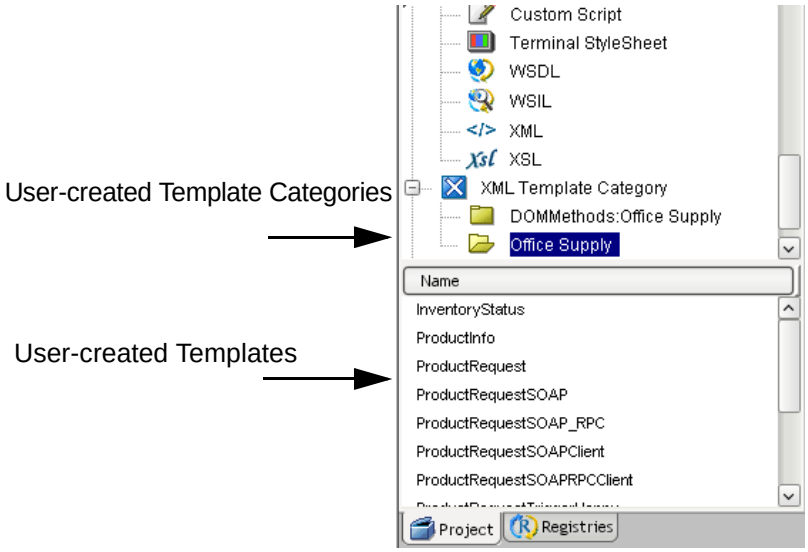
About XML Templates

An XML Template contains the sample documents, document definitions, and XML stylesheets that comprise a set of sample documents that can be used in designing your components. You'll create XML Templates early in the component design phase, then use them to specify the inputs and outputs of the components you build.

XML Templates exist primarily so that you can use and test many types of sample data at design time. It is possible to have two XML documents with different structures that both have to be handled without error by the same component. For example, if you are using an industry standard purchase order document as input, but one of your customers uses a slightly different version of that document in his business (e.g., it has some optional elements missing), you can load your customer's document into a component for testing purposes. Your component must be able to handle the different document versions, and you can test several cases by collecting all your samples into a template that serves as an input for a component.

About Template Categories

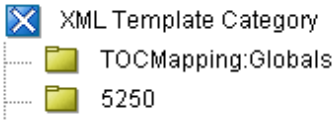
Instances of XML Templates are collected into Template Categories. The Template Categories have user-assignable names and appear as folders in the XML Template Category pane of Composer's navigation frame. The members of a given category appear in the Instance pane under the Category pane. See below.



Your application can have many input and output documents, so you will want to organize them within XML Template Categories. Within an XML Template Category, you can organize templates in a way that makes sense for your application. For example, you can create folders for:

- Specific business processes (e.g., Accepting a Purchase Order, Sending an Invoice, Receiving an Invoice)
- Industry standard XML documents

Here is an example of what your organizational scheme might look like.

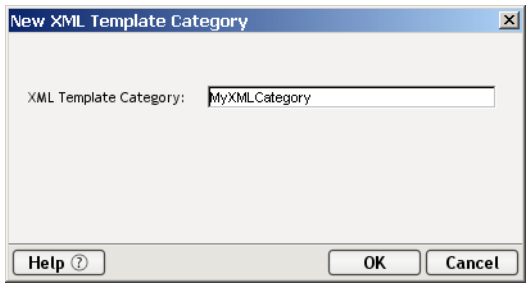


The purpose of the folders is to store your XML Templates, which might contain sample XML documents, schema, and XSL stylesheets.

➢ **To create an XML Template Category:**

1 Select **XML Template Category** in the Composer Category pane.

2 Click the right-mouse button and select **New**.



3 Type a name for the category and click **OK**.

Template Scenarios

XML output from one component is often used as input for the next component in a service. Ease of sample-reuse is, in fact, one of the main benefits of working with XML Templates.

Another benefit is sample organization. It's common for a particular XML Template to hold a *variety* of documents related to a service. For example, a given template might contain four XML files: one to be used as Input to a particular service, one as a Temp document (a sort of scratchpad-doc to hold values that will change throughout the course of your service), one as Output, and one as a Fault document, which can hold values to be used in the case of an error.

NOTE: For more information on Temp and Fault Documents, refer to "Using Temp and Fault Messages with a Component" in the chapter on Components.

Of course, it's entirely acceptable to create individual XML Templates for Input docs and other templates for Output docs. How you organize your documents is up to you.

Creating an XML Template

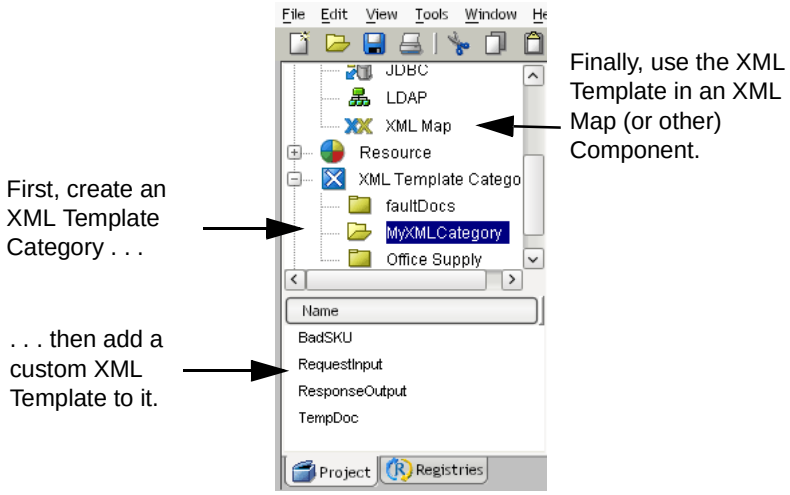
XML Templates can be created in a variety of ways.

The most basic way to create an XML Template (and the way you'd use if you already have the assortment of XML sample docs that you want to use at design time) is to step through the XML Template Wizard. This procedure is outlined below.

You can also start with a schema (.xsd file) and have Composer generate a sample XML document (and associated XML Template) from the schema. This method is described in the section called "Creating XML Templates from Schemas" further below.

A third option is to start with a WSDL file and let Composer generate templates corresponding to the message parts defined in the WSDL. This option is described under "Creating XML Templates from WSDL" further below.

No matter which way you choose, you should begin by making sure a Template Category (a "holder" for your template) exists, as shown in the graphic below. (See the preceding section for information on how to create a Template Category.)



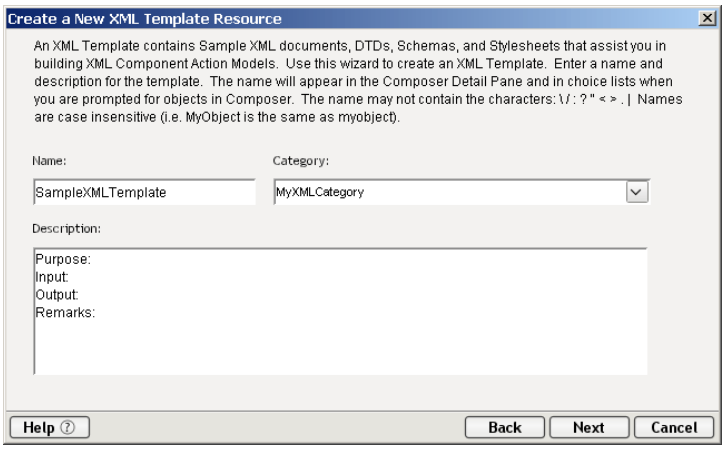
➢ **To create an XML Template using the wizare:**

1 Select an XML Template Category in the Category Pane, then Click the right-mouse button on the category, and select **New**.

or

Use the main menu to select **File > New > xObject,** then select the **Template** tab on the New xObject dialog, and select **XML Template**. (Then click **OK**.)

The **Create a New XML Template** Wizard appears.

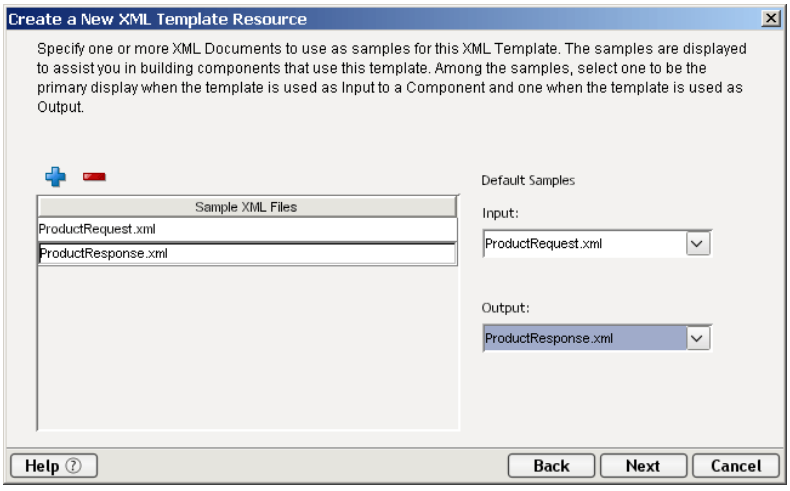


2 Type an arbitrary **Name** for this template.

3 From the pulldown menu under Category, select from among the existing XML Template Categories that you have already created. (See “To create an XML Template Category:” above.)

4 Under **Description**, enter a plain-text description of the intended usage of the template. (Optional.)

5 Click **Next**. The document-selection panel of the wizard appears.



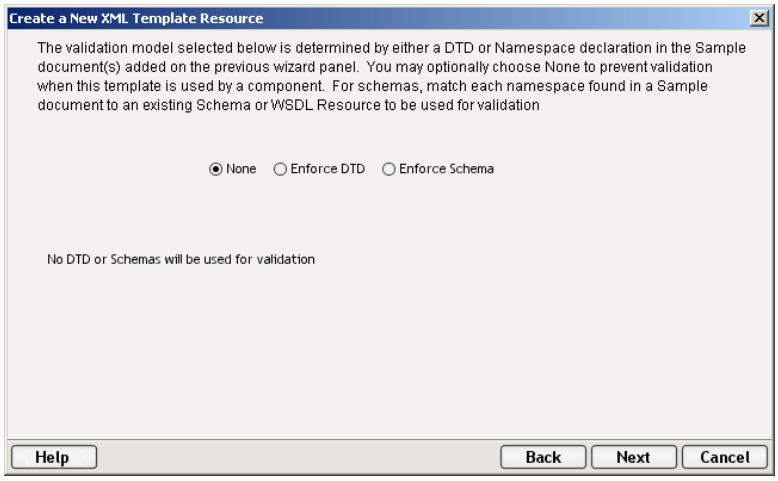
6 Click the blue ‘+’ icon; a file navigation dialog will appear. Use the dialog to specify an XML file on disk that you wish to add to this template. Repeat this step as necessary to add however many XML files you want. You can add files to be used as Temp and Fault documents at this time, in addition to Input and Output Parts. (Click the minus-sign icon to remove a given file from the list.)

NOTE: If you do not specify existing files to be used as your XML samples, an empty default file will be created. You will be able to give this file a name following the last step of the wizard.

7 Under **Default Samples**, below **Input**, use the pulldown menu to select the file you want to see as the default Input Message for any components that use this template. (The pulldown menu will be populated with the names of the files shown in the list you built in the preceding step.)

8 Below **Output**, use the pulldown menu to select the file you want to see as the default Output Message for any components that use this template. (The pulldown menu is populated with the names of the files shown in the list on the left.)

9 Click **Next**. The document validation panel of the wizard appears.



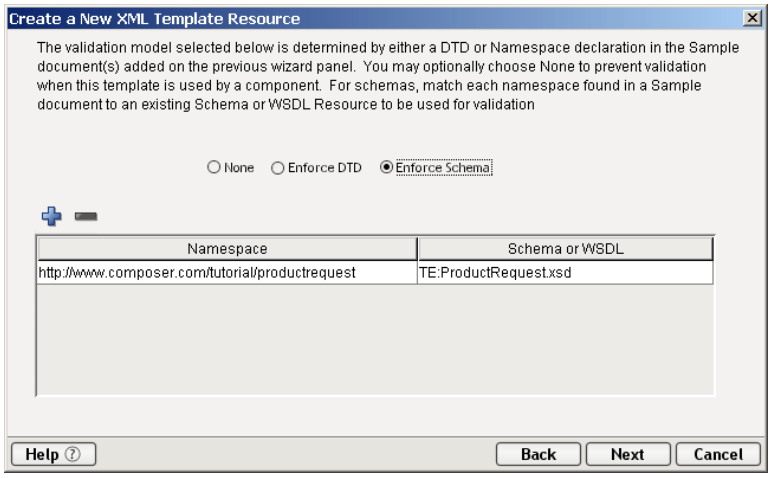
10 To indicate the type of document validation you want to impose on your template documents, click one of the three radio buttons: **None**, **Enforce DTD**, or **Enforce Schema**. The appearance of the dialog will change depending on which button is active. Note that Composer will attempt, based on inspection of the XML template document(s) you specified in the previous dialog, to set the correct radio button for you. You can override Composer's choice at any time. The radio buttons have the following consequences:

None—Choose this option if your application does not require validation of XML documents or if you would like to override the DTD or XSD information specified in your template documents.

Enforce DTD—Documents will be validated against the DTD whose name and/or URI are specified in the text fields shown.

NOTE: If the DTD will be determined dynamically at runtime, you can supply the URI as an ECMAScript expression. If you plan to use a PUBLIC DTD/Schema after deploying the project, you must fill in the PUBLIC Name of DTD field.

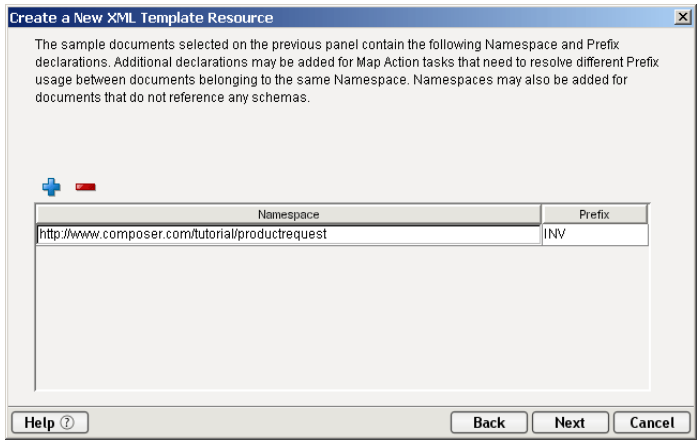
Enforce Schema—Documents will be validated against the XSD or WSDL file indicated. (See illustration below.)



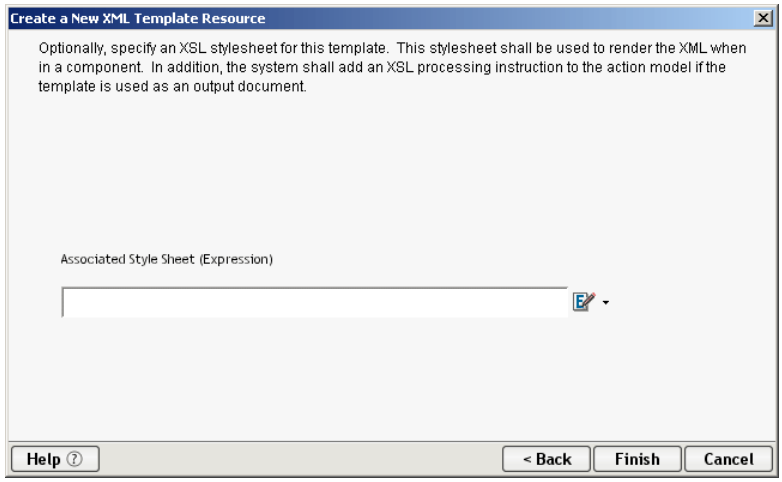
NOTE: Composer will automatically search your sample documents to discover all of the namespaces (if any) declared inside them and the **.xsd** files to which they point. The namespaces and their associated schemas are displayed automatically in the above dialog; in most cases, you will not have to fill in the dialog yourself. If any namespaces are not displayed next to the correct Schema Resource, select the appropriate Schema Resource from the pulldown menu on the right. (That is, use the pulldown menu to associate the correct schema with the correct namespace.)

11 Click **Next** to go to the next panel.

12 If the documents you are using contain namespace information, the namespaces and corresponding prefixes will be summarized in this dialog. If you need to add additional namespace declarations (perhaps for documents that do not reference schemas), use the plus sign (+) icon to do so.



13 Click **Next**. The stylesheet selection pane of the wizard appears.



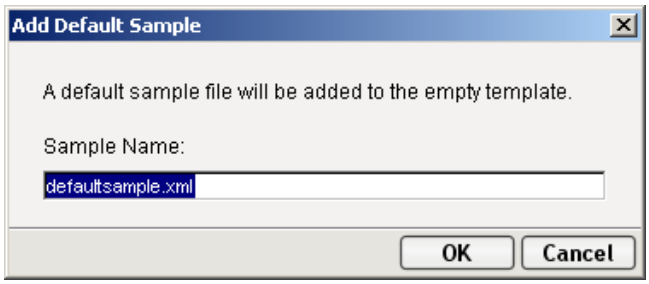
14 (Optional) Specify an XSL StyleSheet to associate with any Service Output that is defined by this XML Template. An XSL Processing Instruction pointing to this stylesheet will be added to the Service Output.

15 If you specified an XSL stylesheet, the following occurs:

- When you create a new service or component, and the template is used for the Output message part, Composer will automatically add a Function action to the new component's Action Model. The Function action adds a *processing instruction* to the Output XML document, specifying the XSL stylesheet for the document.
- The stylesheet referenced in the processing instruction is the one you specified in this XML Template.

16 Click **Finish** to create the XML Template.

NOTE: If you did not use the + sign to add pre-existing files to your template because you wished to create an empty one, at this point, the following dialog window will appear, allowing you to type in a name for your default sample:



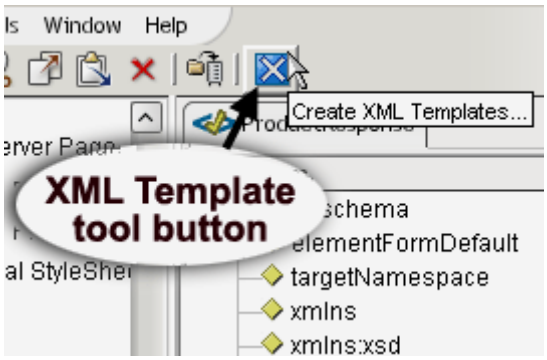
Creating XML Templates from Schemas

Composer can generate a sample XML document and an XML Template resource from a schema (.xsd) document. This is useful in cases where, for example, a business partner may have supplied you with a schema that must be used, but you lack actual sample docs based on that schema.

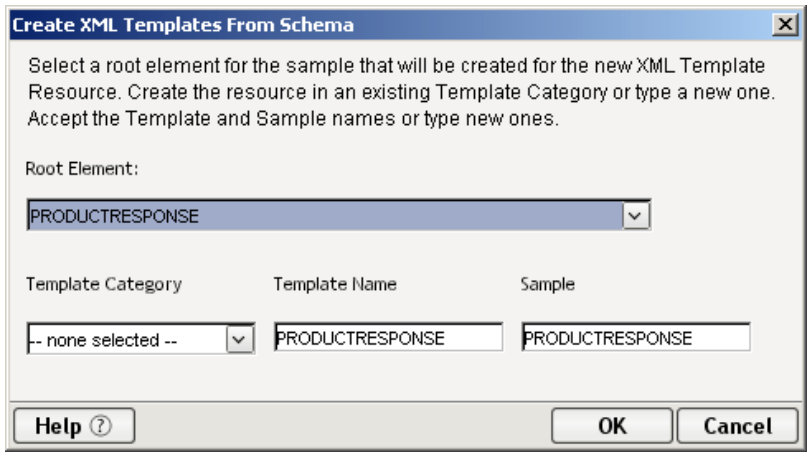
NOTE: The XML "stub document" (sample doc) that Composer generates from a schema will initially have no data values, which means it may not validate against the schema until you edit it to add data values.

➢ **To create an XML Template from a Schema:**

1. If you have not already done so, create an XSD Resource based on the schema file you are using. (See "About XSD Resources" in the chapter on Resources. The XSD Resource wizard can be accessed via **File > New > xObject** and the **Resource** tab of the New xObject dialog.)
2. Open the XSD Resource you intend to use. (You can right-click on it in the explorer instance pane, or use the **File > Open** command in the main menu.)
3. When an XSD Resource is open, a blue Template icon will appear in the main toolbar in Composer:



Click the XML Template tool button. A dialog appears:



4 Choose a Template Category in which the new template should be created, using the **Template Category** pulldown menu.

5 Verify that the **Root Element**, **Template Name**, and **Sample** controls (which are prepopulated with values derived from the schema in question) contain appropriate values. Edit any values as needed.

6 Click **OK**. A new XML Template is added to the Template Category.

7 Edit the newly created sample document as needed (to add data values).

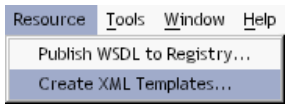
8 **Save** your work.

Creating XML Templates from WSDL

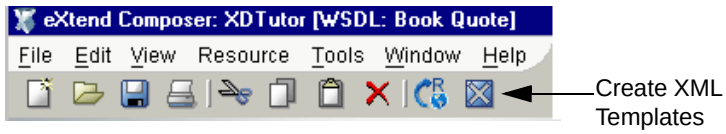
When an external WSDL is downloaded into exteNd Composer, you generally need to have XML templates corresponding to message parts in the WSDL in order to create working components. XML samples must be created which can be validated against the WSDL. (These templates can then be used in components to create actions then used in the WS interchange.) Composer can help with this. If you have a WSDL Resource for a service, simply open the WSDL Resource: Composer's toolbar and menus will change as shown below, and you will be able to create XML stub documents (template docs) at the click of a button.

There are two ways to generate XML Template docs from an open WSDL Resource:

- from the **Resource** menu, select **Create XML Template**



- OR: Click the **Create XML Templates button** on the toolbar.

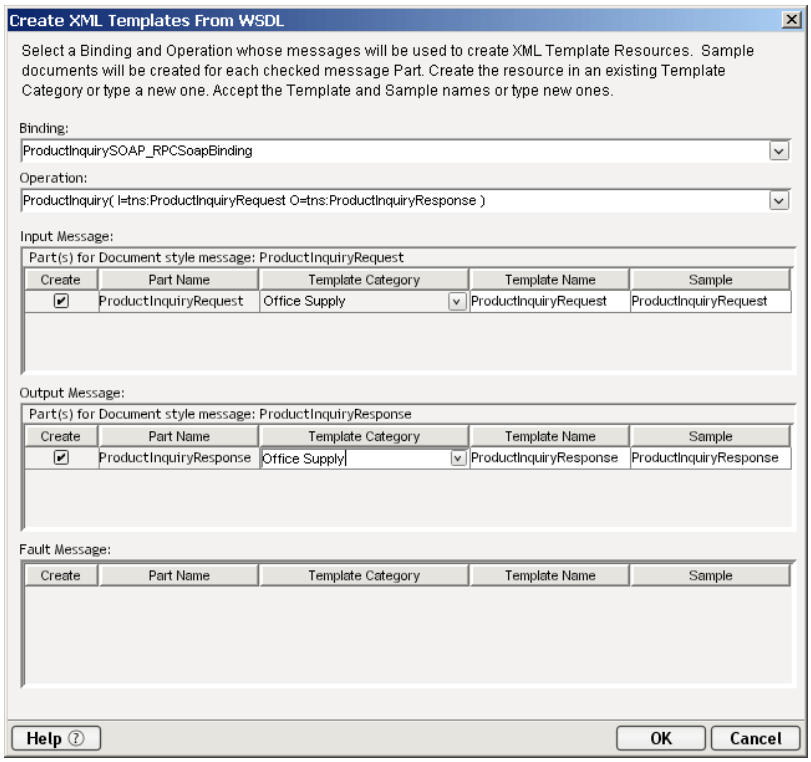


NOTE: The samples created will not contain element data (and as a result, may cause validation errors until you enter dummy values of the appropriate types). You may need to populate various elements with sample data for test purposes. Note also that when elements refer to **##any** or **##other** namespaces, the samples are incomplete and you have to manually complete them.

Template documents can be create in this fashion for document-style as well as RPC bindings.

➢ **To create XML Templates from WSDL:**

1 From the main menu, click on **Resources>Create XML Templates,** or click on the button on the toolbar. A dialog will appear.



2 Select a **Service/Port or Binding** from the dropdown list as a source for creating the XML Template.

3 Select an **Operation** from the dropdown list as a source for creating the XML Template.

4 The bottom portion of the dialog box is divided into Input, Output and Fault **Messages**. Follow the same procedure for each Part:

- Check the box below **Create** if you will be creating the new template from WSDL.
- Select a **Template Category** from the dropdown list. New Categories can be created.
- Type in a **Template Name**.
- The name listed under **Sample** defaults from the Part name. Enter a new name if the default name is not the sample name you want.

5 Click **OK** to finish.

Importing an XML Template

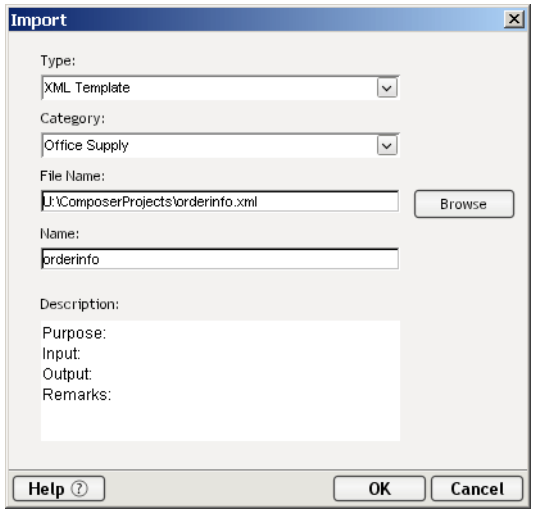
If you've already created an XML Template for another project, you can import it into the current project.

➢ **To import an XML Template:**

1 In Composer's Category Pane, select the **XML Template Category** to which you want to associate the template instance.

2 Right-click and select **Import** from the context menu. A dialog appears.

NOTE: If the Import command is not highlighted, it's because you have chosen a Template Category that belongs to a subproject. This operation is not allowed. If you need to import template docs into a subproject, close the current project. Open the subproject on its own, add templates to it, save it, and close it; then return to the project you were working on originally.



3 Select **XML Template** as the **Type**.

4 Select an XML Template **Category** from the drop down list.

5 Select the **File Name** location using **Browse**. You may also read in a file from a URL by explicitly preceding your filename with "http://," "https://" or "ftp."

6 Type in a **Name**.

7 Optionally supply a **Description**.

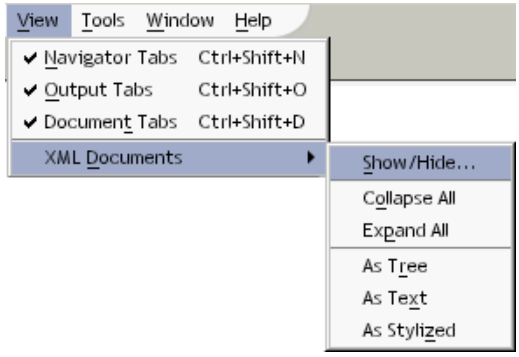
8 Click **OK**.

Showing and Hiding XML Documents

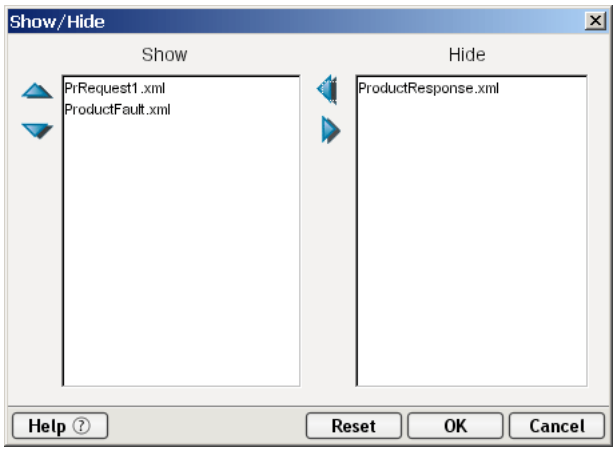
It can be convenient to toggle the visibility of XML document views when working in Composer's main window.

➢ **To toggle XML document visibility:**

1 From Composer's main menubar, choose: **View>XML Documents>Show/Hide**.



A dialog will appear:



2 The **Show/Hide** dialog displays the names of the XML documents associated with the open template or component.

NOTE: In a component, the *Input* and *Output* XML documents default to the **Show** column. Message parts created as a result of a component action default to hidden.

3 In the **Hide** column, select any XML documents you want to be *displayed* and click the **left arrow** button. Conversely, in the **Show** column, select any XML documents you want to be *hidden* and click the **right arrow** button.

4 Select the XML document you want to display as the *top document* and click the **up arrow** button until the document is displayed as the uppermost document in the **Show** column. Conversely, use the **down arrow** to move the document down further in the list.

5 Continue to select XML documents in the **Show** column and use the up- and down-triangle buttons to move the XML documents into the desired order until they are displayed the way you like.

6 Click **OK**. The dialog closes and the Component editor's data panes are rearranged accordingly.

XML Template Editor

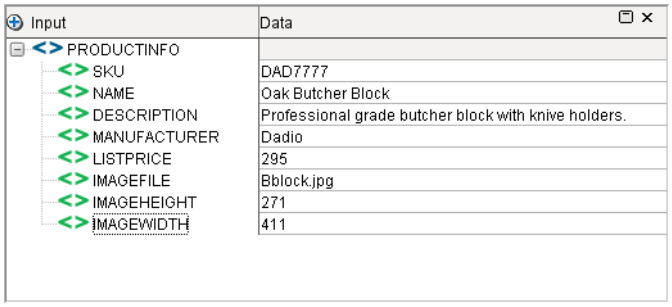
The XML Template Editor allows you to edit the template in Composer, rather than using an external editor.

Viewing the documents in the Template Editor and Context Menus

The View option from the main menubar allows you to select the way you want the XML information displayed in the Component Editor. You can choose from tree, text or stylized. Each view has its own unique context menu accessed by the RMB.

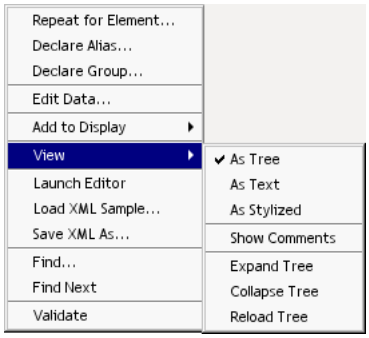
Tree View and Context menu options

The default view displays the message part as a tree, as shown below.



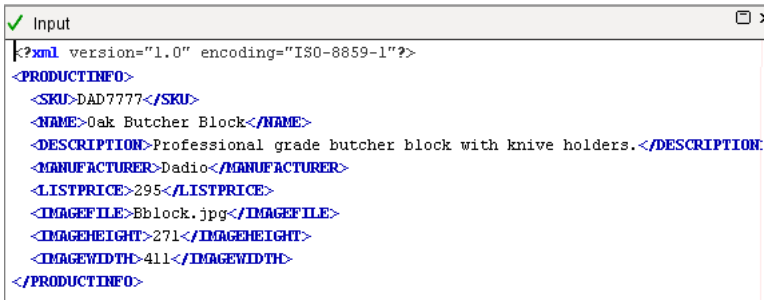
This view allows you to edit element and attribute *values* (that is, document data) but not the XML structure.

The Context menu commands accessible via the right mouse button are shown below.



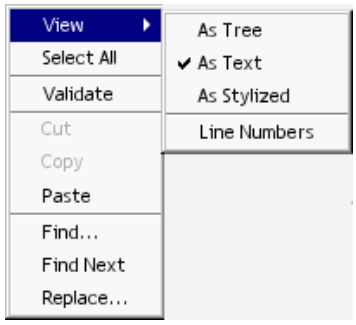
Text View and Context menu options

In Text View, you can see and edit the complete XML file, including structural elements.



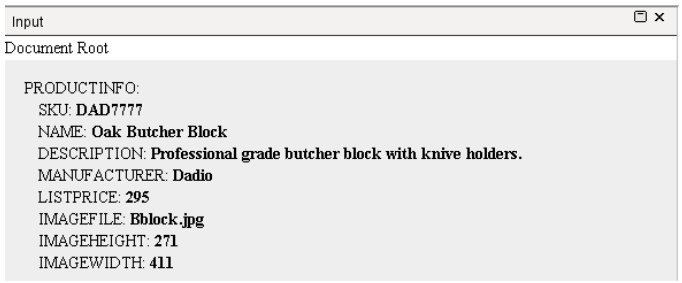
Text view offers a convenient way to inspect non-content-model portions of the Input, Temp or Output Parts, such as comments, processing instructions, DOCTYPE declarations, and so forth.

The Context menu options accessed by RMB as shown below.



Stylized View and Context menu options

When the Stylized view is selected pane, your view of the message part contents looks like this:

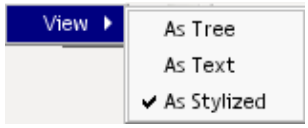


This view gives a “report” style overview of the XML contents so that you can see at a glance what the content is for all attributes and elements. This view uses the following algorithm to render XML.

If there is an associated stylesheet with this document component, evaluate the expression and use that one.

If this fails, use the default stylesheet: **com/sssw/b2b/dt/default.xsl**

To change to a stylized view, click the RMB to access the Context menu as shown below.



Working with an XML Template

Each XML Template you create resides in an XML Template category. To view the name and creation date of the XML Template, select an XML Template category. All XML Templates for the category are listed in the Detail pane of Composer. Each template has a context menu, giving you ways to work with the template.

Viewing an XML Document

Each XML Template contains one or more sample documents. You can open a sample document in an XML editor (a separate application external to Composer).

➢ **To view a sample document in your XML editor:**

1 Click the right-mouse button on an XML Template.

2 Select **Edit Sample** and select the sample document you wish to edit. Whatever XML editor you identified during your Composer installation will open (by default, Internet Explorer is used).

Editing an XML Template

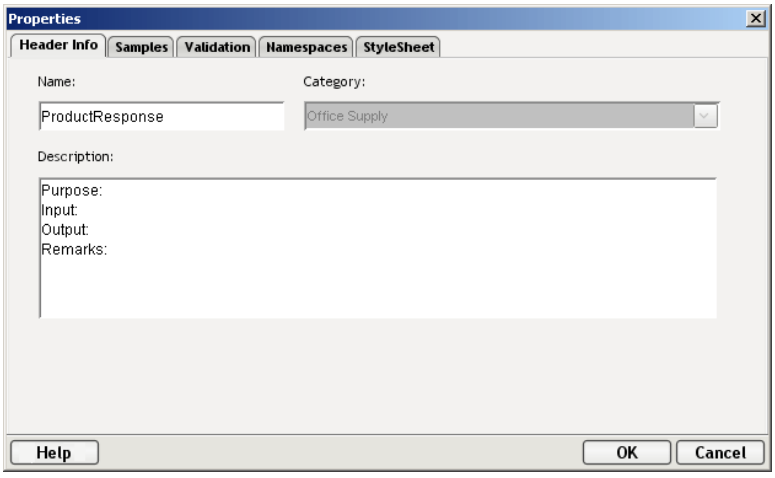
You can modify the XML Template by adding and deleting sample documents, schema, and XSL stylesheets.

➢ **To edit an XML Template:**

- Doubleclick the XML Template instance to open it in the Content Editor. Once a sample file is open, right-click the mouse button to display a contextual menu which gives you several options (see "The XML Template Editor Context Menu" below) including **Edit Data**.

  or

- Single-click the XML Template in the Instance Pane, then click the right-mouse button and select **Properties** from the context menu.

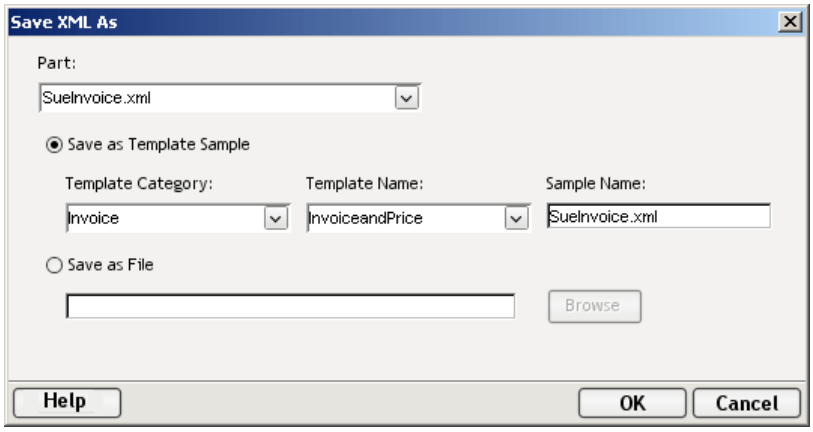


NOTE: Changing the name of the template on the Properties page causes a **Save As** operation, preserving the original and creating a duplicate with a new name. To change just the name of an XML Template, use the **Rename** option on the context menu.

Saving Changes to XML Documents

Once you have made changes to your XML using the methods described above, you will, of course, want to save them. There are four ways to save sample XML documents i n Composer.

- Select **File>Save** from the main menubar.
- Select **File>Save As** from the main menubar. This brings up a tabbed dialog window resembling the Properties screen shown above, allowing you to type over the current name of the document with a new name.
- Select **File>Save All** to save changes to all the XML documents you currently have open in the Content Editor.
- Within the Content Editor, right-click on an open document and select **Save XML As** from the context menu (see "The XML Template Editor Context Menu" below). This brings up the Save XML As Dialog window shown below:



➢ **To use the Save XML As dialog:**

1 Select a "Part", or XML document name from the drop-down list.

2 If you want to save the open document as a sample, then choose **Save as Template Sample**.
   - Select a Template Category
   - Select a Template Name
   - Type in a Sample Name

3 If you want to save the open document as a file, then choose **Save as File**.
   - Click on Browse to select a directory in which to store your file and give it a name.

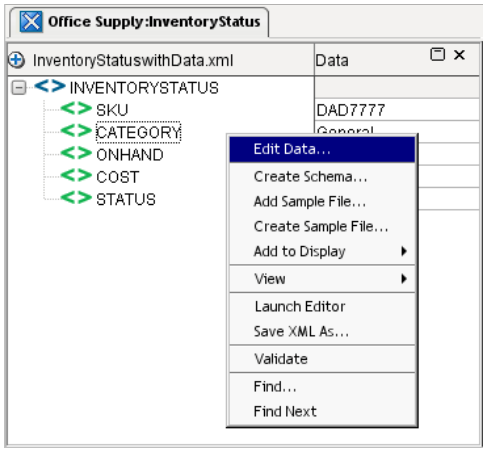
4 Click on **OK** to close the window and Save the XML.

Printing an XML Document

To print the XML document, Select **File>Print** from the main menubar. The document component is formatted according to the template.

The XML Template Editor Context Menu

When you open an XML sample file in the Content Editor and right-click on it, a menu appears allowing you to perform several functions.



These functions are explained in the table below:

Edit Data	Allows you to edit element and attribute *values* (that is, document data)
Create Schema	Brings up a dialog allowing you to create a new schema resource
Add Sample File	Brings up a dialog with a file directory so you can select a pre-existing XML file to add to the template
Create Sample File	Brings up a dialog which allows you to type in a name for a new sample XML file.
Add to Display	Allows you to display additional XML files which are part of the current template but are not currently open in the editor pane
View	Change the view of the document (see “Viewing an XML Document” above)
Launch Editor	Opens the default XML editor you specified during installation
Save XML As	Opens the Save As dialog window, which allows you to specify a part name, save the file as a sample or save as a file
Validate	Runs a validation routine to check that your XML is sound
Find	Opens the Find dialog allowing you to search for strings in the XML data or structure
Find Next	Repeats previous search

Deleting an XML Template

When you create an XML Template, Composer makes copies of the original XML, document definition and XSL files and places them into an “Imports” directory under the proper XML category. When you delete an XML Template, you are going to delete the copies, not the original files. To delete an XML Template, highlight it in the Detail Pane of the Navigator, right-mouse click it, and select **Delete**. The file must be closed in order to delete it.

Moving an XML Template to a Different Category

➢ **To move an XML Template from one category to another:**

1. Select the template you wish to move.
2. Click the right-mouse button and select either **Cut** or **Copy**.
3. In the Category pane of Composer, click on another XML category.
4. In the Details pane, click the right mouse button and select **Paste**.

Renaming an XML Template

➢ **To rename an XML Template:**

1. Select the template you wish to rename.
2. Click the right mouse button and select **Rename**.
3. Type the new name.
4. Click **OK**.

NOTE: Be sure to rename a template using the above procedure. If you change the name of the template on its **Properties** page, it causes a **Save As** operation, preserving the original and creating a duplicate with a new name.

Understanding Where XML Templates Are Stored on Your Hard Drive

XML Templates are stored as part of a project. For information on where project files are stored, see "Understanding Where Project Files are Stored" on page 69.

NOTE: Copies of the samples, definitions and XML stylesheets used in the template are stored in a folder. The original documents are not modified.

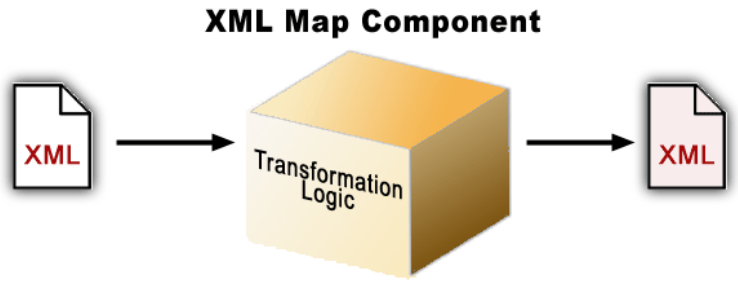
6 Creating an XML Map Component

In many ways, the Composer XML Map Component is the simplest yet most important of Composer's component types. You will use it to perform XML transformations of input documents to output documents. It is essential that you understand how XML Map Components (and related resources) work if you are to build useful Composer services.

This chapter introduces you to XML Map Components and describes how they work within an exteNd Composer service. After reading this chapter, you will understand what comprises an XML Map component, what it can do, and how to design, create, and use an XML Map component.

What is an XML Map Component?

An XML Map component is an object that accepts one or more XML documents as inputs, uses a collection of actions to operate on these inputs and returns an XML document as output.

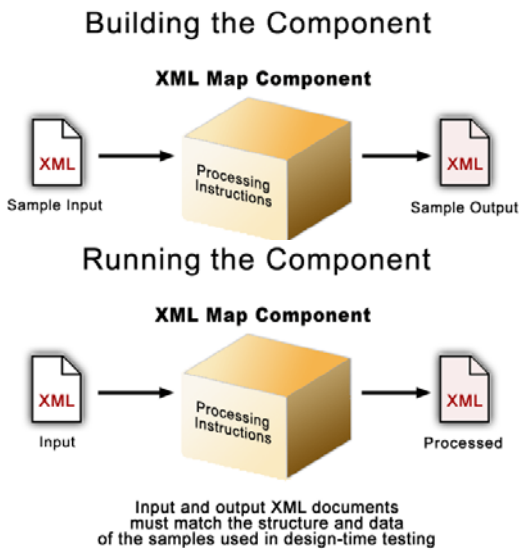


An XML Map component can perform simple data manipulation, such as mapping and transferring data from one XML document to another. It can also carry out sophisticated manipulations, such as transforming both the data and structure of a document. You can even create XML Map components that process XSL, send mail, and post and receive XML documents using the HTTP protocol.

The concept behind a component is to pass one or more XML documents in as inputs, process these inputs, and return one output XML document. The output XML document is then used as input for other components or returned as the final output of a service. In this way, you can create components that work together in a service to carry out complete business-to-business solutions.

Using XML Template Sample Documents to Build an XML Map Component

The XML documents you use when building an XML Map component are samples of the actual documents that will be processed in a running application. Sample documents are added to Composer in XML template objects. You use samples of documents whose structure and data representation are identical to the documents that will be processed. The illustration below shows the difference between documents you use for building purposes and those that are actually processed by the component.



The samples used to build a component are not actually used, or even referred to by name, at runtime. They are simply templates that represent the structure and data to be manipulated. The samples are temporary aids that help you construct processing actions that perform the correct runtime manipulations. What exteNd actually uses at runtime to process XML data is an object representation of the XML document called a DOM.

What is a DOM?

In the XML Map component editor, a sample document is represented in a format recommended by the W3C known as the document object model (DOM). A DOM is an XML document constructed as an object in a software program's memory. It provides standard methods for manipulating the object. Using DOMs, Composer lets you build XML documents, navigate within their structure, and add, modify or delete elements and content. Anything found within an XML document can be manipulated using a DOM method. Composer supports all DOM methods recommended by the W3C ECMA to DOM Binding Specification (See **http://www.w3.org/**).

NOTE: In some dialogs, Composer refers to DOMs as Messages.

Understanding DOM Structure

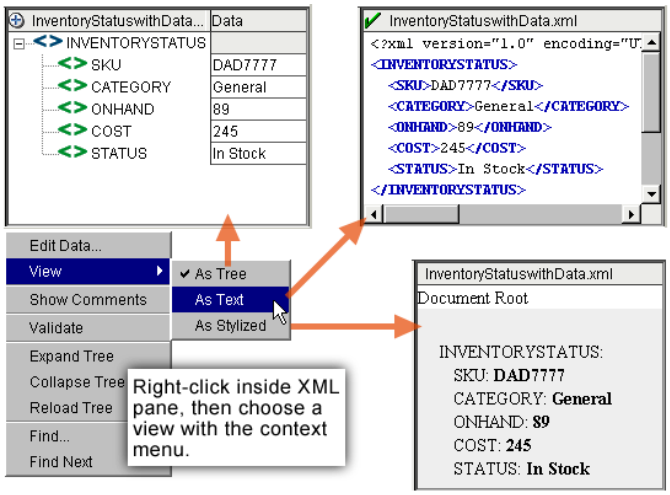
When the XML Map component editor is active, every sample document is converted to a DOM.

DOMs are used because:

- A DOM uses a standard way of naming and organizing XML structural elements so that the elements can be selected easily and clearly
- A structural element of a DOM can be operated on
- A DOM is the structure that is created and manipulated at runtime

A DOM is organized hierarchically, which means it forms a tree structure. To understand how a particular DOM is structured, it's often useful to be able to view the DOM from different perspectives. For example, sometimes it's helpful to see the raw text of the XML document underlying the DOM. Other times, you may be interested in seeing a summary view of the data (rather than the element and attribute names) in a document. Composer allows you to change views as necessary in order to switch between tree, text, and summary presentations.

To change views, simply right-click inside any DOM window, then select the view you want to see from the View submenu. The three available view types are shown below.



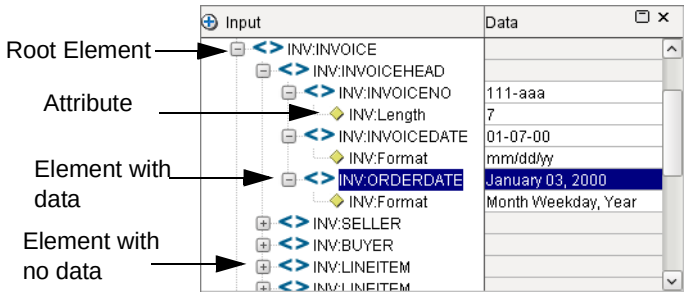
Elements in a DOM are defined by tags in an XML document. For instance, in the above example, there is an element tag in the XML document named <INVENTORYSTATUS>. The tag has an end tag (</INVENTORYSTATUS). All structural elements within <INVENTORYSTATUS> and </INVENTORYSTATUS>, such as <SKU>, are represented at a lower (or in DOM terminology, *descendant*) level in the DOM tree.

All element names are *case-sensitive*, meaning that <INVOICENO> is not the same as <InvoiceNo>.

An element in a DOM tree is referred to as a *node*. A collection of nodes is represented in a hierarchy and is referred to by the following naming conventions:

Node Type	Description
Root	The topmost element in a DOM tree from which all other elements are descendants. Only one is allowed.
Descendant	Any node that is below (contained within) another node
Child	The immediate descendant of a node
Sibling	All nodes that share the same parent node
Ancestor	Any node that is above (contains) another node
Parent	The immediate ancestor of a node
Leaf	Any node without a descendant

Each element type in a DOM has its own icon, as identified below.



Using DOMs at Runtime

It is through DOMs that components pass and return data to one another in a running application. At runtime, when a component is executed, it is passed a DOM. The passed DOM becomes the Input Part to be operated on. As each of the component's mapping actions executes, the Output Part is created, element by element.

DOM Behaviors during Runtime

The first time you open a component, the original samples are loaded into the Input and Ouput DOMs. When you begin animation, the Input Part remains and the Temp and Ouput DOMs are cleared from any data originally contained in them. At the end of execution, data appears in all DOMs.

Creating Different Types of Messages

When you create an XML Map component, you select input and output XML templates for it. However, within the Component Editor you can also:

- Create an Output Part without using a template, as described in "Creating an Output Document without Using a Template" on page 113
- Create a Temporary Message Part, as described in "Creating a Temporary Message Part" on page 115
- Create a Fault Message Part, as described in "Creating a Fault Message Part" on page 116
- Dynamically create a DOM from an external XML document using the XML Interchange action.

Creating an XML Map Component

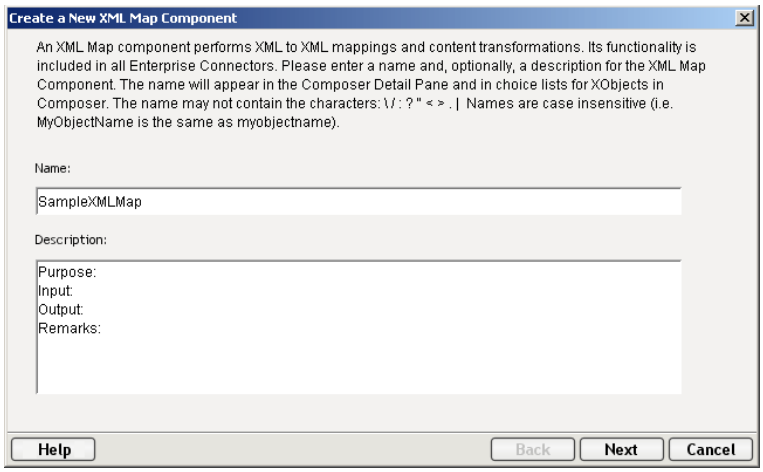
The first step in creating an XML Map component is to specify the XML templates for the component. For more information, see "Creating an XML Template" on page 81.

Once you've specified the XML templates, you can create your component, using the template's sample documents to represent the inputs and output processed by your component.

NOTE: Various other component types (such as the JDBC Component, JMS Component, etc.) are covered in detail in the appropriate Enterprise Connect product user guides. The same basic principles are used in the creation and editing of all components, however. Also, the various Basic Actions (see next chapter) available in the XML Map Component are also available in all other Composer component types.

➢ **To create an XML Map component:**

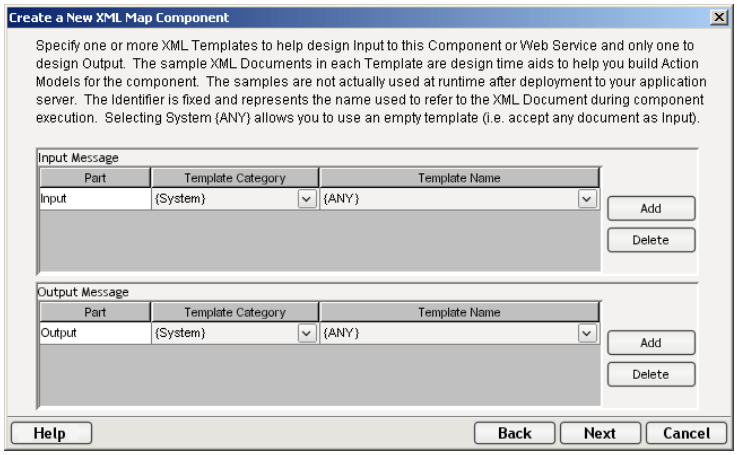
1 From the Composer File menu, select **New** then **xObject**. Select the **Component** tab and then **XML Map**. The New xObject dialog box appears.



2 Type a **Name** for the component.

3 Optionally, type **Description** information.

4 Click **Next**. A new panel appears as follows.



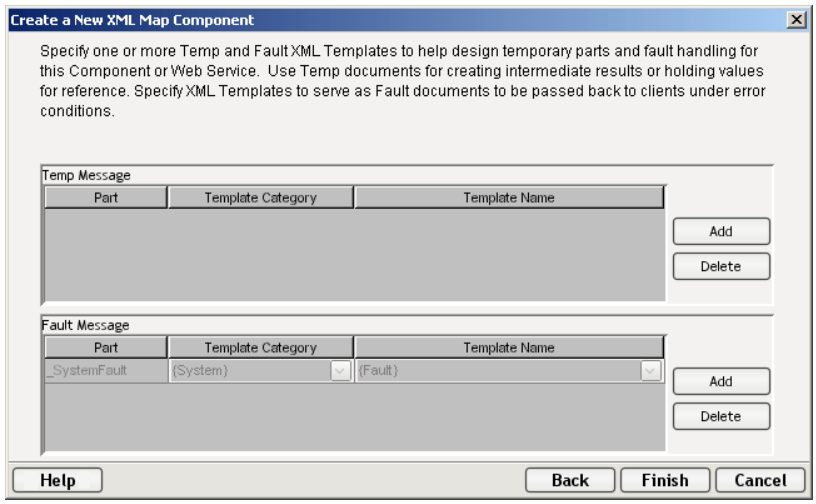
5 Specify the Input and Output templates (also called Messages).

- Type in a name for the template under **Part** if you wish the name to appear in the Component Editor as something other than “Input” or “Output.”
- Select a **Template Category** if it is different than the default category.
- Select a **Template Name** from the list of XML templates in the selected **Template Category**.
- To add additional input XML templates, click **Add** and repeat steps 2 through 4.
- To remove an input XML template, select an entry and click **Delete**.

6 Select an XML template as an output using the same methods described in the previous step.

NOTE: You can specify an output XML template that contains no structure by selecting {ANY} as the Output template. For more information, see “Creating an Output Document without Using a Template” on page 113.

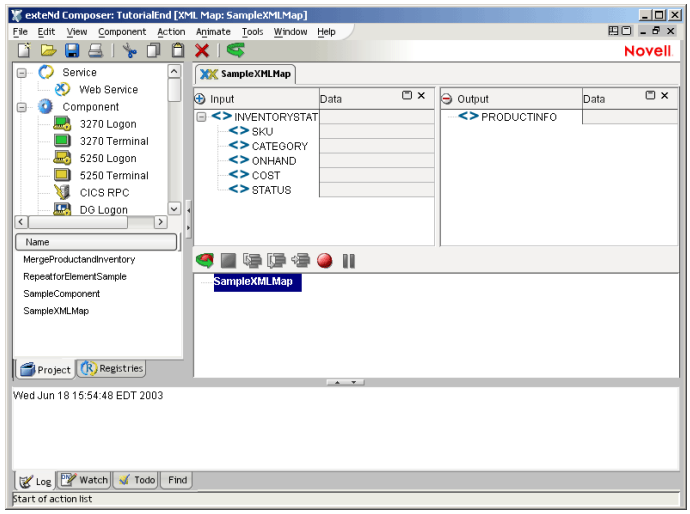
7 Click **Next** to go the Temp and Fault XML template dialog. If desired, specify a template to be used as a scratchpad under the “Temp Message” pane of the dialog window. This can be useful if you need a place to hold values that will only be used temporarily during the execution of your component or are for reference only. Under the “Fault Message” pane, select an XML template to be used to pass back to clients when a fault condition occurs.



8 As above, to add additional XML templates, click **Add** and choose a Part Name, a Template Category and Template Name for each. Repeat as many times as desired. To *remove* an input XML template, select an entry and click **Delete**. Temp and Fault Message Parts are discussed in more detail below, beginning on page 115.

9 Click **Next**. For several of the component types, the Connection Info panel will appear, allowing you to select a previously created Connection Resource.

10 Click **Finish**. The component is created and the XML Map Component Editor appears.



Namespaces and Output Parts

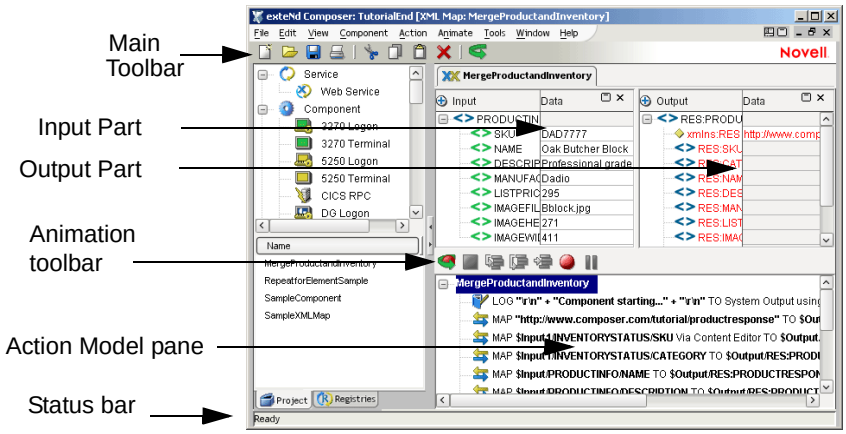
You should be aware that when a new XML Map Component is created whose Output template uses namespaces, a new *Map action* will automatically be placed into the action list, mapping the namespace URI to an attribute in the Output Part. It's important not only that you not *delete* this Map action, but that you avoid accidentally overwriting it. An overwrite can happen if you place a Component action *downstream* of the namespace-URI Map action, and the Component action returns its results to the Output Part. The solution in this case is to move (by Cut and Paste) the original Map action to a spot downstream of the Component action.

Understanding the XML Map Component Editor

The XML Map Component Editor is where you specify the mapping, transformation, and transfer of all input and output structure and data.

The XML Map Component Editor provides a logical working environment for the inputs, output, and actions of your component. The XML Map component editor is composed of multiple Mapping panes and a single Action Model pane. The Mapping panes display the XML for your sample Input and Output message parts. The Action Model pane displays actions that operate on the Mapping panes.

The following illustration shows the XML Map Component Editor with its menu and toolbar, one input Part, one output Part, and several actions in the Action Model pane.



About the Menu and Toolbar

The main menu in Composer and it's sub-menus and toolbar options change according to the type of component you currently have open in the Component Editor. The following component-specific options occur on the main menubar when an XML Map Component is open.

XMLMap Component Menu	Options
File	**You can create, open and delete any type of component from the File menu, just as you can from the Composer File menu.** Some menu choices have implications that are particularly significant for XML Map Components: **Save** saves the inputs, output and actions in the component **Save As...** saves the component under a new name and allows you to change the inputs, output, and actions. See "Saving Your Component" on page 120. **Save All...** saves all components currently open **Save XML As . . .** allows you to save the structure of the message Part into an XML document. See "Saving a DOM as an XML Document" on page 121. **Load XML Sample . . .** allows you to load other sample documents from a template into a message part for testing the component. See "Loading a Sample Document" on page 119. **Properties** lets you view the component's templates and other information. See "Viewing Component Properties" on page 124. **Print** lets you print your component

XMLMap Component Menu	Options
Edit	You can undo or redo an action, cut, copy, and paste text anywhere in the component editor. In addition, you can do the following: **Find** finds an element in a mapping pane or text in the Action Model. See “Finding a Document Element” on page 110. **Find Next** finds the next element in a mapping pane or text in the Action Model. See “Finding the Next Document Element” on page 111. **Replace** replaces selected text in the Action Model only. See “Replacing Text in the Action Model” on page 112.
View	**Navigator Tabs** toggles the visibility of the nav frame at the left of the Composer main window. Use this command to hide or unhide the whole nav frame. **Output Tabs** toggles the visibility of the Message frame at the bottom of the Composer main window. Use this command to hide or unhide the whole Message frame. Document Tabs toggles the visibility of the XML Document Tabs in the Component Editor. XML Documents brings up a submenu with the following choices: ◆ **Show/Hide** allows you to change the order and visibility of all message parts associated with the open component (see “To set the visibilities of XML documents:” below) ◆ **Collapse All** hides all XML nodes except for the root node in all mapping panes. ◆ **Expand All** displays all XML nodes in all mapping panes. ◆ (as **Tree**, as **Text**, as **Stylized**) displays controls how XML contents are displayed in the Component Editor. **Window Layout** allows you to specify which DOMs to display and how to arrange the DOM and Action Model panes within the component window. See “Using Window Layout and Show/Hide in the Component Editor” on page 104.
Component	**Execute—**Runs the component from start to finish, for testing purposes. **Reload XML Documents** reloads the original samples from the Input and Output templates, clearing whatever is currently shown in the XML message panes. See “Reloading an XML Document” on page 118. **Add Watch** allows you to identify certain data items and examine their data values during the execution of a component as a debugging aid.
Action	**New Action** contains all actions that you can add to the Action Model. **Edit** allows you to edit a selected action **Disable** allows you to disable a selected action
Animate	This menu contains all processing animation tools that you can use to test the component. The Toolbar contains buttons that allow you to run animation tools. For more information, see “Testing and Debugging” on page 295.

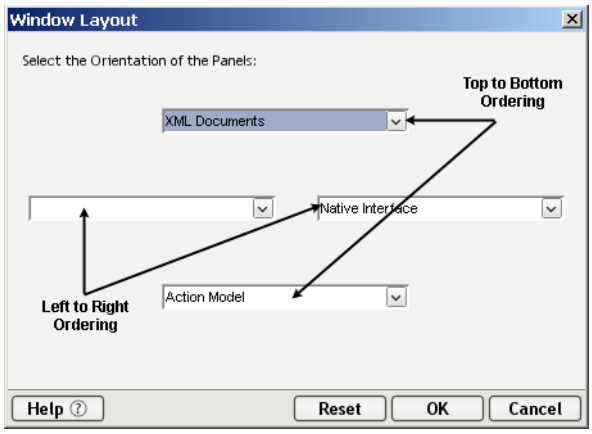
Many of the menu items are on the toolbar. Rest your mouse on a toolbar item for a brief description of it.

Using Window Layout and Show/Hide in the Component Editor

The panes of the component editor can be displayed, hidden, repositioned and resized, making it easier for you to work with their contents. Use the Window Layout option and the Show/Hide option from the View Menu.

➢ **To arrange the panes of the component editor:**

1 Select **Window Layout** from the View menu. The Window Layout dialog appears and allows you to adjust the placement of the panels in the Window.



2 Select the orientation of the XML documents and Action Model as follows:

- If you want the panes to be top-to-bottom, select either XML Document or Action Model from the upper-most pull-down menu. If you selected XML Document for the upper pane, select Action Model from the lower-most pull-down menu. If you selected Action Model for the upper pane, select XML Document for the lower.
- If you want the panes to be left-to-right, select either XML Document or Action Model from the left-most pull-down menu. If you selected XML Document for the left pane, select Action Model from the right-most pull-down menu. If you selected Action Model for the left pane, select XML Document for the right.

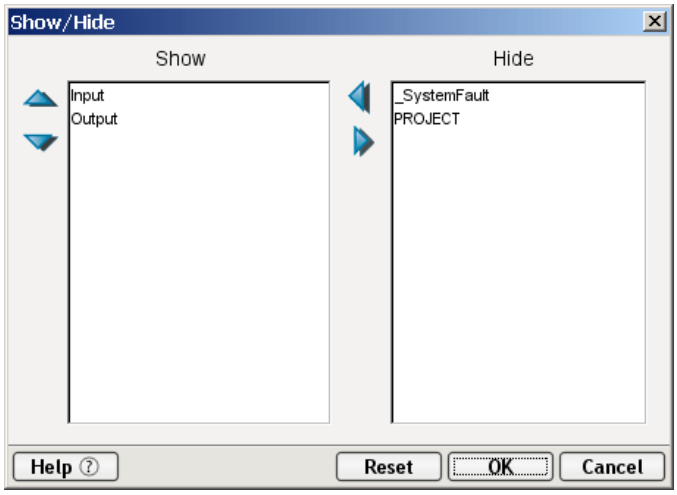
3 Click **OK**.

4 If you're not satisfied with the result, select **View** then **Window Layout**, and then click the **Reset** button. The panes revert to the default setting.

➢ **To set the visibilities of XML documents:**

1 Select **View/XML Documents>Show/Hide**. The Show/Hide dialog displays the visibility status of XML documents.

NOTE: The Input and Output XML documents default to the **Show** column. DOMs created as a result of a component action default to **Hide**.



2 Select any XML documents you want to be displayed from the **Hide** column and click the **left-triangle** button. Conversely, select any XML Documents that you don't want to display from the **Show** column and click the **right-triangle** button.

3 Select the XML document you want to display as the top document and click the **up-triangle** button until the document is displayed at the level you want in the **Show** column.

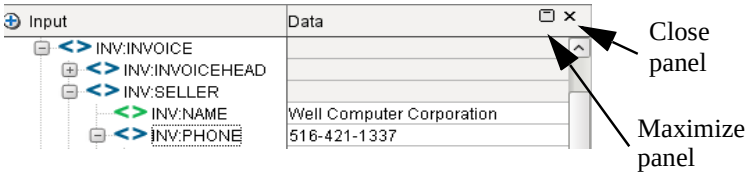
NOTE: If you selected a left-to-right orientation in the preceding procedure ("To arrange the panes of the component editor:", above), the order of your XML documents appears from left-to-right: Higher-precedence documents will appear on the left.

4 Continue to select XML documents in the **Show** column and use the up and down buttons as necessary until the XML documents are in the desired order.

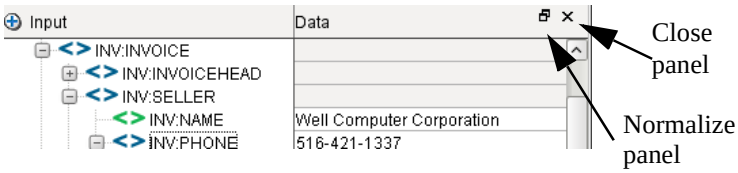
5 Click **OK**. The Window Layout dialog closes and the Component editor is rearranged accordingly.

Managing Document Panes from within the XML Panel

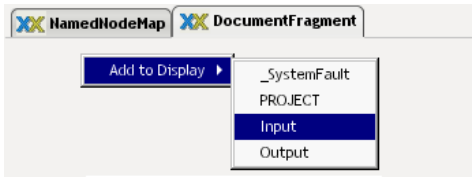
Individual XML document panels can be maximized, normalized or closed using icons located within the header area of the document panel:



Clicking the Maximize icon will cause the document to take over the entire XML Panel area, temporarily hiding any other open message Parts. Once a document has been maximized, the icon will change to the Normalize icon, so you can restore the document to its previous size.



Clicking the Close icon will hide the document from view. Closing all XML documents leaves the XML Panel open but empty. To re-open a document when the editor panel is in this state, click with the RMB. A context menu is displayed, from which you can select a document to open.



About the Mapping Panes

The default XML Map component editor has the following window pane configuration:

- One or more Input Mapping panes (each displaying a representation of the XML for one of the samples in their respective XML Template), one Output Mapping pane
- One Action Model pane
- Temporary message part mapping panes, or XML returned as a result of executing another component via a component action

Note that there is a color coding aspect of the mapping pane. Red indicates the first direct mapping of an element, so that if you wanted to look at what elements in the output tree have been mapped, you can identify that by the color red. Green indicates the first occurrence of the element that has a repeat alias defined for it.

NOTE: You must use Window Layout discussed above to display a dynamically created message part, such as mapping panes created dynamically by the XML Interchange action.

About the Input Mapping Pane

The Input Mapping pane displays one sample document from the input XML template as a document object in the Component Editor. (If your component contains multiple templates, each input message is displayed in its own pane.) The panes can be sized by dragging borders up, down, left, or right.

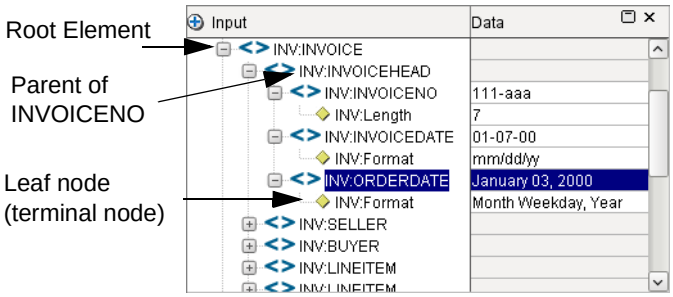
If your component contains multiple input XML templates, they are displayed with the following names:

Display Order	Document Name
First template	Input
Second template	Input1
Additional templates	Input*n*: (n = the template order minus one. For instance, the the last input template is the fifth input template, the DOM name will be Input4.)

The template display order is determined by the order you specified when selecting XML templates at the time of the component creation. You can change this order using the up and down arrows using **View**>**XML Documents**>**Show/Hide**, as indicated above.

About DOM Elements and Data Values

Each input pane consists of two areas: the DOM tree and the Data values. The following illustration shows an Input Part with several elements and data for those elements.



Notice that when you select an XML element, the element name appears in the status bar showing the fully qualified element name. Any data associated with the element (from the XML document) appears in the Data area. Although the data is not used when the component processes at runtime, the data is helpful for setting up and testing the component. You can leave the data in the DOM elements, or you can change the data. For more information on changing the data, see "Editing a Value for a Document Element (Edit Data command)" on page 108.

About the Input Mapping Pane Context Menu

The Input Mapping pane has a context menu you can access to perform tasks on the Input Parts. To access the context menu, click the right-mouse button anywhere in the pane. The context menu is shown below.

Creating a Repeat for Element, Declare Alias, or Declare Group Action

You can create a Repeat for Element, Declare Alias, or Declare Group action on an element that repeats in the Input Part. See "The Declare Alias Action" on page 134.

You can also create a group of DOM elements which aids in transforming DOM elements into a structure that is different from their original positions in the sample document. When you create a group, you can perform aggregate operations against a group. For instance, you can arrange DOM elements into a group by U.S. state (e.g., Alabama, Arizona) then sum the total sales by each state.

Groups always work in conjunction with a Repeat for Group action. For more information about creating and using groups, see "The Repeat for Group Action" on page 185.

Editing a Value for a Document Element (Edit Data command)

You can select an element in an Input Part and set the value for it. This is helpful when you run the animation tools to test the component.

NOTE: The value is temporary for the editing session.

➢ **To edit a value for a Document element:**

1 Select an XML element from your document.

2 Click the right-mouse button in the input pane.

3 Select **Edit Data**.

4 Notice that the document element you selected appears in the dialog box. Type in the value you wish to set for the element.

5 Click **OK**. The value appears in the Data area, next to the element you selected.

Add to Display

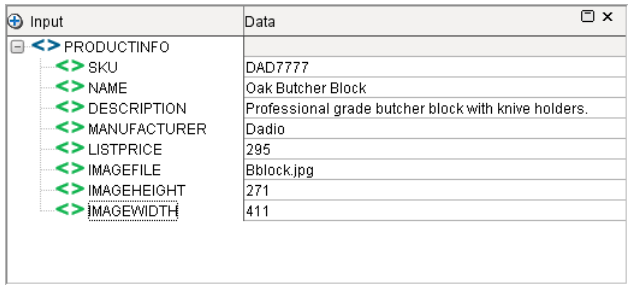
Use this context menu option to add additional XML documents to the display area of the XML Panel. When you select **Add to Display** from the menu, a list appears containing the currently unopened documents associated with the component, as well as the PROJECT variable and _SystemFault. Select from this list to open any of these items.

View Commands

You can view individual XML Message Parts as a Tree, as Text, or in stylized form (using an XSL stylesheet).

Tree View

The default view displays the Message Part as a tree, as shown below.

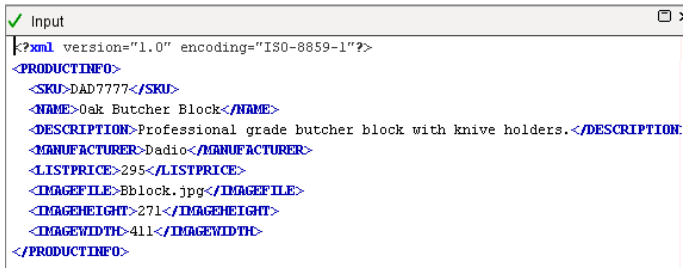


This view allows you to edit element and attribute *values* (that is, document data) but not the structure of the XML.

Text View

In Text View, you can see and edit the complete XML file, including structural elements.

Display the Message Part as text by clicking on the right mouse button (anywhere in the XML panel) and selecting **View**, then **As Text**. The XML then appears in plain-text form, as shown here.

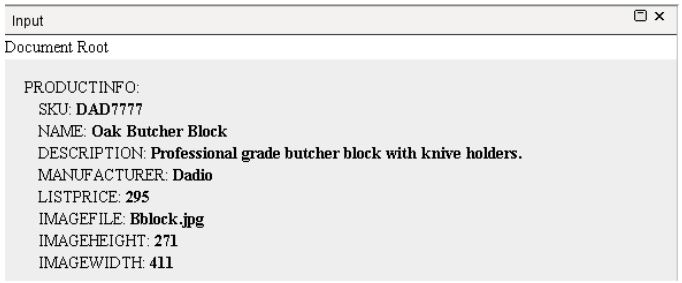


Text view offers a convenient way to inspect non-content-model portions of the Input, Temp or Output Parts, such as comments, processing instructions, DOCTYPE declarations, and so forth.

NOTE: The Text view, like the Tree view, is updated dynamically during animation so you can see the results of individual Map actions as they are executed.

Stylized View

When the Stylized view is selected (by clicking with the **RMB** and selecting **View>As Stylized**), you get a view of the XML message contents that looks like this:



This view gives a “report” style overview of the XML contents so that you can see at a glance what the content is for all attributes and elements.

NOTE: The default XSL stylesheet that Composer relies on for creation of this view can be found inside the **xcd-all.jar** file in your **\bin** directory; its name is **default.xsl**. You can edit or replace this file by extracting (unzipping) it from the jar file and reinserting an editing version in the same place in the jar.

Show Comments

Another View function is **Show Comments** which allows you to toggle the visibility of comments in the source XML file. Comments, signified by markers `<!--` and `-->` wrapping a section of content, constitute DOM nodes, but you may not always want to view them, particularly if you have a lengthy document.

Expanding a Document Tree

You can display all elements in a document tree by clicking the right-mouse button and selecting **View>Expand Tree**.

Another way to expand the tree is by clicking on the plus icon just to the left of the Part name (e.g., the word Input) at the top left-hand corner of the pane .

Collapsing a Document Tree

You can hide all elements in a document tree by clicking the right-mouse button and selecting **View>Collapse Tree**.

Another way to collapse the tree is by clicking on the minus icon just to the left of Part name (e.g., the word Input) at the top left-hand corner of the pane.

Reloading a Document Tree

You can reset a specific XML Message by bringing up the context menu and selecting **View>Reload Tree**. This allows you to reload an individual document tree within the XML Map component. You may wish to reload a tree if during testing, you halted animation, leaving the document in an unfinished state.

Launch Editor

Selecting **Launch Editor** from the context menu opens your document in the default XML editor you specified during installation.

Load XML Sample

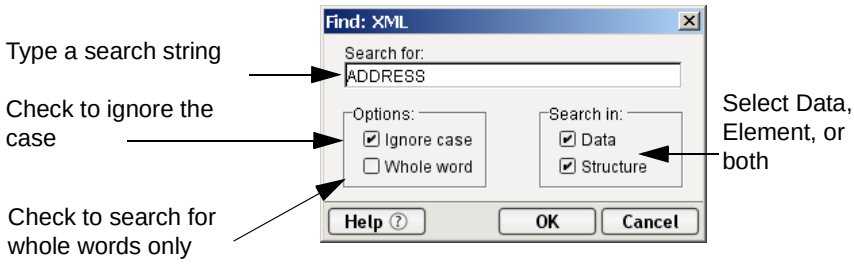
The **Load XML Sample** function available in the context menu allows you to load other sample documents from a template into a message part for testing the component. See “Loading a Sample Document” on page 119.

Save XML As

Selecting **Save XML As** from the context menu allows you to save the structure of the currently open Message Part into an XML document. See “Saving a DOM as an XML Document” on page 121.

Finding a Document Element

You can search for element names and element data using the **Find** command (which appears in the XML Panel context menu). The Find dialog allows you to enter a value and search the document tree. You can search for partial words or whole words only, and you can ignore the text case when searching. The Find Text dialog box is shown below.



Finding the Next Document Element

You can search for the next occurrence of a word or string you searched for previously. There is no dialog box when you select **Find Next** or press the **F3** key. Instead, Composer locates the next occurrence of the last find. If no match is found, nothing happens.

Validating a Dom

You can validate the DOM against its DTD or schema definition file by picking **Validate** from the context menu. Validating is useful during the construction and testing of your component.

About the Output Mapping Pane

The Output Mapping pane displays the Output Part. The Output Mapping pane also has a context menu, as shown.

The options on the Output Mapping pane context menu are similar to the ones on the input mapping pane context menu, but with the differences described below.

Mapping an Input Element to an Output Element

You can use the Map action by selecting it from the right-mouse context menu.

Setting a Value

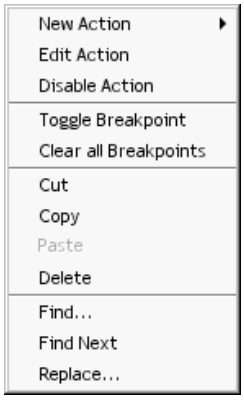
The Edit Data option on the Output Part allows you to inspect the value of a node but not change it.

About the Action Model Pane

All components have a single Action Model. The Action Model represents the mappings, transformations, and other actions that will be performed on XML documents during runtime processing. The Action Model Pane is also resizable within the XML Map Component Editor window. Most of your activity that takes place in the Action Model pane involves adding and editing actions.

About the Action Model Context Menu

If you right-click in the Action Model, you see the menu shown below.



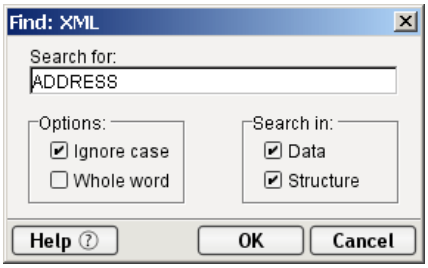
From this menu, you can select actions and perform other tasks.

Replacing Text in the Action Model

You can replace a word or string using the Replace option on the right mouse menu or on the component editor Edit menu.

➢ **To replace text:**

1 Right click in the Action Model and select **Replace** (or select an action and select **Replace** from the **Edit** menu).



2 Enter the search text and click **OK**.

3 Composer finds the first occurrence and asks you to confirm the replacement. You can then replace the next or all occurrences.

Adding Actions to a Component

Once you have specified the Input and Output templates, the XML Map Editor opens, and you are ready to start adding actions. Actions are the processing steps that take place within the component. You will read more about actions in later topics.

Within components, you add actions to map DOM elements, read and write data from files, send e-mails, and other common tasks. A collection of actions is referred to as an Action Model.

An action in the Action Model is displayed as a line and contains an icon for the action type along with an abbreviated definition of the action. Some actions are subordinate to other actions. For instance, you can create a Repeat action that controls loop processing, then add actions inside the loop. The actions inside the loop are subordinate to the Repeat action and appear indented beneath it. They process as long as the Repeat action is true.

➢ **To add actions to the Action Model:**

1 Position the cursor in the Action Model pane above where you want the next action inserted.

2 Add an action using any of following methods. The new action is inserted below the line you highlighted.

- **Drag and drop**. You can add Map actions by dragging and dropping elements from an Input Part to the output or temp DOMs. Simply click on an element in the Input Part and drag it on top of the output or temp DOM.
- **Copy and Paste**. You can copy an action in the Action Model pane and paste it somewhere else in the pane, or into an Action Model pane of another component.
- **The Action menu**. Highlight a line in the Action Model pane and select an action from the **Action** menu. The new action is placed directly under the highlighted line.
- The **Action Model pane context menu**. Click the right-mouse button anywhere in the Action Model pane and its context menu appears.
- The input and output **mapping pane context menus**. You can add actions to the Action Model pane by selecting DOM tree elements and then selecting actions from their respective context menus.

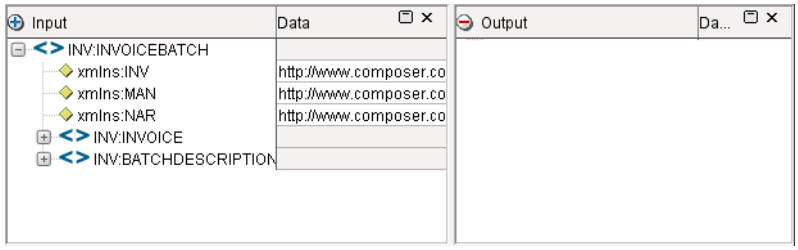
NOTE: You can reorder actions in the Action Model by dragging them to a new position.

Once you've created the Action Model, and before you process the component with live data, you should test the component. Perform testing by using Composer's Animation tools. With the Animation tools, you can set breakpoints, start a animation, step into and over actions, and pause the animation.

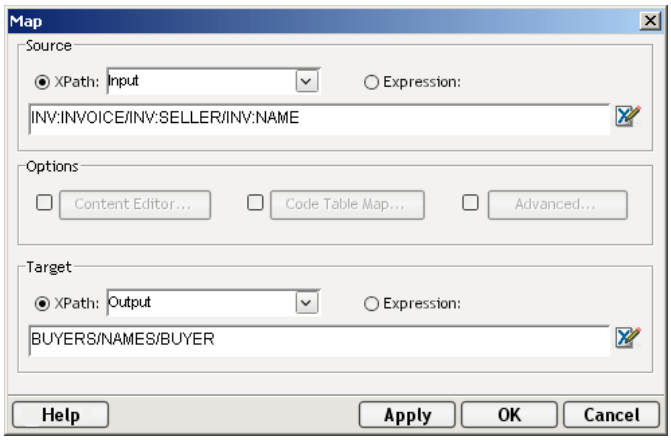
Creating an Output Document without Using a Template

You can specify an output XML template that contains no structure by selecting {SYSTEM}{ANY} as the Output template when you create the component. You can then build the Output Message Part dynamically by mapping input Part elements to output Part elements that do not yet exist.

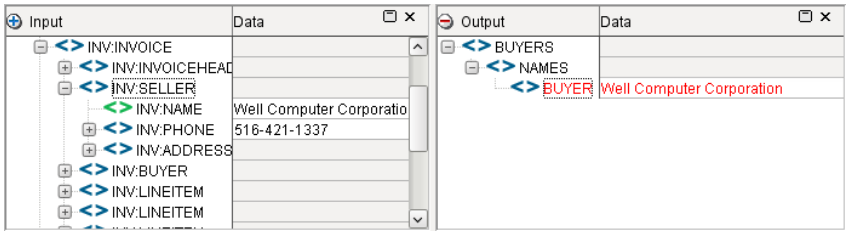
For example, the following illustration shows a component with an input Part containing elements, and an output Part that has no elements.



Notice there is nothing in the output document. To dynamically build an output document, you can map input Part elements to a structure in the output Part that does not exist. In the next illustration, the Seller in the input Part is mapped to a line item called Buyer in the output Part.



The next illustration shows the resulting XML Document Panels.



You can create any output document structure by mapping an input document element to an output XPath. Make sure you map to a fully-qualified document name.

NOTE: In reality, Output and Temp Message Parts are always built dynamically. The presence of a sample document is merely a productivity aid to help you define actions.

If the Output Part you created can be used for building other components, you may want to save it as an XML document and use it as a sample inside an XML Template. See “Saving a DOM as an XML Document” on page 121 for more information.

Using Temp and Fault Messages with a Component

In addition to creating Input and Output Message Parts, you can also create Temporary and Fault Message Parts. Temp Parts are used as work areas for performing complex manipulations of between Input and Output Parts. Within a Temp document pane, you can add elements from any of the Input Parts by using any of the five mapping methods. See “About the Action Model Pane” on page 112.

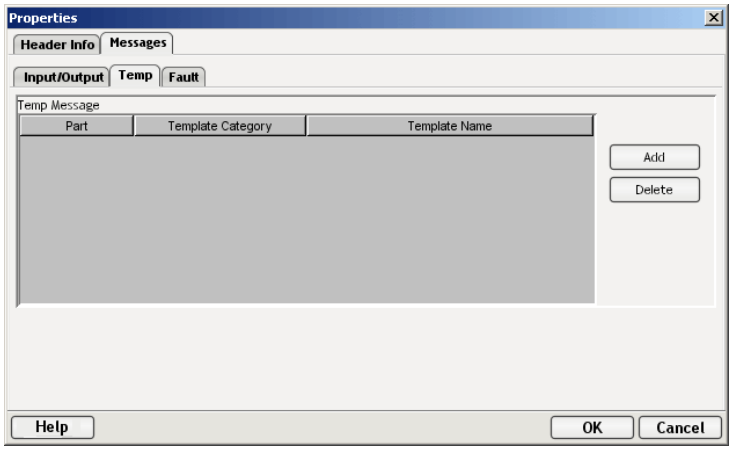
Fault Documents allow you to pass information back to clients when a fault condition occurs in a component or service. In Composer, a Fault is, essentially, an in-memory XML document or Message Part, defined along with a component, just like Input, Output and Temp Parts. Fault Message Parts are used to store information received when a Fault or Error occurs in your service or component. It is a good programming practice to anticipate places in your program where errors may occur and surround them with Try/On Fault and Throw Fault actions. (See “The Throw Fault Action” on page 163 for an example demonstrating the use Fault documents in fault-related actions.)

Creating a Temporary Message Part

The Temp Part differs from the Input and Output Parts in the directionality of actions allowed. The Temp Part can be both a source and a target of mapping actions, whereas the Input Parts are only sources and the Output Part is only a target. You can add more than one temporary Part (you are only limited by memory), and you can delete them whenever you wish. Also, you can assign your own name to temporary Parts by typing over the **Temp** label. Temporary Message Parts can either be defined during the creation of a component (see “To create an XML Map component:” on page 101), or they can be added to an existing component.

➢ **To add a temporary Message Part to an existing component:**

1 From the **File** menu, choose **Properties**. The Properties dialog appears.



2 Click on the **Messages** tab.

3 Click on the **Temp Documents** tab.

4 Click the **Add** button at the far right. Selections for Part, Template Category and Template Name become available.

5 Enter an **Part** (your own label) for the temporary message part.

6 Select an XML Category from the **Template Category** dropdown menu.

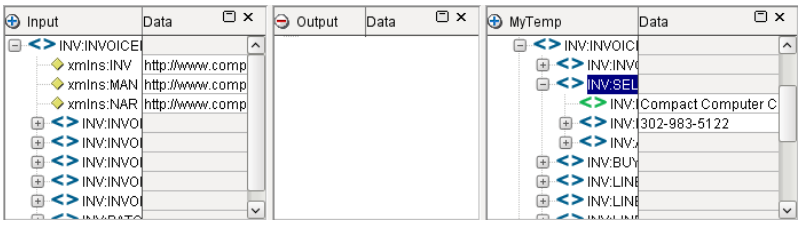
7 Select an XML Template from the **Template Name** dropdown menu.

8 Click **OK**.

9 Open the component. By default, the Component Editor will probably only show the Input and Output parts. To make the Temp document visible, go to the **View** menu and select **Show/Hide** to add the Temp Part you just created.

NOTE: Alternatively, you can click with the **RMB** in the Component Editor and select **Add to Display**, then pick the Temp document from the list.

The XML Map Component Editor displays a Temp message pane, as shown in the next illustration.

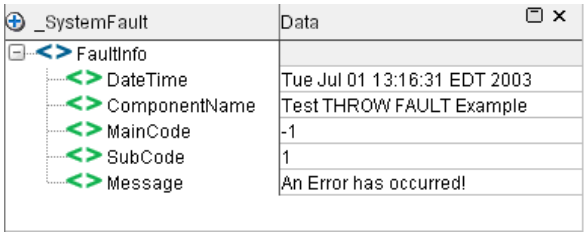


NOTE: You can also create a document object dynamically without using a template, as described in “Creating an Output Document without Using a Template” on page 113.

Creating a Fault Message Part

The _SystemFault Document

You can define the XML yourself for your Fault document, using your favorite editor, or you can use the default one provided by Composer, which is called _SystemFault. The XML information contained in _SystemFault also gets written to a global object called ERROR. The structure of the _SystemFault document is shown below:



Beneath the FaultInfo root are the following elements:

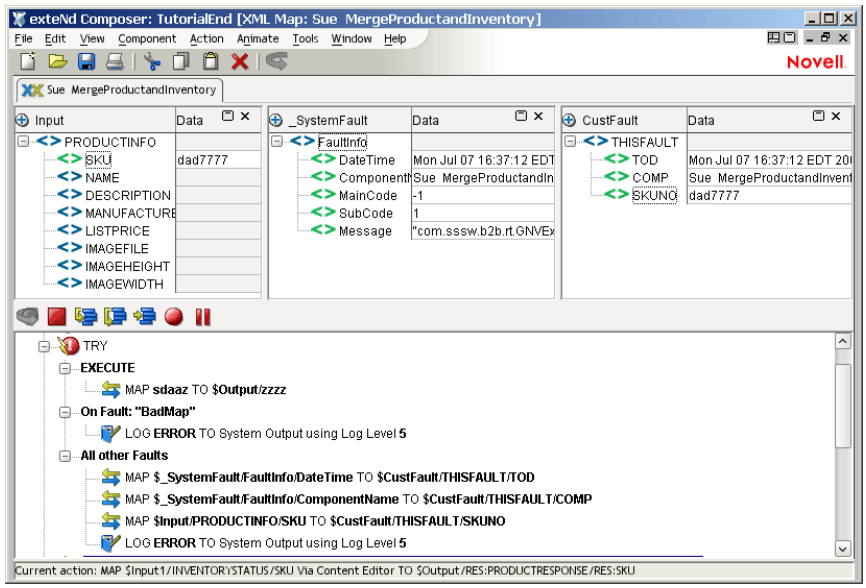
- DateTime which contains the Date and Time the fault occurred.
- ComponentName which contains the name of the Component which threw the fault.
- MainCode which contains the main code number for the error.
- SubCode which contains a sub-code number for the error.
- Message which contains the error message you specifically define when you set up a Throw Fault action (p.163). If you do not specify an error message in your Throw Fault action, you will see “A user defined Fault occurred!”. If the error occurred within a Try/On Fault action, and you did not specify a Fault, this element will be populated with an Exception message.

NOTE: By default, the Fault document will not be visible in the Component Editor. To View it, click with your **RMB**, select **Add to Display** and choose **_SystemFault**.

Creating a Custom Fault Document

The procedure for adding Fault Message Parts to your component is very similar to the procedure for adding Input, Output and Temp Parts. You begin by using your favorite editor to create an XML document that will be used to hold fault information. You can create as many Fault Message Parts as you need (you are limited only by memory) and Fault documents can have any structure that makes sense to your application.

You might want to use a custom Fault Part along with the _SystemFault document. For example, you could use the DateTime, Component and Message elements from _SystemFault to populate your own Fault document which woul also contain information about the service itself, a log message indicating the last action that was sucessfully executed and some information from the Input document that might have been missed due to the application halting before completion. Below is an example depicting a custom Fault message.



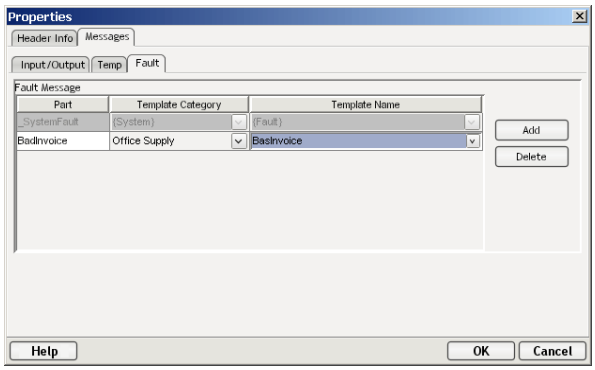
Once you've decided what Faults you need to capture and created the XML structures to support them, the Parts can either be defined during the creation of a component, or added to an existing component.

➢ **To add a Fault Part to an existing component:**

1 From the **File** menu, choose **Properties**. The Properties dialog appears.

2 Click on the **Messages** tab.

3 Click on the **Fault Documents** tab.



4 Click the **Add** button at the far right. Selections for Part, Template Category and Template Name become available.

NOTE: If you do not specify a Fault Part, error information will go into the basic fault document, called _SystemFault.

5 Enter a **Part** (your own label) for the fault message part.

6 Select an XML Category from the **Template Category** dropdown menu.

7 Select an XML Template from the **Template Name** dropdown menu.

8 Repeat as necessary for additional Fault documents.

9 Click **OK**.

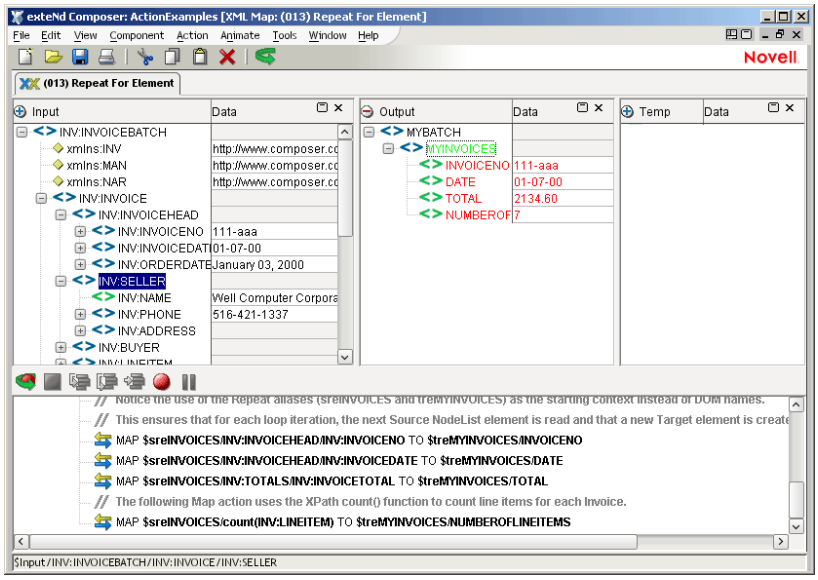
10 Open the component. By default, the Component Editor will probably only show the Input and Output parts. To make the Fault Part visible, go to the **View** menu and select **Show/Hide** to add the Fault Part you just created.

NOTE: Alternatively, you can click with the **RMB** in the Component Editor and select **Add to Display**, then pick the Fault document from the list.

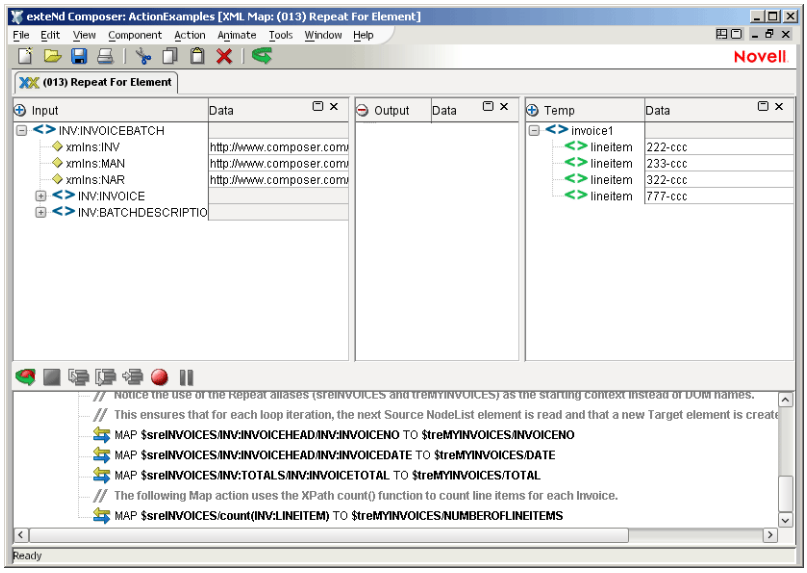
Reloading an XML Document

If you have made changes to the document structures through mapping, and wish to return the Message Parts to their original state, you can reload the XML documents. When you reload the XML documents, all Parts, including Temp and Fault (if these were created) are returned to the state defined by the input and output XML documents. Keep in mind, however, the Map actions in the Action Model pane remain.Thus, if you were to execute the component, all Map actions will run.

The illustration below shows a component that contains several Map actions that are reflected in the Input, Output, and Temp Parts.



Notice the detail in the XML and the Map actions in the Action Model pane. The next illustration shows the same screen after the XML documents have been reloaded.



The documents are back to their original state but the Action Model pane remains the same. Reloading XML documents is accomplished by selecting **Reload XML Documents** from the **Component** menu.

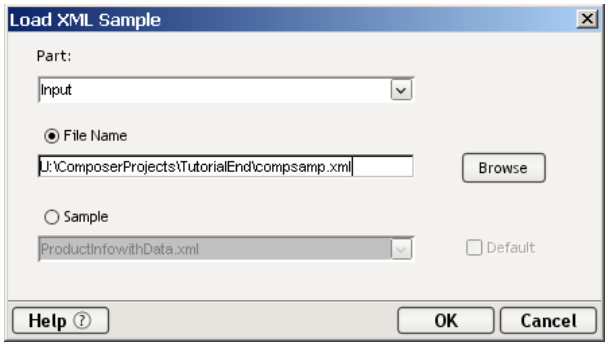
Loading a Sample Document

You can load different sample documents into any of the DOMs and use the new DOM structures for mapping elements or testing the Action Model. Loading a different sample document from your template allows you to test if your Action Model can handle all cases of XML documents your component might receive at runtime.

When you load an XML sample, the DOM changes, but the Action Model remains unchanged. When you are finished testing with the sample XML document, you can reload the original XML document(s) by repeating the Load XML Sample procedure.

➢ **To load a sample document:**

1. From the **File** menu, select **Load XML Sample**. Alternatively, click with the **RMB** in the XML Editor Panel and select **Load XML Sample** from the context menu. The Load XML File dialog appears.



2. Select appropriate message Part from the **Part** dropdown box where you want the new sample document loaded.
3. If you want to load a sample document that is not included in the original XML template, click **File Name** and type the name of the file. Alternatively, you can click **Browse** and find the file on your computer or network. You may also read in a file from a URL by explicitly preceding your filename with “http://,” “https://” or “ftp.”
4. If you want to load an XML file that is included in the original XML template, click **Sample** and select the XML document.
5. Check **Default** if you want to make the selected sample the default XML document for the selected Part in this component only. (This does not apply to the file name option).
6. Click **OK**.

Adding a Watch Variable

During the execution of your component, it can be very useful to examine the value of certain variables as a debugging aid. For this purpose, Composer offers a Watch List, and the ability to create Watch variables to add to the list.

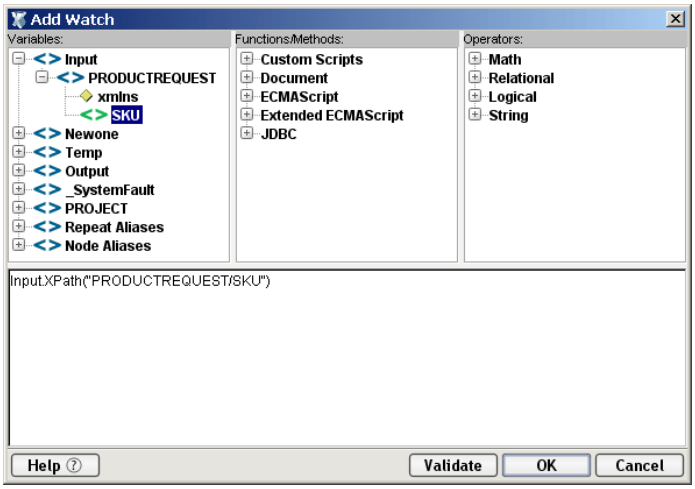
You can identify the following objects as Watch variables:

- Input, Temp, and Output Document location paths
- Location paths from PROJECT
- Repeat Aliases
- Node Aliases
- ECMAScript expressions and variables

➢ **To add an item to the Watch Variable List**

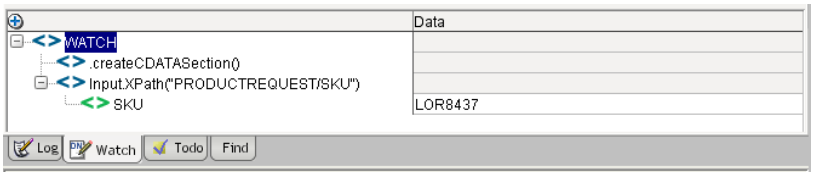
1 From the **Component Menu**, select **Add Watch**.

2 The Add Watch Dialog displays, giving you access to all the Variables, ECMAScript Functions and Methods and Operators associated with your project.



3 Doubleclick on the item you wish to add to the Watch list and click **OK**.

4 During the execution of your component, click on the Watch tab in the Output pane to view the status of the items in your Watch List.



The use of a Watch List, including examples of how this could be used as a debugging aid, are discussed in greater detail in Chapter 12.

Saving Your Component

Save your component often to make sure your work is not lost due to hardware or software failures. You can also save the component with a different name, making a backup copy. When you save it with another name, you can also change the XML properties, including the input and output XML templates.

➢ **To save the component with a new name:**

1 From the **File** menu, select **Save As**.

2 In the **Name** field, type a new name.

3 To change input and output XML documents, click the **XML Property Info** tab.

4 Change or add input XML documents.

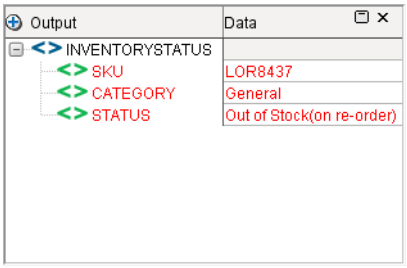
5 Change the output XML document.

6 Click **OK**.

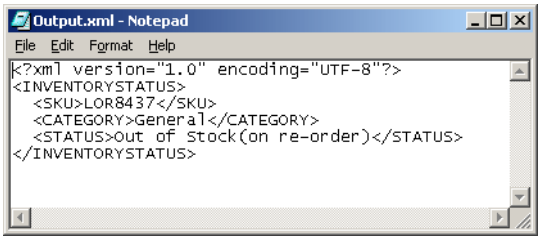
NOTE: If you have more than one component open at a time, clicking on **File>Save All** (or pressing **Ctrl-Shift-A**) will save all the open components at once. Similarly, **File>Close All** (**Ctrl-Shift-F4**) will save all components at once.

Saving a DOM as an XML Document

You can also save any DOM as an XML document. This creates (or overwrites) an XML document that contains the structure and data of the DOM. The following illustration shows an Output Part and the resulting XML document.



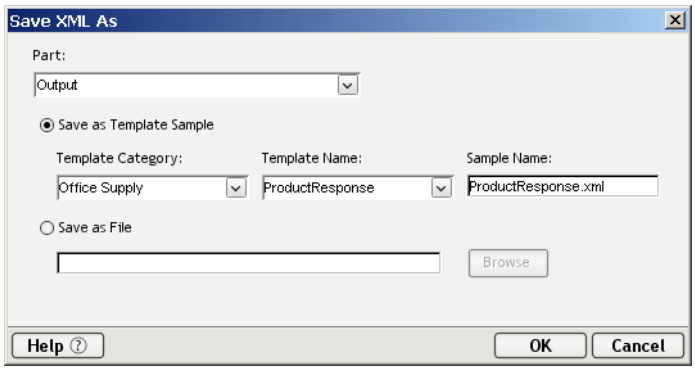
Output Part with data.



Resulting XML document

➢ **To save an in-memory DOM to an XML file:**

1 From the **File** menu, select **Save XML As**. The Save XML As dialog box appears.



2 Use the pulldown menu under **Part** to select the source DOM that you want to save to disk. In the above example, Output is selected.

3 Check the **Save as File** radio button.

4 Type a path and name for the XML document, or click **Browse** and select a path.

NOTE: If you select an existing XML document, it will be overwritten with the source DOM's structure and data.

5 Click **OK**.

Saving an XML File as a Template

Any DOM that's visible in the component editor can be saved as an XML Template directly (rather than first saving the DOM to a file, then importing it into a template). The target Template does not have to exist already; you can create one on-the-fly.

➢ **To save a DOM to an XML Template:**

1. From the **File** menu, select **Save XML As**. The Save XML As dialog box appears (as shown above).
2. Use the pulldown menu under **XML Document** to select the source DOM that you want to save as a template.
3. Check the **Save as Template** radio button.
4. If you are creating a template on-the-fly, enter a name for the **Category** (or else pick an existing category from the pulldown menu provided).
5. If you are creating a template on-the-fly, enter a name under **Template Name** (or else pick an existing XML Template name from the pulldown menu).
6. Enter a **Sample Name** for this XML document. (This will be the name of the file on disk. The file will be saved under **\xmlcategories\[CategoryName]\imports** in your project directory.)
7. Click **OK**. You will see the new XML Template appear in the Instance Pane of Composer's nav frame.

During on-the-fly creation of an XML Template using the above technique, you will not be prompted for any additional information (such as schema name or XSL stylesheet) to associate with the new template. If you want to inspect or edit the validation, stylesheet, or other properties of the new template, follow the procedure outlined below.

Inspecting and/or Editing XML Template Properties

At any after an XML Template has been created, you can inspect or change its properties. See"Editing an XML Template" on page 92.

Avoiding Out-of-Memory Problems

When you are working with large DOMs, it is advised that at design time, to avoid memory errors, you add the following line to the **xconfig.xml** file. See the sample **xconfig** file excerpt below.

Line to add (or edit): `<VM_PARAMS>-Xms64m-Xmx128m</VM_PARAMS>`

To adjust available memory for *deployed* services (in the runtime environment), you will have to alter the VM command-line options for the app server's VM. Consult your app server documentation for information on how to do this.

You can avoid many out-of-memory problems (at runtime *as well as* design time) by appropriate use of Performance Filters as described in the next section.

Using Performance Filters

The **Define Performance Filter** command (under the **Component** menu on Composer's main menubar) offers the potential for greatly improved performance when processing large incoming documents. It also offers significant benefits in terms of memory conservation, since a filtered document can require much less memory at runtime than an unfiltered document.

Performance filters work by stripping superfluous document elements (and attributes) from incoming XML documents. You specify which elements to ignore; Composer does the rest. In essence, the input document is "rewritten" on the fly in much-streamlined form, eliminating parts of the XML that are not necessary for your service. This results in a smaller in-memory DOM.

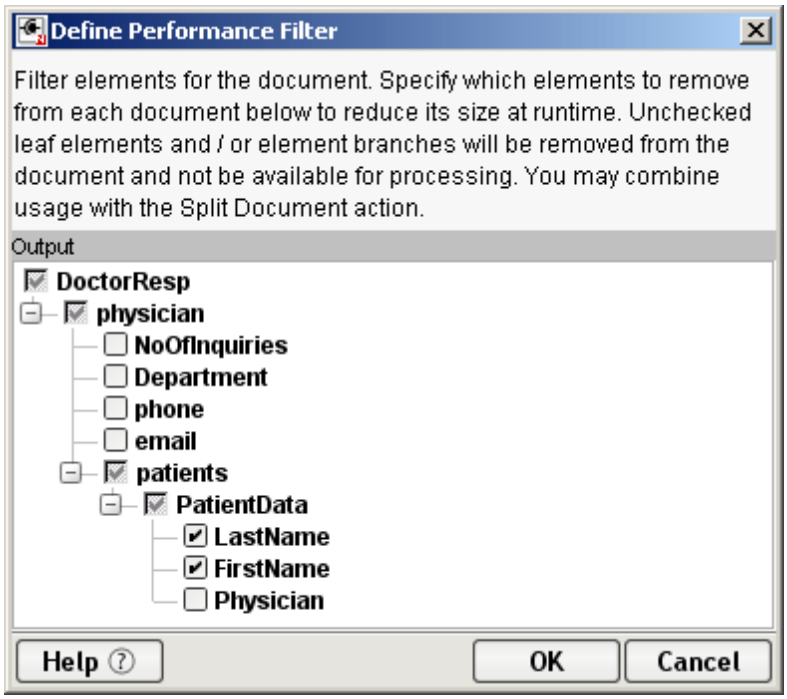
Document filtering is useful because it is very common for a service to operate on only a few XPath locations in a given type of document. For example, one service might operate on the "Customer Info" nodes in an order form; a different service might operate on the "Product Request Detail" nodes of the same order form; and so on. It makes sense for each service to see and use only the portions of the document that apply to that service.

➢ **To create a Performance Filter**

1 Open a Service xObject, if one is not already open.

NOTE: Performance filters cannot be defined on Components. They can be defined on Service xObjects only.

2 Under the **Component** menu on Composer's main menubar, select **Define Performance Filter**. A dialog appears.



Note that the document shown in tree-view form in this dialog is the *Input* document for the service. (It is not possible to view other documents in this dialog.)

3 Check the checkbox(es) next to the nodes you want to *keep* in the document. Unchecked nodes will be stripped off (discarded) so that the parsed DOM does not contain the elements in question. (See additional discussion below.)

4 Click **OK** to dismiss the dialog.

In the preceding illustration, the incoming document, with root node **DoctorResp**, will have a **/physician** node with a **/patients** node under it at runtime, and the **/patients** element, in turn, will have a **PatientData** element under it. Likewise, the latter will have child nodes **LastName** and **FirstName**. But since **Physician** is *not* checked, the incoming document will not have anything under the XPath:

```
DoctorResp/physician/patients/PatientData/Physician
```

NOTE: At design time, you will initially, upon opening a service, see the complete (non-filtered) Input document, with all its nodes present in tree view, even if you have defined a Performance Filter. But when you begin stepping through the action model in animation mode, the document tree view will update to show the reduced (filtered) runtime structure of the document.

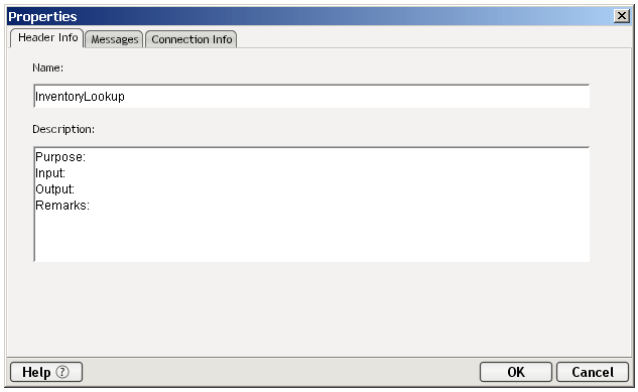
Viewing Component Properties

You can inspect (and in some cases edit) various properties of a component at any time.

➢ **To view or change component properties:**

1 Select **Properties** from Composer's **File** menu. The Properties Dialog will appear. Note that the dialog has three tabs:

- **Header Info**—This is the descriptive commentary you entered (or didn't enter) when you first created the component.
- **Messages**—This is equivalent to the second dialog in the New XML Map Component wizard: It shows the templates and template categories used in this component.
- **Connections**—This tab will be present only in components that are associated with a particular Connection Resource. (For example, a JDBC Component would have such a tab in its Properties.) The plain XML Map Component does not have this information, and this tab does not appear.



2 To view or edit descriptive commentary for this component, click the **Headers** tab and enter the desired information.

3 To view or change XML template choices, click the **Messages** tab. You can add or remove template documents and/or template categories as need be.

4 If your component uses special Connection Resources, click the **Connections** tab to view Connection Resource info for this component. (Not applicable to ordinary XML Map components.)

5 Click **OK** to dismiss the dialog.

6 **Save** your component.

Printing a Component

You can print the contents of a component. The printout contains:

- Time and date you printed the component
- Name and description of the component
- All XML documents that make up the Input, Output, and Temp Parts
- All actions in the Action Model

➢ **To print a component:**

1 From the **File** menu, select **Print**.

2 Select a printer.

3 Click **OK**.

Designing, Testing, and Running a Component

The following table shows how sample documents are used when designing, testing, and running a component.

DOM	While designing in Composer	While using Animation Tools in Composer	While executing in Server
Input	Samples can be loaded and used as design time aids for building actions and test data.	The default sample document is loaded and used to simulate a runtime Input Part.	XML data is passed in by another component, a service, or a Service Trigger.
Temp(n)	Samples can be loaded and used as design time aids for building actions and test data.	The sample document is not loaded. The Part is built by the Action Model.	The Part is built by the Action Model
Output	Samples can be loaded and used as design time aids for building actions.	The sample document is not loaded. The Part is built by the Action Model.	The Part is built by the Action Model
Fault	Samples can be loaded and used as design time aids for building actions and test data.	The sample documents is not loaded. The Part is built by the Action Model.	The Part is built by the Action Model

7 Basic Actions

Up to this point, you've learned how to create XML templates and an XML Map component that uses templates for inputs and outputs. Now it's time to learn about the actual work that takes place. This is where the action is.

NOTE: This chapter defines the basic actions available within the XML Map component. The next chapter covers more powerful actions and Chapter 11, "Applying Actions to Common Tasks" covers detailed examples of using some of these actions.

What is an Action?

An *action* is similar to a programming statement in that it takes input in the form of parameters and performs specific tasks. For instance, the Send Mail action sends an e-mail when you supply the recipient's e-mail address as one of the parameters.

Before looking at individual actions, you should first understand exteNd's Action Model. You may remember an earlier discussion that a component is a set of instructions for processing XML documents or communicating with non-XML data sources. This set of instructions is called an *Action Model*. In Composer, an Action Model performs all data mapping, data transformation, and data transfer within components and services.

An Action Model is made up of a list of actions. All actions within an Action Model work together. As an example, the Action Model for a component might read invoice data from a disk, retrieve the e-mail addresses from the invoices, and send e-mail messages to notify the recipients that their invoices were received.

The Action Model mentioned above would be composed of several actions. These actions:

- Open an invoice document and read invoice data into memory
- Extract the e-mail address from the invoice
- Compose and send an e-mail
- Update the invoice record to show that an e-mail was sent then close the file

Using Composer Actions

Composer provides actions with the basic XML Map component. These actions are also available for *all other component types*, such as JDBC Components, JMS Components, etc. Actions are grouped on the Action menu as Basic Actions and Advanced Actions. The following table lists the suite of basic actions available in Composer. The Advanced Actions are described in the next chapter.

Basic Action	Description
Comment	Documents the Action Model. You can use comments to clarify the processing, especially if Decisions and/or Repeats are used in the Action Model. Keyboard shortcut: **Ctrl-E**
Component	Executes another component or service and defines runtime DOMs to be passed to, and received from the called component. Keyboard shortcut: **Ctrl-T**
Decision	Allows you to execute one of two sets of actions based on a condition you specify. Processing branches along a True or False path, depending on how your condition is resolved as the component executes. Keyboard shortcut: **Ctrl-D**
Declare Alias	Allows you to assign an arbitrary label to any XPath, for convenience purposes. The label expands to the full XPath at runtime or animation time.
Function	Executes either an ECMAScript script function or a custom script you have previously created. You can create custom scripts using Composer's Custom Script Resource Editor. Keyboard shortcut: **Ctrl-U**
Log	Writes information to various log files specified in the component. There are three Log types: System Output, System Log, and User Log. Keyboard shortcut: **Ctrl-L**
Map	Transfers and optionally transforms element data from one XML DOM to another. Keyboard shortcut: **Ctrl-M**
Send Mail	Automatically sends an e-mail to a specified e-mail address during execution of the component.
Switch	Allows program control to branch to a particular block of actions based on a match between an input value and a Case value. This is essentially a convenience action that can be used to eliminate long, hard-to-read if/else (Decision action) chains.
Todo	Gives you a place to maintain a Todo list that organizes and tracks your tasks.

Creating an Action

There are four methods for creating a new action:

- From the **Action** menu in the main menubar
- From the **Context** menu available by right-clicking within the **Component Editor**
- Using keyboard shortcuts (available for the most commonly used actions only, see table above)
- Using Cut/Copy and Paste

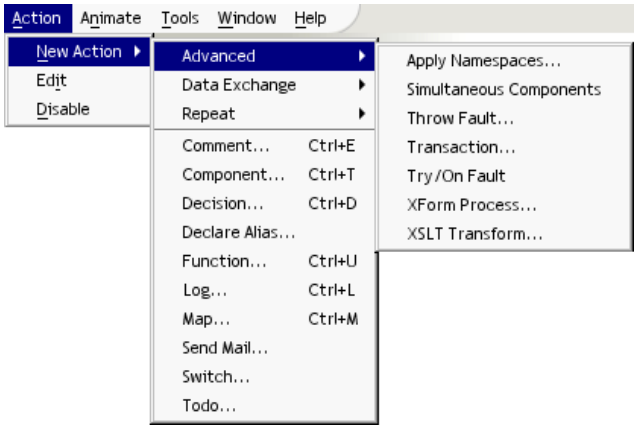
In all cases, you must have the component open before you can create an action.

➢ **To create an action using the Action menu:**

1 Open a component.

2 Click the mouse on (that is, highlight or *select*) a line in the Action Model just above the place where you want a new action. The new action will be inserted below the line you selected.

3 From Composer's **Action** menu (main menubar), select **New Action** and then the type of action you wish to create.

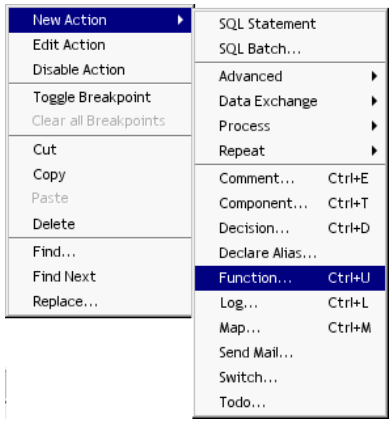


4 If a dialog appears, type or select parameters pertinent to the action, as required. (These are described individually in subsequent topics. See below.) Then dismiss the dialog, as applicable.

➢ **To create an action using the Context menu:**

1 Select a line in the Action Model where you want to place the action. The new action will be inserted below the line you select.

2 Click the right mouse button to display the **Context** menu:



3 Select an action from the Context menu.

4 Interact as necessary with any dialogs that appear.

5 Dismiss the dialog(s).

➢ **To create an action using a shortcut key:**

1 Select a line in the Action Model where you want to place the action. The new action will be inserted below the line you select.

2 Create your new action by pressing the key combination indicated in the table above. For example, pressing **Ctrl-L** will add a Log action to your model.

➢ **To Cut, Copy, or Paste an action:**

1. Select (click on) the action in the Action Model pane.
2. Choose **Cut, Copy, Paste**, or **Delete**, as appropriate, from the Edit menu in the main menubar, or from the context menu available via right-mouse-click.
3. Type Control-Z (or choose Undo from the Edit menu) if you want to undo the operation.

In addition to adding actions, you can edit existing actions and disable actions within an Action Model. When you disable an action, it does not execute, but it remains in the Action Model, and you can enable it at a later time.

➢ **To edit an action:**

1. Doubleclick any action in the Action Model and edit it.
2. Alternately, you can select the action in the Action Model pane.
3. From the **Action** menu, select **Edit**. A dialog box for the action type appears.
4. Make any necessary changes to the action.
5. Click **OK**.

➢ **To disable an action:**

1. Select the action in the Action Model pane.
2. From the **Action** menu, select **Disable**. The action is grayed out.
3. Repeat steps 1 and 2, selecting **Enable**, to enable the action again.

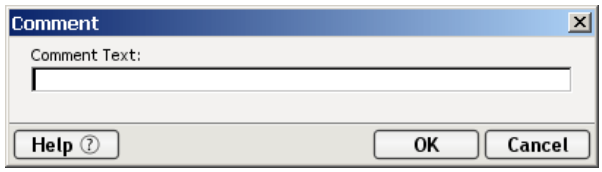
The rest of this chapter describes each basic action and gives examples on how to use them.

The Comment Action

You can use the Comment action to document your Action Model and clarify the processing that takes place. You can add comments anywhere within an Action Model. They perform no processing of their own.

➢ **To add a Comment action:**

1. Open a component.
2. Select a line in the Action Model where you want to place a comment. The new comment is inserted below the line you selected.
3. From the **Action** menu, select **New Action**, then **Comment**, or press **Ctrl-E**. The Comment dialog appears.



4. Type your comment.
5. Click **OK**.

The Component Action

The Component action calls and executes another component or service with runtime inputs and outputs that you specify. You can call any component in your project. To call another component, you must specify four parameters to the action:

- Component Type
- Component Name
- Passed IDs
- Returned ID

The Component Type is simply the category of component you wish to call. The component types do not correspond to those listed in the Composer Category pane under the Component heading. The following strings are valid values and are case sensitive:

- service
- map
- jdbc
- 3270
- 5250
- cicsrpc
- html
- jms
- vt100

depending on whether or not you have the Connect installed that implements that Component Type.

The Component Name is the name of the component you wish to call or target component. The Component Name must be one that exists within the Component Type you select.

The Passed ID are document names within the current component or service. You can specify none, one, or more documents to pass into the target component. The document names you specify here will be passed into the target component as its Input documents.

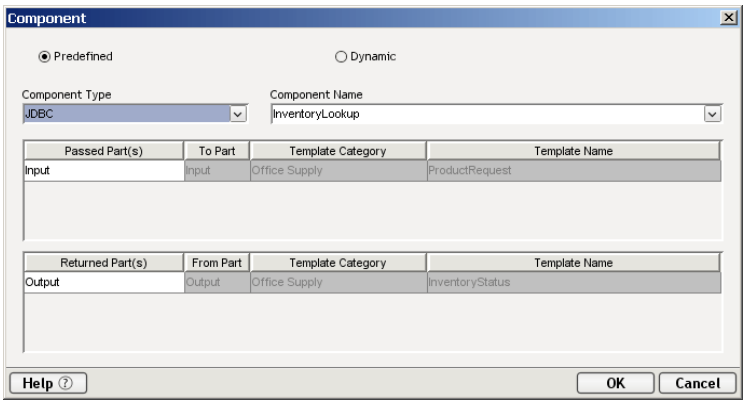
The Returned ID is the name of a document within the current component or service that will receive the results of the target component. You can use the name of an existing document or force the creation of a new document by specifying a name that does not already exist.

You can specify these parameters in one or two ways: *Predefined* or *Dynamic*. A Predefined Component action populates the four parameters with values derived from the current state of the project. Once specified, these values remain fixed for all executions of the action unless you manually change them. A Dynamic Component action populates the four parameters at runtime with values calculated from expressions you create. This allows the Component action's behavior to be flexible and vary based on runtime conditions each time it is executed. One Component action can execute a different component depending on various runtime conditions, or pass in different Input documents, or receive results into different result documents.

➢ **To add a Predefined Component action:**

1. Open a component.
2. Select a line in the Action Model where you want to place a call to a component. The new action is inserted below the line you selected.
3. From the **Action** menu, select **New Action**, then **Component**, or press **Ctrl-T**. The Component dialog appears.

4 Select Predefined, by clicking on the radio button, if it is not already selected.



5 Select the relevant Component Type from the drop down list.

6 Select a Component Name to execute (the list of Components is context sensitive to the Component Type selected).

7 In the **Passed ID** field, select a source component DOM.

8 In the **Returned ID** field, select the source DOM into which the called component will return its Output. If you wish to create a new DOM, you may type the name in the **Returned ID** field.

9 Click **OK**.

➢ **To add a Dynamic Component action:**

1 Open a component.

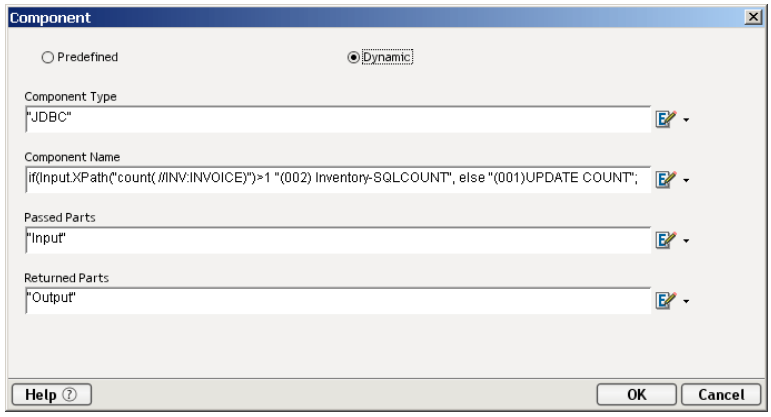
2 Select a line in the Action Model where you want to place a call to a component. The new action is inserted below the line you selected.

3 From the **Action** menu, select **New Action**, then **Component**, or press **Ctrl-T**. The Component dialog appears.

4 Select Dynamic, by clicking on the radio button, if it is not already selected.

5 Create an ECMAScript expression that evaluates to one of the following valid Component Types: map, service, JDBC, 3270, 5250, CICSRPC, JMS, HTML

NOTE: A Component Type will only be valid if the Connect implementing that Type is installed in your version of Composer.



6 Create an ECMAScript expression that evaluates to a valid component or service name in your project.

7 Create an ECMAScript expression that evaluates to a valid document ID at runtime in the current component or service. This document will be passed to the target component or service as its Input document. If passing more than one document, the expression must evaluate to a single string containing a comma-separated list of document IDs (e.g. Input, Input 1, Temp, MyDoc).

8 Enter an ECMAScript expression that evaluates to a document ID that will receive the results of the target component.

The Decision Action

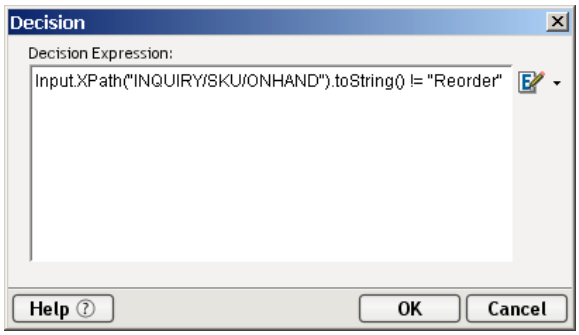
The Decision action creates an *if. . . then* branching between actions or group of actions. You use a Decision action to select one branch or another, based upon a condition you supply. The condition must use an ECMAScript comparison operator, such as = =, <, >,!, >=, <=, (&), OR (||), or <>. The expression must resolve to the Boolean true or false statement. For instance, you can check to see if an invoice is older than a certain date and send an e-mail if it is.

➢ **To add a Decision action:**

1 Open a component.

2 Select a line in the Action Model where you want to place the Decision action. The new action is inserted below the line you selected.

3 From the **Action** menu, select **New Action**, then **Decision**, or press **Ctrl-D**. The Decision dialog appears.



4 Type the expression using any of the ECMAScript comparison operators or click the **Expression Builder** button and create a Decision script (ECMAScript expression) that will evaluate to *true* or *false* at runtime.

5 Click **OK**. The Action Model displays the following Decision action, which tests for the existence of an INVOICE node.

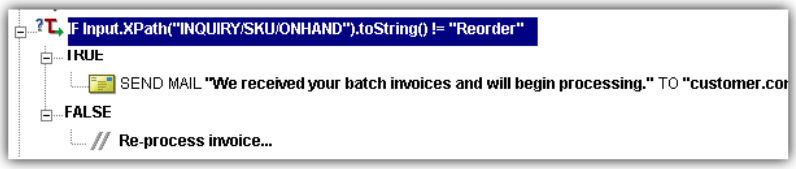
6 In the Action Model pane, select the **TRUE** icon.

7 Add one or more actions that will execute if the expression is true. you can, of course, cut/copy actions via drag and drop from outside the true branch to within the true branch.

8 Select the **FALSE** icon.

9 Add one or more actions that will execute if the expression is false.

You can nest other Decision actions inside the **TRUE** and/or **FALSE** branches of the Decision action. The following illustration shows a complete decision in the Action Model pane.



The Declare Alias Action

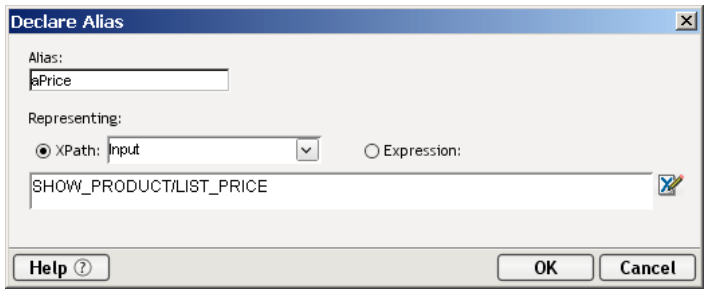
The Declare Alias action allows you to apply your own arbitrary custom label to a given XPath expression (valid within the scope of a given action model). You would use this action to make your action model more readable and save typing.

➢ **To add a Declare Alias action:**

1 Open a component.

2 Select a line in the Action Model where you want to place the Declare Alias action. The new action is inserted below the line you selected.

3 From the **Action** menu, select **New Action**, then **Declare Alias**. The Declare Alias dialog appears.



4 Type the name you intend to use in the **Alias** text field.

5 Choose either the **XPath** or the **Expression** radio button.

6 If you have chosen the XPath radio button, select a target DOM (representing the document containing the target XPath) from the dropdown menu. Then enter the XPath to the target node in the text field below.

7 If you have chosen the Expression radio button, type the appropriate ECMAScript representation of the target XPath in the text field, or click the Expression Builder icon (to the right of the text field) and use the Expression Builder pick-lists to build an expression.

8 Click **OK**. The new action is added to your action model

In the above example, the Input Part has a node called SHOW_PRODUCT/LIST_PRICE. Rather than type $Input/SHOW_PRODUCT/LIST_PRICE repeatedly throughout the action model, one could, for convenience, assign an alias (an arbitrary name) to the XPath expression. In this case, the alias “aPrice” has been assigned to $Input/SHOW_PRODUCT/LIST_PRICE. From this point on, throughout the action model, one can use “aPrice” instead of $Input/SHOW_PRODUCT/LIST_PRICE. At runtime, the alias will be expanded to the complete XPath.

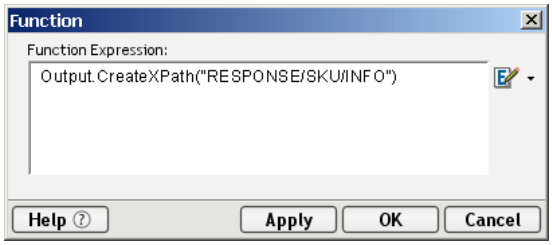
The Function Action

The Function action executes either an ECMAScript function or a custom script function you have already created in the Custom Script Resource Editor. To manipulate a DOM element, the script you call in the Function action must reference a fully qualified DOM element name in the current component.

Custom Script functions you create and add to an Action Model can act upon any XML tree element. For instance, you can create a function that changes the format of a date element. You can create a function that performs a math function on the contents of an element. You can also perform file system, database, or URL functions that have no interaction with a Message Part. The Function Action can also be used to call Java methods that you have registered in the Custom Script Resources. This gives an ability to visually integrate complex (and simple) Java processing directly onto the Action Model.

➢ **To add a Function action:**

1. Open a component.
2. Select a line in the Action Model where you want to place the Function action. The new action is inserted below the line you selected.
3. From the **Action** menu, select **New Action**, then **Function**, or press **Ctrl-U**. The Function dialog appears.

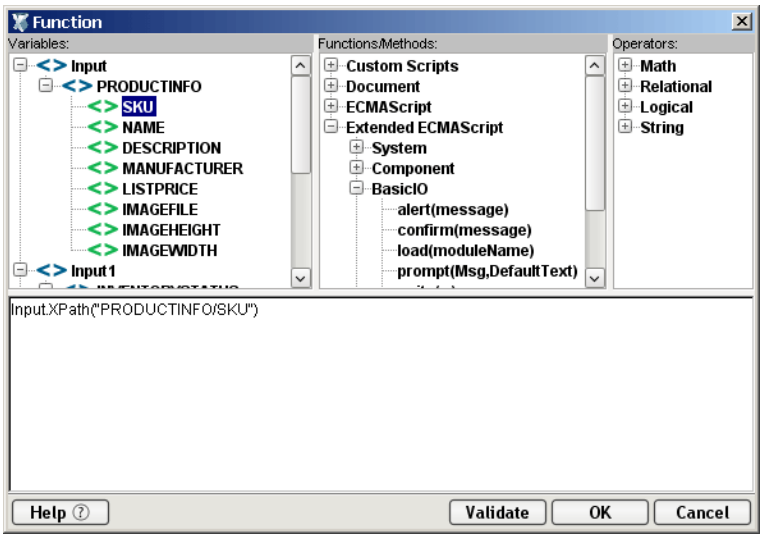


4. Type the function in the **Function Call** field or click the **Expression Builder** button to build an ECMAScript expression (discussed below). Function calls are case sensitive. Also, if the function requires parameters, make sure to include them in the function call.
5. Click **OK**. Alternately, you can click on the Apply button to see the affect of the Function action without closing the dialog. This allows you to make repetitive edits to a Function action quickly see the results.

➢ **To use the Expression Builder:**

1 Add a new function action as described in the previous section.

2 Click the **Expression Builder** button to open the Function Expression Builder dialog.



3 Doubleclick variables, functions/methods, or operators to insert them into the function. You can also type directly into the function.

NOTE: Make sure the function follows ECMAScript standards or it will not compile or run correctly. It is usually more efficient to create functions within a Custom Script resource and test them before using them. When creating a Function action, you can simply refer to the Custom Script function name and supply it any parameters.

4 Click **Validate** to verify the script before saving it.

5 Click **OK** to save the script.

6 Click **OK** again to add the function action.

NOTE: Since ECMAScript is an interpreted language, **Validate** doesn't check any runtime dependent expressions other than to see if they conform to valid ECMAScript syntax.

The Log Action

Log actions are designed to provide customizable reporting capabilities (design-time as well as runtime) for Composer applications. You can exercise fine control over the degree of reporting desired, by the use of Log Level settings (see further below); Log Actions needn't simply be turned "on" or "off."

Some examples of where the Log Action might be used are:

- To write out certain error information to the operator console when a Try On Fault condition is reached.
- To aid in debugging. (Since Log messages can be constructed as ECMAScript expressions, you can log information about variables or DOM contents whose values are known only at runtime.)
- To capture specific information from each cycle of a Repeat for Element loop.
- To help create self-reporting components during development.

Log File Locations

The Log action writes information to any of various locations external to Composer and exteNd Composer Enterprise Server. The actual locations are specified by the action. There are three locations for log output: System Output, System Log, and User Log (see below).

System Output

The System Output option writes out messages you specify in the Log Expression field to the Java Virtual Machine process window at design time or the Application Server console at runtime.

To create a Log message you can write any valid ECMAScript expression or use the Expression Builder to generate a Log Expression. Each message logged is preceded by a Date/Time stamp and the Component doing the logging. These messages also appear in the Message frame of the main Composer window.

System Log

The System Log option writes out messages you specify in the Log Expression field to the filename specified in the <LOGFILE> element of the Composer configuration file: **xconfig.xml.** You can change the name and location of the log file from the Composer Tools menu by selecting **Tools > Preferences** from the Composer menubar and going to the **General** tab.

User Log

The User Log option writes out messages you specify in the Log Expression field to a file you specify in the User Log File field of the Log Action dialog (see below).

To create a Log message, you can enter a static string or write any valid ECMAScript expression (or use the Expression Builder to generate a Log Expression). The results of the Log Expression will be written out to the Log as text. Each message logged is preceded by a Date/Time stamp and the Component doing the logging.

To create a User Log File you can also write any valid ECMAScript expression to generate the file name, click the **Expression Builder** button to use the Expression Builder.

Log Priority Levels

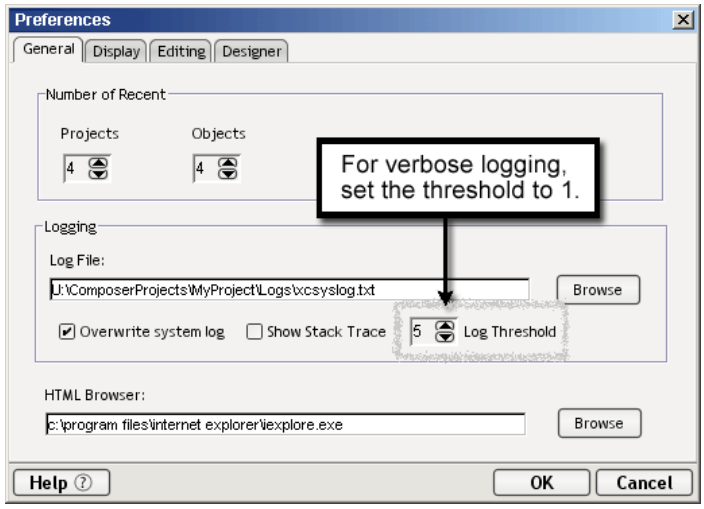
Individual Log Actions can be assigned priority levels (from 1 to 10). At runtime, a Log Action's priority level is compared against a reporting threshold value which you set in the General tab of the Preferences dialog under the Tools menu. Any Log Action whose priority is equal to or greater than the reporting threshold will be executed (that is, its message will be logged to system output or to disk, as appropriate), while Log Actions of lower priority will *not* have their messages reported.

Priority levels for individual Log Actions can be set in the Log Action dialog. The reporting threshold is set in the General tab of the Preferences dialog (as explained below). Once a threshold value is set, only Log Actions of equal or greater priority will execute. For example, if Log Action A has a priority setting of 4 and Log Action B has a priority of 9, and the threshold setting in the Preferences dialog is 8, then at runtime only Log Action B will execute. Log Action A will be ignored.

NOTE: The reporting level can also be adjusted after deployment of your project, via the exteNd Composer Enterprise Server console screen. Consult your Composer Enterprise Server documentation for details.

➢ **To set the reporting threshold for logging:**

1 Go to the **Tools** menu, then choose **Preferences**. The Preferences dialog appears.



2 In the General tab, set the **Log Threshold** to a value from 1 to 10. The value you set here is a *threshold value*, which means that only Log Actions with a priority equal to or greater than this value will execute.

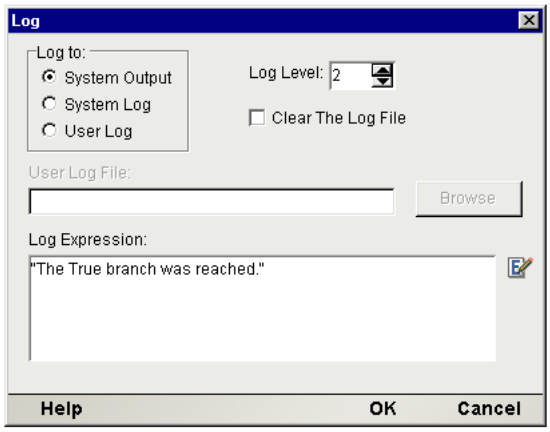
3 Click **OK** to dismiss the dialog.

➢ **To create a Log action:**

1 Open a component.

2 Select a line in the Action Model.

3 From the **Action** menu, select **New Action**, then **Log**, or press **Ctrl-L**. The Log dialog appears.



4 In the **Log to** radio group, choose the location to which you want messages written. (See explanation of locations further above.)

5 Use the **Log Level** spin control to select a priority level (1 to 10) for this Log action. The default is 5. In general, you should assign high numbers to messages with high importance. The priority you assign here will be ompared to the threshold number you chose in the last section (see further above). If the priority is equal to or greater than the threshold, the message is logged; otherwise it is not.

6 Enter a String or ECMAScript expression in the **Log Expression** text field. (You can use the Expression Builder—accessed by clicking the small icon to the right of the text field—to build an expression via pick-list selections).

7 Check **Clear the Log File** if you want the data in the log file to be cleared each time the component is executed.

More information about log files can be found in "Viewing System Messages" on page 69.

The Map Action

The Map action is a DOM-node input/output mapping. It transfers (and optionally, transforms) data from one document context to another document context. A context has two parts. The first part usually identifies a DOM and the second part identifies a location within the DOM. The basic document context in Composer is expressed as a DOM name combined with an element location (referred to as a location) identified through an XPath expression. The DOM name is usually Input, Input1, Input(n), Temp, Output, or any named DOM you have loaded in the component. The XPath expression identifying a location in a DOM has the path elements delimited by "/".

NOTE: A context in Composer can also be a Group name that itself is simply an alias or short-hand for an XPath expression.

About XPath and ECMAScript Expressions

When you create a Map action, you can choose between two methods for addressing locations in XML Documents: XPath and ECMAScript. The default choice is XPath, and it is the basic method of addressing.

The Basic Method: XPath by Itself

The primary purpose of XPath is to address or locate parts of an XML document (i.e., elements and attributes). XPath also provides basic facilities for manipulation of strings, numbers and booleans through a simple expression syntax. XPath addresses message Part nodes, including element nodes, attribute nodes and text nodes.

XPath is based on pattern matching. You specify a pattern of element names that resolve to the nodes in the target document. Most of the time, XPath returns a *node list* containing the particular nodes that match your pattern. (Many XPath expressions return only one node, but it is very common to return multiple nodes.) Other times, XPath can return a primitive value (string, number, or boolean).

In all Composer dialogs that take an XPath expression, you can build the expression with the aid of pick-lists in an Expression Builder. (See "To build an expression using ECMAScript:" further below.)

The complete XPath specification can be seen at **http://www.w3.org/TR/xpath**.

NOTE: The XPath spec is also available under the \Doc directory of your Composer installation.

The Alternative Method: XPath within ECMAScript

The second method to address locations in DOMs is to use ECMAScript with XPath. Choose this method if you wish to go beyond strict XPath addressing. ECMAScript is an object oriented scripting language for manipulating objects in a host environment (i.e., Composer). ECMAScript (ECMA-262 and ISO/IEC 16262) is the standards-based scripting language underpinning both JavaScript (Netscape) and JScript (Microsoft). It is designed to complement and extend existing functionality in a host environment such as Composer's graphical user interface. As a host environment, Composer provides ECMAScript access to various objects (including DOM objects) for processing. ECMAScript in turn provides a Java-like language that can operate on those objects.

Composer's built-in ECMAScript interpreter recognizes a custom Composer method called XPath(). It allows expressions such as:

```
Input.XPath("Inventory/Books/Engineering")
```

Construction of this type of expression is greatly facilitated by the user of Composer's Expression Builder facility. (See "To build an expression using ECMAScript:" further below.)

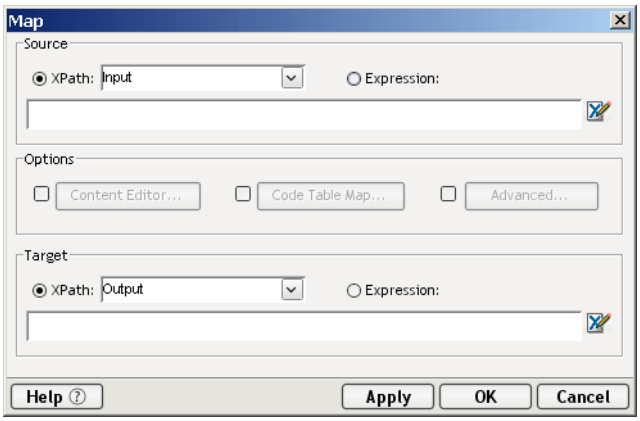
Adding a Map Action

➢ **To add a Map action:**

1 Open a component.

2 Select a line in the Action Model where you want to place the Map action. The new action is inserted below the line you selected.

3 From the **Action** menu, select **New Action**, then **Map**, or press **Ctrl-M**. The Map dialog appears.



4 The **Source** type is XPath. Select a Part (Input, Output, or Temp) from the pulldown menu, then type the appropriate XPath expression, locating the element you want.

NOTE: Alternatively, you can click the **Expression Builder** to have Composer assist you in building the XPath expression. See "Using the XPath Expression Builder" on page 143.

Together, the Part name and XPath specify the Source context for the Map action.

5 Repeat steps 4 and 5 for the **Target**.

6 Under **Options**, in the middle of the dialog, check **Content Editor** and/or **Code Table Map** and/or **Advanced** to exercise finer control over the mapping.

NOTE: More information on the **Content Editor** and **Code Table Map** option is available in "Transforming Elements" on page 283. A discussion of **Advanced** options appears below. Note that you will

7 Click **OK**. The Map action appears in the Action Model pane as shown.

NOTE: You can press the Apply button to see the affect of the Map action without closing the dialog. This allows you to make repetitive edits to a Map action and quickly see the results.

Default Mapping Behavior

When you use the Map action to map elements and attributes within XML Documents, certain default behaviors occur. The following table lists those default behaviors.

Map Type	Default Behavior
Leaf Element to Leaf Element	Transfers the element data only.
Leaf Element to Branch Element	Transfers the element data only.
Branch Element to Leaf Element	Transfers the entire branch including all child elements and attribute data under the branch.
Branch Element to Branch Element	Transfers the entire branch as above after removing the target's current branch.
A particular Leaf Element in a list of Leaf Elements, to Element	Transfers the element data from the selected leaf (or element instance) to the target element.
Attribute to Attribute	Transfers the attribute data only.
Element to Attribute	Transfers element data to attribute data.
Attribute to Element	Transfers the attribute data only.

Many of these behaviors can be altered, on an action-by-action basis, through the use of options exposed in the Advanced mapping dialog (see next section).

Leaf Elements that Contain Markup

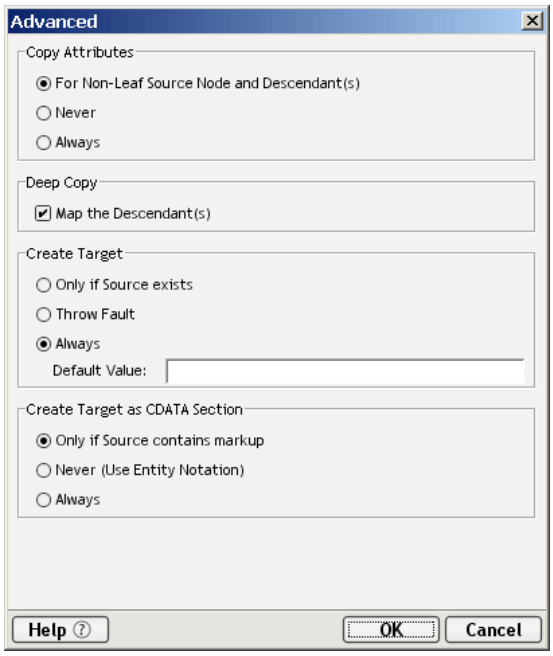
A special situation can arise when an element is populated at runtime by a Java or ECMAScript operation. It's possible that the element might receive data that contains markup—in other words, strings with illegal characters, such as < and >. This presents a mapping challenge, in that if Composer were to merely map the raw contents of such an element, unchanged, to a node in the Output DOM, the output document would be malformed.

Composer resolves this issue by mapping any data that contains markup to a CDATA section created on-the-fly in the target document.

NOTE: A somewhat different behavior applies at design time, when markup is entered by hand. At design time, if you type markup data into a node (via right-mouse-click/**Edit Data**), the markup characters are entitized on the fly. If you examine the raw XML in Text View, you'll see that any '<' characters entered by hand are converted to `<` (and so on). The entitized data are then mapped directly to output.

Advanced Mapping Options

When the Advanced checkbox is checked in the Map Action Dialog, the following dialog appears. Note that the options you set in this dialog affect only the current Map Action; not subsequent ones.



The options in this dialog give you finer control over how input Part nodes are mapped to the output Part.

Copy Attributes

This grouping of controls allows you to specify how attributes are mapped. Three radio buttons appear under this grouping.

- **For Non-Leaf Root Nodes and Dependents**—This button, checked by default, represents the standard (default) mapping behavior of Composer: When a non-terminal (non-leaf) element is mapped to output, the element—minus its attributes—and its children are mapped to output. Attribute data for the children are included, but not for the original (parent) element.
- **Never**—This option means *no* attribute data (whether for parent or leaf nodes) will be carried over during mapping.
- **Always**—All attribute data, for all nodes, will be mapped to output.

Deep Copy

By default, Composer maps whole branches at a time (that is, the target node plus all of its children). In some cases, you may want to turn off this “deep copy” behavior so that you can copy just the parent element without its children. Uncheck the checkbox labelled “Map the Dependents” if you want to disable Composer’s standard deep-copy behavior.

Create Target

The Create Target option allows you to optionally create the destination node (or branch) that you specified under Target in the Map Action dialog, based on whether or not the source node (or branch) is present in the source DOM. The default behavior is that Composer always creates the target, whether or not the runtime source DOM contains the node(s) that you specified in the Source XPath for mapping.

For example: In the Map Action dialog, you may have specified a Source XPath that looks like

```
$Input/Root/MySourceElement
```

while under Target, you may have specified something like

```
$Output/Root/MyParentNode/SomeOtherElement
```

If the arriving Input document *doesn't have* a node corresponding to `Root/MySourceElement`, Composer will (by default) nevertheless create an empty `Root/MyParentNode/SomeOtherElement` node in the output DOM. In some cases, this might not be what you want. Using the radio buttons in the Advanced Mapping dialog, you can change the default behavior.

NOTE: The Create Target options are disabled if Code Table Map was selected in the Map Action dialog.

The options under this radio button grouping are:

- **Only if Source Exists**—This means that the Map Action will simply be skipped (no target nodes created in the output DOM) if the node specified in the Source XPath doesn't exist in the input document.
- **Raise Error**—If the input document doesn't contain the node specified in the Source XPath, it will be considered an error at runtime, if this button is selected. You should plan accordingly by wrapping your Map Action in a Try/OnError block so you can handle the error.
- **Always**—Default behavior. (Target node is always created.) When this button is selected, the nearby Default Value text field becomes enabled so that you can optionally enter a default data value for the target element.

Create Target as CDATA Section

This radio-button group allows you to control the way element data gets mapped into CDATA sections. The options are:

- **Only if source contains markup**—This choice means that if the source data contains XML tags, HTML tags, or other types of markup where "illegal" characters are used, the data will be placed, unmodified, in a CDATA section in the target DOM. This is the default behavior of Composer.
- **Never**—With this option set, source data is guaranteed *not* to be wrapped in a CDATA section for output. Any illegal characters that occur in the source data will be converted to properly escaped entities, such as **>** for >, on the output side.
- **Always**—This means that whatever form the source data might take, it will get wrapped in a CDATA section on output.

Using the XPath Expression Builder

When you are in the Map Action dialog, you can build your own XPath expressions by choosing the Expression Builder button at the far right of the appropriate text field. The XPath Expression Builder dialog that appears will display pick-lists to help you construct valid XPath syntax in point-and-click fashion. This can be especially handy when you wish to go beyond basic XPath addressing and use some of the more powerful features of XPath.

exteNd Composer uses the XPath addressing syntax adopted by W3C. The XPath syntax is similar to URI address syntax in basic appearance but includes many subtle and powerful features for addressing and manipulating XML. Some of the more common syntax rules are listed in the following table.

XPath Syntax	Description
/	The single forward slash represents an absolute path to an element. /ABC selects the root element ABC.
//	Double slashes represents all elements in a path. //ABC selects all occurrences of ABC. //ABC//DEF selects all DEF elements which are children of ABC.
*	The asterisk selects all elements located by the preceding path. *ABC/DEF selects all elements enclosed by elements ABC/DEF. //* selects all elements.

XPath Syntax	Description
[ ]	Square brackets specifies a particular element. /ABC[3] selects the third element in ABC. This can also be used as a filter (similar to a Where clause in SQL). //ABC[“Table”] selects all elements that have the content “Table.”
@	The At sign selects elements with a specified attribute. /ABC@name selects all elements in ABC that have an attribute called name.
|	The vertical bar allows you to specify multiple paths. //ACB|//DEF selects all elements in ACB and in DEF.
$	The dollar sign allows you to reference other documents besides the current one. INVOICEBATCH/INVOICE[SELLER/NAME= $PROJECT/USERCONFIG/COMPANYNAME]
function()	XPath has numerous functions that you can add to your XPath addresses. For instance, //*[count(*)=2] selects all elements that have two children.
math operator()	XPath has numerous math operators that you can add to your XPath addresses. For instance, /ABC|position() mod 2 = 0] selects all even elements in ABC.

The complete list of operators can be seen at **http://www.w3.org/TR/XPath**.

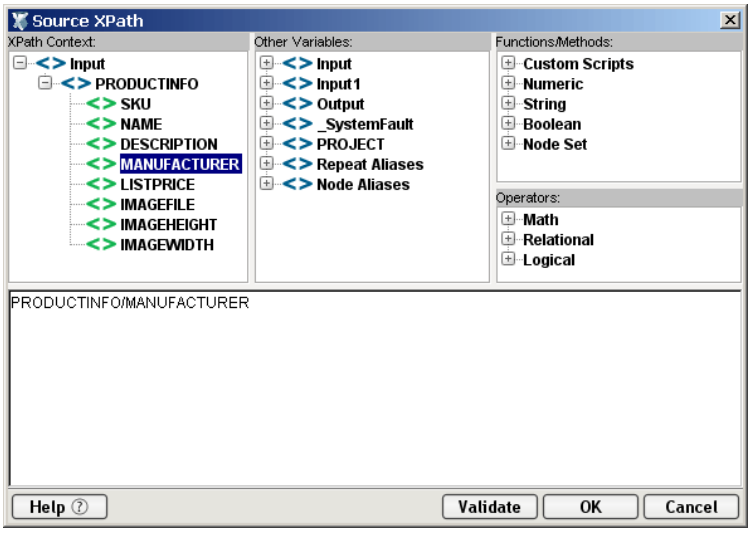
➢ **To build an expression using XPath**

1 Open a component.

2 Select the **Map** action from the **Action** menu.

3 Ensure that the **XPath** radio button is selected.

4 Click the **Expression Builder** button. The Source XPath dialog displays.



5 Create an expression by doubleclicking on the items from the panes.

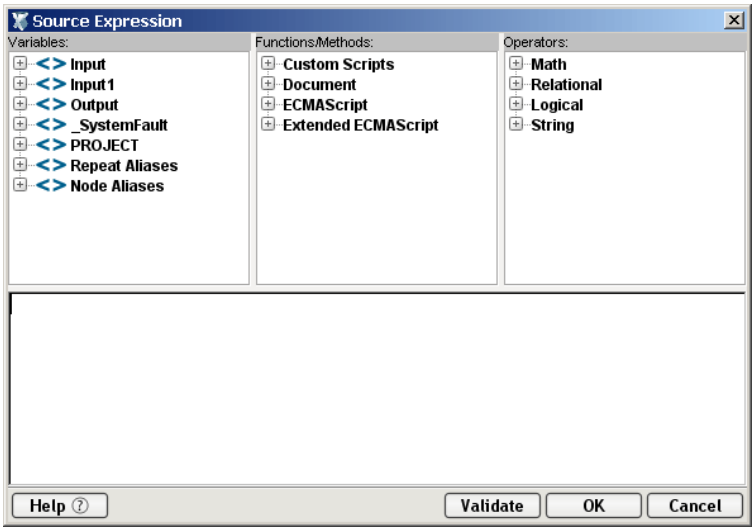
6 Verify that your expression’s syntax is correct (using the **Validate** button).

7 Click **OK**.

Using the ECMAScript Expression Builder

When you select the ECMAScript radio button in the Map action, the ECMAScript Expression Builder appears and helps you construct valid ECMAScript syntax. This is desirable when you want to go beyond strict XPath addressing and use some of the more powerful features of Composer's ECMAScript addressing.

The illustration below shows the ECMAScript Expression Builder.



Objects in the pick-lists are ordered with most frequently used objects first. All properties and methods for an object are also ordered. Properties are always listed first alphabetically, followed by all the object's methods alphabetically.

All of the items in the **Functions/Methods** pick-list and the **Operators** pick-list have tool tips associated with them. To view a tool tip, simply hover your cursor over the item you'd like to know more about. If you hover your cursor over the items in the **Variables** pick-list, data associated with that item will be displayed.

NOTE: While you can create complex ECMAScript expressions, they must evaluate to a document context consisting of a DOM and an address within the DOM.

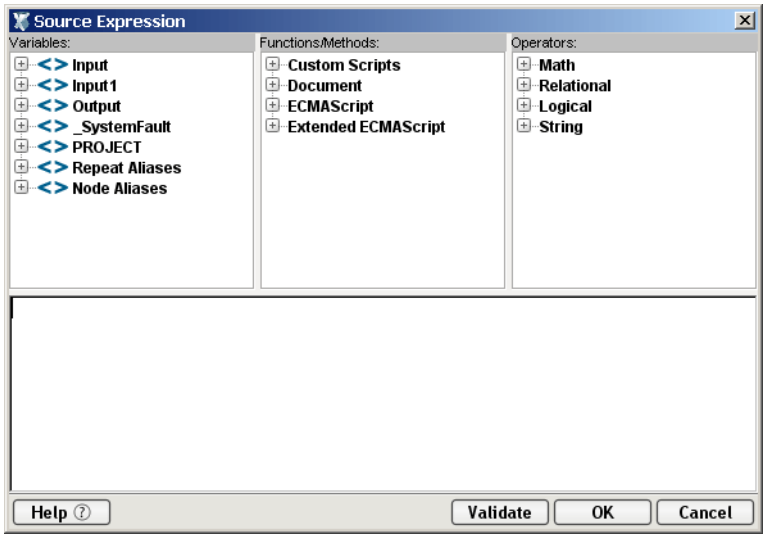
➢ **To build an expression using ECMAScript:**

1 Open a component.

2 Select the **Map** action from the **Action** menu.

3 Select the radio button next to **Expression**.

4 Click the **Expression Builder** button. The Source Expression dialog displays.



5 Create an expression by doubleclicking on the items from the panes.

6 Optionally click **Validate** to verify that your expression's syntax is correct. (This does not execute the expression. The expression is merely parsed.)

7 Click **OK**.

The Send Mail Action

The Send Mail action creates and sends e-mail messages dynamically during the execution of a component. When you create a Send Mail action, you specify the various parameters needed in order for Composer to know where and how to send the e-mail. The parameters can be hard-coded or (alternatively) ECMAScript expressions that evaluate at runtime.

Some possible uses of the Send Mail action include:

- Sending an "order status" notice to a customer after he or she has placed an order via the web.
- Triggering human intervention in a service that requires such intervention as part of normal workflow.
- Notifying system administrators (or others) of critical error conditions requiring immediate action. (The mail could even be routed to a pager or other mobile device.)

The e-mail you send with the Send Mail action can have attachments of any arbitrary MIME type. Also, various Send Mail actions can use various mail servers (with or without user name and password).

Mail via SMTP Simple Authentication

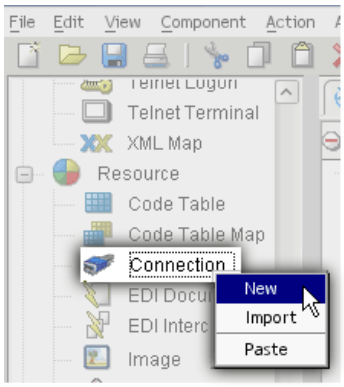
Although some in-house mail servers might not require a user name or password for outbound mail, many SMTP servers issue an authentication challenge before granting access. If your Send Mail actions will be using a mail server that requires user ID/password authentication, you will need to create a **Mail Simple Authentication** connection resource. This resource simply stores the network address for the mail server you want to use, along with a user name and password. The resource, once created, can be reused by any number of components and/or services within your project.

It's worth noting that you are *not* required to create one Mail via SMTP resource for each server (or for each user name and password combo) you intend to use. All parameters in the Mail via SMTP connection resource can be indirected through ECMAScript, so that server names or user credentials (or both) are late-bound— perhaps obtained by lookup from a directory or database, at runtime. Using ECMAScript, you can apply your own business logic to decide which mail server (or which credentials) to use in a given circumstance at runtime.

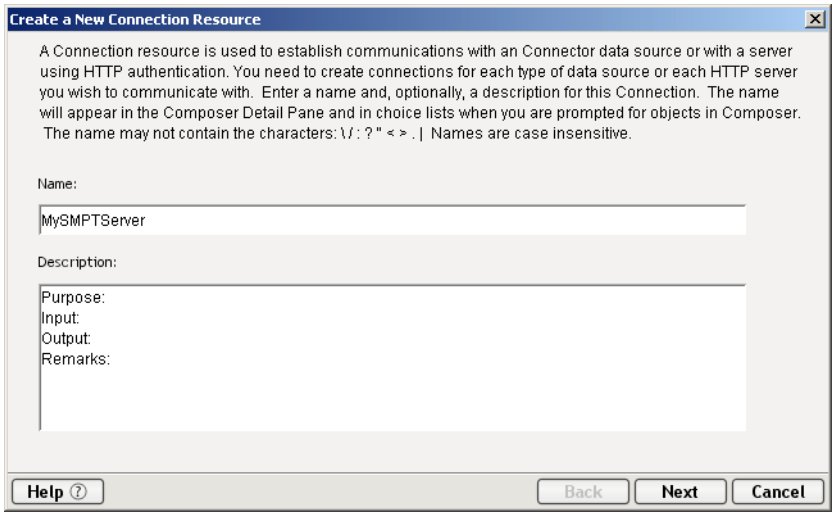
NOTE: See the discussion at "About Constant vs. Expression Driven Connections" (in the chapter on Resources) for additional information on how ECMAScript can be used for late binding of connection-resource parameter values.

➢ **To create a Mail Simple Authentication connection resource:**

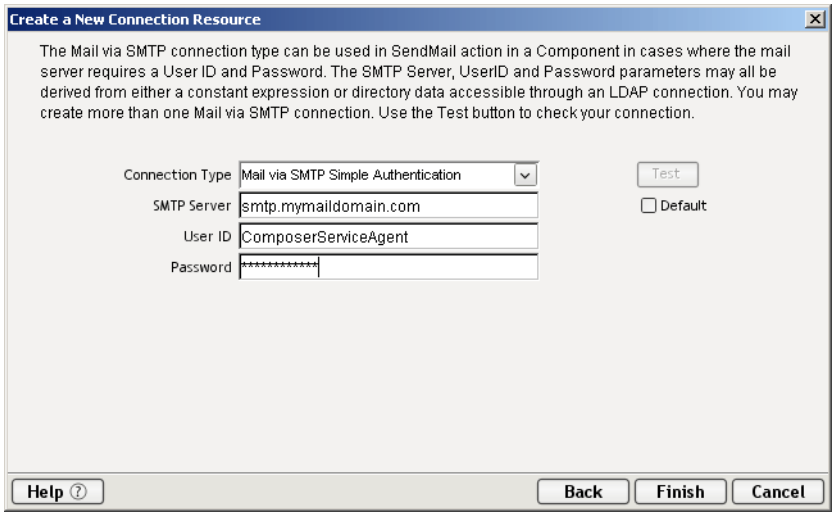
1 Under Resource in the navigation (explorer) frame, right-click on **Connection** and choose **New** from the context menu as shown below:



2 In the wizard pane that appears (see below), enter an arbitrary **Name** for this connection resource and (optionally) descriptive text.



3 Click **Next**. The second (and final) panel of the wizard appears:



4 Using the pulldown menu control, select Mail via SMTP Simple Authentication as the **Connection Type**.

5 Next to **SMTP Server**, enter the name or IP address of the mail server you intend to use.

6 Next to **User ID**, enter the user name associated with the mail account you wish to use.

7 Next to **Password**, enter the password associated with the user account in question.

NOTE: Again, note that any of these parameters may be entered as ECMAScript expressions. See the discussion at “About Constant vs. Expression Driven Connections” (in the chapter on Resources) for additional information on using ECMAScript here.

8 Click **Finish**.

How to Create a Send Mail Action

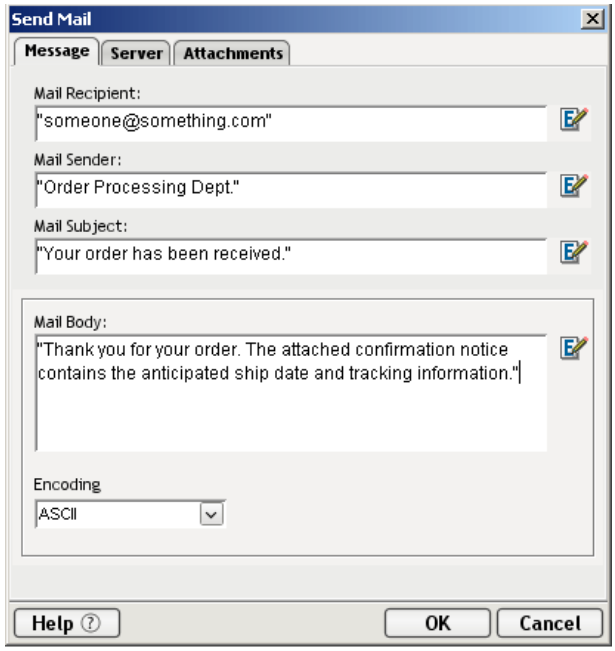
➢ **To create a Send Mail action:**

1 Open a Component.

2 Select a line in the Action Model where you want to place the Send Mail action. The new action will be inserted below the line you select.

3 From the **Action** menu, select **New Action**, then **Send Mail**. The Send Mail dialog appears. Note the presence of three tabs: **Message, Server,** and **Attachments**.

4 Select the **Message** tab if it is not already selected.



5 In the **Mail Recipient** field, type an ECMAScript expression to specify the e-mail address of a recipient. The expression should evaluate to a string of the general form `name@domain.extension`. If you are hard-coding a string value, make sure the text is enclosed in quotation marks.

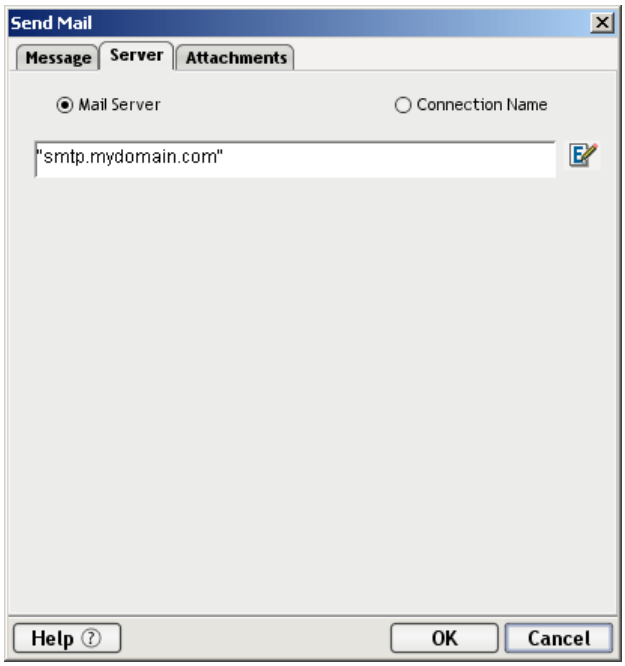
6 In the **Mail Sender** field, enter an ECMAScript expression to specify the string you wish to show as the sender’s e-mail address. (It can be any arbitrary string; it does not have to be an actual e-mail address.) Again, if you are hard-coding a text value, make sure the text is enclosed in quotation marks.

7 In the **Mail Subject** field, type a valid ECMAScript expression to specify the e-mail subject or type a subject line. Again, if you are hard-coding a string, make sure the text is enclosed in quotation marks.

8 In the **Mail Body** field, type a valid ECMAScript expression to specify the e-mail body text (or type the body text enclosed in quotation marks).

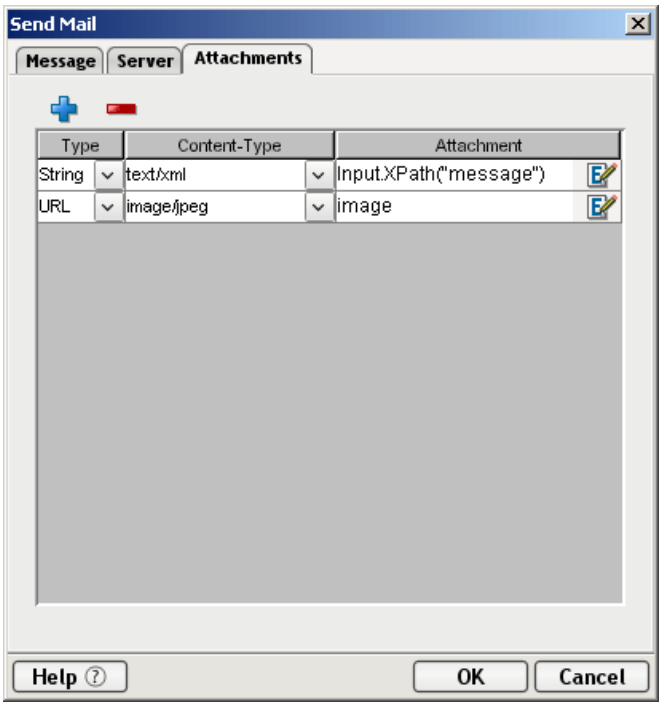
9 Under **Encoding**, specify (using the pulldown menu) the type of encoding your message should use. The default is ASCII.

10 Select (click on) the **Server** tab. The dialog changes appearance:



11 Click the **Mail Server** radio button if you wish to specify an ECMAScript expression that will resolve to your mail server's address (as shown above). Alternatively, click the **Connection Name** radio button if you wish to use a server that has been specified in a "Mail via SMTP Simple Authentication" connection resource. (See the discussion of this resource type at "Mail via SMTP Simple Authentication" earlier in this section.) You would use the latter option in cases where user authentication (via username and password) is required in order to access the server.

12 If you want to include attachments with the e-mail, click the **Attachments** tab. (Otherwise, click **OK** to return to the component editor.) The dialog changes appearance:



13 Click the **plus-sign (+)** button to add an attachment.

14 Under **Type**, specify *String* or *URL* (using the pulldown menu control).

- Specify *String* if you want the value in the Attachment column of the table to be the (literal) attachment to the e-mail.
- Specify *URL* if you want to indirect the attachment target through a URL (using file: or http: protocol schemes).

15 Under **Content-Type**, specify the MIME type of the attachment. You can either choose from the MIME types shown in the dropdown menu, or you can enter your own MIME type in the editable field.

16 Under **Attachment**, enter the ECMAScript expression that will serve as the attachment content (if you chose *String* under Type) or as the URL to the file you wish to send. In the example above, the first attachment is a String consisting of the data associated with the **/message** node of the component's Input document. The second attachment (a JPEG image) specifies a URL string contained in the previously declared ECMAScript variable named "image." The variable in question could resolve at runtime to something like **"file:///d:/server-1/resourcestore/images/stockimage.jpg."**

17 Click **OK**. A new Send Mail action appears in the action model of your component:

The Switch Action

The Switch Action (inspired by the Java and C-language switch statement) is designed to allow your application to branch to the appropriate custom logic based on the value of a particular input variable or XPath expression. The Switch Action is a convenience action that obviates the need for a series of nested Decision Actions. It increases the readability of your action model significantly by eliminating multiple actions and consolidating them into one coherent, easily documented, easy-to-read action.

About Cases

The Switch Action compares a series of values or choices ("cases")—which may be either static *or* dynamic—against an input value. If an exact match occurs between the input value and one of the available choices, execution branches to the action(s) listed underneath the choice. Just as with a series of if/else statements, cases are tested *in the order listed*; and once a match is found, execution of the match logic precludes execution of any other logic in the Switch Action.

The custom logic associated with any Case can consist of a single action *or* a block of actions; and the actions can include any of the standard (basic or advanced) Composer actions, as well as actions specific to a particular Connect.

A Switch Example

Suppose your incoming XML document represents a retail order for goods, and one of the tasks your application must perform is the determination of a shipping method based on the customer's location. The input to the Switch Action might be an XPath expression like:

```
$Input/Order/Customer/Address/Country
```

The case values for the Switch Action, and the associated logic for each choice, might look like:

```
CASE:"USA"
   CALL shipMethod = (weight < 10) ? "FedEx" : "UPS";
CASE:"ANGOLA"
   CALL shipMethod = "Air Gemini";
CASE:"ARGENTINA"
   CALL shipMethod = "International First Services";
CASE:"AUSTRALIA"
   CALL shipMethod = "Ansett International";
.
.
.
DEFAULT:
   CALL shipMethod = "UPS";
```

At runtime, the value of the Input DOM element at `Order/Customer/Address/Country` will be checked against each successive Case value, starting with "USA," until a match is reached. In this example, if the match occurs at "ANGOLA", the Function Action that assigns "Air Gemini" to the (ECMAScript) variable `shipMethod` will execute, then the Switch Action will exit immediately, and execution will continue with the first action (if any) *following* the Switch Action.

NOTE: No explicit Break action need be inserted in any Case action group, because the built-in "fall-through" behavior of Java and C-language case statement is *not* a feature of Composer's Switch Action. Once a match happens, fall-through to the next Case *never* occurs.

The foregoing example could be equivalently written as a series of Decision Actions. The pseudo-logic for the chain of Decision Actions would be:

```
country = inputValue
if (country == USA)
   ship via A or B
   else if (country == ANGOLA)
      ship via C
      else if (country == ARGENTINA)
         ship via D
         else if (country == AUSTRALIA)
            ship via E
[etc]
               else ship via Default shipper
```

The Switch construct eliminates the stairstep indentation and repetitive if/else logic that characterize this kind of code. It also results in easier-to-read-and-maintain code. In general, any time you are faced with a long series of conditionals, you should consider using a Switch Action.

About the Default Case

The final "Case" under every Switch action is always labelled Default. This line is generated automatically and cannot be removed. Actions placed under Default are executed if and only if the Switch Action, at runtime, encounters no matching Case in the list of Cases.

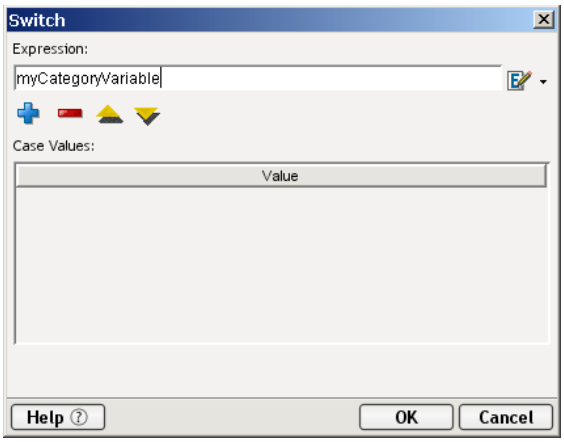
NOTE: While you are not *required* to place actions under Default, it is good programming practice to have at least *some* kind of fallback logic for the "no match" case, even if it's only a Log action or a Raise Error action.

➢ **To add a Switch action:**

1 Open a component.

2 Select a line in the Action Model where you want to place a Switch Action. The new action will be inserted below the line you selected.

3 From the **Action** menu, select **New Action**, then **Switch**. The Switch Action dialog appears. (The text values in the dialog shown below are not defaults. Values were entered for purposes of illustration only.)



4 Enter an XPath or ECMAScript expression in the top of the dialog under **Expression**. This is the input value to the Switch Action.

5 In the combo box, enter the static string values or the ECMAScript expressions that will be checked against the input value that you specified in the previous step. Remember that at runtime, each Case value will be checked in turn, in the order you list them. (Tip: For optimal performance, *list the most likely matches first*.)

NOTE: New Case entries are, by default, added to the *end* of the existing list. But you can change the order of the choices by highlighting a given choice and clicking the Up and Down buttons as need be.

6 Click **OK**. The dialog goes away and the new Switch Action appears in your action model. See example below.

Once you have added a Switch Action to your action model, you will see a list of Case values. To associate your own custom logic with a given Case, click on the Case, then add new actions one at a time as needed, by clicking the right mouse button and choosing New Action from the context menu. Your Actions block can contain any number of actions (of any type).

➢ **To add custom case-handling logic:**

1 In the action model, find the Case to which you want to add processing logic.

2 Click on the Actions line below the Case.

3 Right-click to bring up the context menu. Select **New Action** and pick from any of the actions available on the submenus.

4 Repeat the previous step as needed to add additional actions.

Editing Switch Actions

The primary tool for editing Switch Actions is the Switch Action dialog, which allows you to edit the input expression, reorder Cases, edit Case expressions, and add or delete Case values. To access this dialog, just doubleclick on any Switch Action within an action model.

Only a limited amount of editing can be done from the action model itself (without opening the settings dialog). The following limitations apply:

- Cut, Copy, Delete, and Paste operations on the Switch Action (top line) itself result in the entire Switch block, including all matches and associated Action lists, to be cut, copied, etc.
- You can Cut or Delete a Case value that has been selected in the action model, but you cannot *add* a new Case value (by pasting).
- A Cut or Delete operation will cut/delete *not only the Case itself but all associated actions.*
- Actions in Case action lists can be edited in the normal ways.

The Todo Action

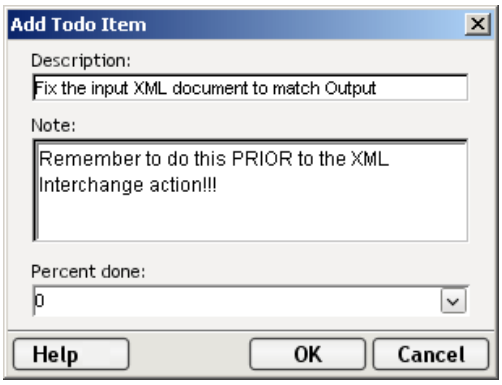
Developing Web Services and XML-integration applications can be a very complex undertaking. Composer provides the ability for you to maintain a Todo list to help you organize and manage the many tasks associated with application development.

➢ **To add a Todo action:**

1 Open a component.

2 Select a line in the Action Model after which you want to place your Todo list. The list item will be inserted below the line you selected.

3 From the **Action** menu, select **New Action**, then **Todo**. The Todo dialog appears.



4 Enter a Description for the item that will be displayed in your Todo list.

5 If desired, enter a Note containing additional information. This text displays as part of the item's tool tip when the mouse pointer is over the item.

6 Use the down arrow to select a Percent Done value for your task, or leave it at 0. As tasks near completion, you should edit this action item and update the percentage complete.

7 Click **OK** to add the item to your Action model.

Project-Wide Todo Lists

Todo Lists are not only available within components. They can also be associated directly with a project.

➢ **To add a Todo list outside of a component:**

1 Open a project.

2 Click on the **Todo** tab in the Message Frame (see “Navigation, Message, and Content Frames” on page 38.

3 Right-click with your mouse and select **Add Item** to add a new Todo list item. Create the item as indicated above.

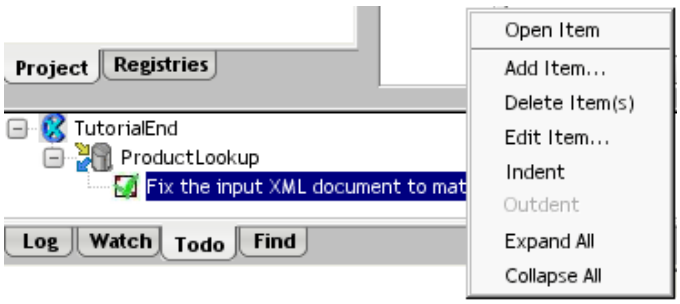
Tracking Todo items using the Message Frame tab

Once it has been added to your component or project, you will be able to track the progress of the Todo Item using the Todo tab on the Message Frame.

When viewing items in the Todo tab, you will be able to see at a glance how far along you are in your list:

- a blank checkbox indicates that the task has not begun
- a gray checkmark indicates partial completion
- a green checkmark indicates that the task has been completed

Todo items can be managed either from the Action Model or by right clicking on them in the Todo tab of the Message frame. Items can be edited, added and deleted and re-grouped in the list using the Indent and Outdent menu selections.

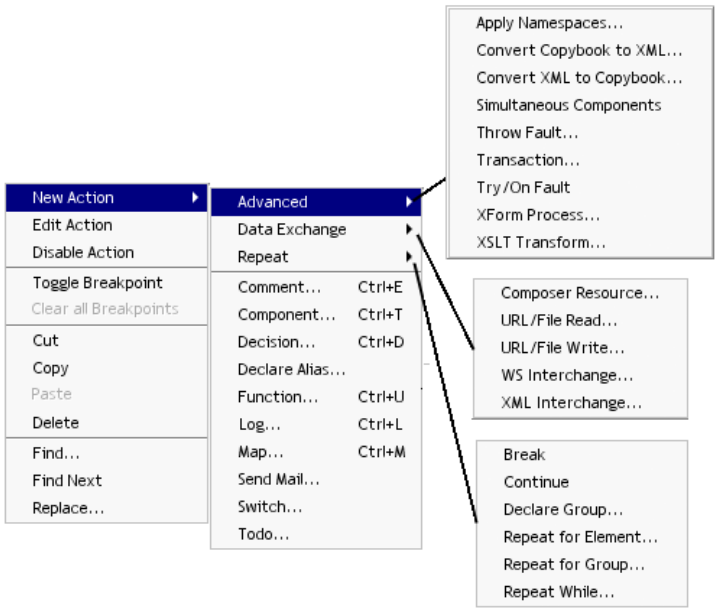


8 Advanced Actions

The previous chapter introduced you to the basic actions you can use when building components. The actions discussed in this chapter are of a more advanced nature than those discussed earlier. They include I/O-related actions, control-flow constructs, and miscellaneous additional actions.

The actions discussed in this chapter can be created using commands under the Action menu's nested submenus. The submenus include **Advanced**, **Data Exchange** and **Repeat**. They can also be accessed via right-mouse-click inside the action model.

The menu structure looks like this:



The table below summarizes the Advanced actions. (The Data Exchange and Repeat actions are discussed in their own sections further below.)

Advanced Action	Description
Apply Namespaces	Provides a way to override NameSpace prefixes, declare a new one or ignore a NameSpace altogether.
Convert Copybook to XML	Converts XML data into a ByteArray object using a COBOL Copybook
Convert XML to Copybook	Converts a ByteArray object into XML data using a COBOL Copybook
Simultaneous Components	Allows two or more components to be executed simultaneously (that is, in multithreaded fashion).

Advanced Action	Description
Throw Fault	Evaluates a condition which if true, writes the contents of an expression to a fault document. If used alone, it throws an exception, stops a component, and returns control to the service. If used within the Execute branch of a Try On Fault action, it is evaluated and control passes to actions in the On Fault branch.
Transaction	Allows you to invoke *User Transaction* commands (such as *begin*, *commit*, and *rollback*) in components that will be deployed as part of non-Container-managed services, or *setRollbackOnly* in components that will be part of Contained-managed EJB deployments.
Try On Fault	Responds to actions that produce errors by executing a set of actions depending on the type of Fault that occurs. The Try On Fault action is essentially an error trapping and solution action, and works in a fashion similar to Switch.
XForm Process	Allows you to preprocess an XForm document before mapping it to output
XSLT Transform	Transforms an XML file according to instructions in an XSL file. The output is commonly used for rendering XML files in the Web browsers.

NOTE: See Chapter 11, “Applying Actions to Common Tasks” for examples of using some of these actions.

Apply Namespaces Action

Ideally, a component will always receive valid XML documents (i.e. the documents validate against their schema), map and transform data appropriately, and send valid XML documents. But in the real world, this is not always the case. Therefore, it is important to have some means of validating XML documents.

Schemas combined with Namespaces provide a mechanism that allow validation enforcement. However, Schemas, Namespaces and Prefixes can easily become problematic when performing XML transformations. For simple straight-through processing involving document validation and marshalling, Composer's schema support, XML Template features, and drag and drop mappings mean you won't normally have to worry about managing Namespaces and Namespace Prefixes. But there are many cases involving document transformations where documents may need special treatment of Namespaces and Namespace Prefixes, such as when

- Business partners exchange valid documents belonging to the same Namespace but each uses a different Namespace Prefix. For one party to validate or work with the other's document, the Namespace Prefix of each partner needs to be declared in the document.
- An XML Template is not available to resolve a Prefix to a Namespace for a document (i.e. the Input XML Template is System {Any}). Yet for Map actions to work properly, the Prefixes used in the Map Source and Target need to be resolvable to a Namespace.

And there are still other cases where you simply wish to ignore Namespaces altogether. These and many other XML processing cases require a method of modifying or overriding the Prefix and Namespace handling provided by Composer's default Schema and XML Template support.

The *Apply Namespaces Action* provides a mechanism for managing Namespaces and Namespace Prefixes in effect for XML documents within a component’s Action Model. The action allows you to consolidate all your Namespace and Prefix declarations for a document in one place as well as override those declared in the XML Templates used by the component, or ignore Namespaces altogether.

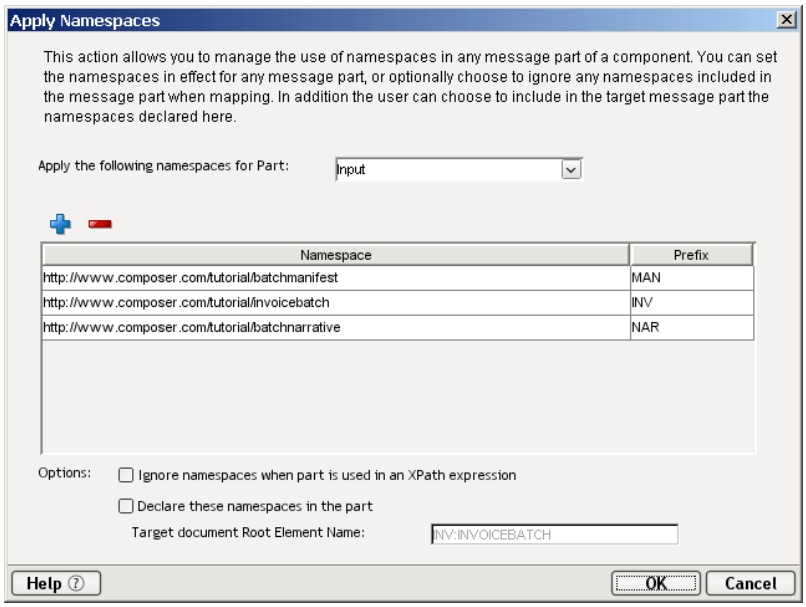
The Apply Namespaces action can be applied to any Message Part (Input, Input1, Temp, Temp1 or Output.) You may also have multiple Apply Namespaces actions for a single Message Part, effectively changing Namespaces in effect based on conditions specified in your Action Model. The Namespaces declared for any part will be in effect until the end of the Action Model is reached or another Apply Namespaces action for that Part is executed. In other words, only the most recent Apply Namespaces action is in effect for any single Part.

When creating a new component, an Apply Namespaces action is created automatically for the Output Part if it's XML Template declares any Namespaces. After component creation, you can manually create additional Apply Namespaces actions for any or all Message Parts. In both cases, the Namespaces and Prefixes initially specified when you first open the action dialog, are pulled directly form the XML Template. You can then add, change or delete Namespaces and Prefixes as needed within the action.

➢ **To Create an Apply Namespaces action:**

1 Open a component that you want to apply the Namespace action.

2 From the Action menu, select **New Action**>**Advanced**>**Apply Namespaces**. The Apply Namespaces dialog box appears as below.



3 Select from the dropdown list, **For Part**, where you want to apply the NameSpace (i.e. Output). This control displays available Message parts to which the list of Namespace declarations can be applied.

4 Click on the (+) icon to add a row, conversely, click on the (-) minus icon to delete a row. When adding a **NameSpace**, enter the URI and **Prefix** in the columns displayed.

NOTE: The Prefix table displays all the Namespace declarations in effect for the document displayed in For Part control. After creating a new Apply Namespaces action, the table may or may not contain a list of declarations for a selected Part. The list of declarations is initially constructed from the declarations defined in the XML Template's Namespace Declarations panel. If the XML Template for the Part is System{Any} or not Schema based, then the list will be empty, unless declarations have been added in the Template's Namespace Declarations panel.

NOTE: Within the declaration list for a single Message Part, the Prefixes must be unique. However, you are allowed to have duplicate Namespace URI entries provided they are associated with unique Prefixes. This allows you to declare multiple Prefixes that are associated with the same Namespace URI.

5 **Options**: Click in the checkbox to **Ignore Namespaces** when document is used in a Map action Source option when you want Map Action Source XPaths to find elements by their XML local name only.

NOTE: This provides for a less restrictive method of specifying Map actions and is useful when Map actions under some processing circumstances may contain the wrong or no Prefixes in their Source specifications. This allows you to put the Apply Namespaces action inside a Decision action that tests whether the Input Message contains Prefixes or not yet still have one set of Map actions to Map the Input to another document. In other words, the component normally expects the Input to contain Prefixes so you design all your Map actions with Prefix names. For the occasional Input that has no Prefixes, the Decision action activates the Apply Namespaces action defined to ignore Namespaces for Input allowing the Map actions to work in either case.

NOTE: This option performs the same function as the setSkipNameSpaces() method available for any Part (i.e. Input.setSkipNameSpaces(true)). Between this method and the Apply Namespaces action, whichever was executed last in an action Model will be in effect.

6 **Options**: Click in the checkbox to **Declare These NameSpaces** when document is used in a Map Target when you want to declare a set of Namespaces in the root element of an Output document built by your Action model. This option is almost always checked for Output to insure that prefixed elements created in the Output, as a result of Map actions, will resolve to the proper namespaces.

NOTE: This allows a recipient of the Output to validate the document properly. The Apply Namespaces action with this option checked could also be used to add new declarations to an existing document that already contains declarations.

7 Target Document Root Element Name specifies the name of the root element to contain the Namespace declaration attributes. If the target Message Part is based on a XML Template with Schema validation, then this control will be filled in automatically by Composer. If the target Message part is not an XML Template with Schema validation (e.g. System{Any}), then you must enter a value.

8 Click **OK** and the new action will be added to the Map Action Pane in your component.

Map Actions, XML Templates, Namespaces, and Prefixes

XML Templates and the Namespaces and Prefixes in XML documents processed by a component may all have an impact on whether a Map action works as expected. By default, for a Map action to work, the prefix / element name combinations in the Source XPath are expanded to their full names. A similar process occurs in the Message Part referred to by the Map action. If a match is found between the Source specification and the Message Part, the data or content model is mapped to the Target of the Map action. The most critical factor is whether Prefixes are expanded to their Namespace when a Map Action's Source is compared to an XML Message Part. If Namespace resolution is not performed (i.e. turned off) then Map actions will always work.

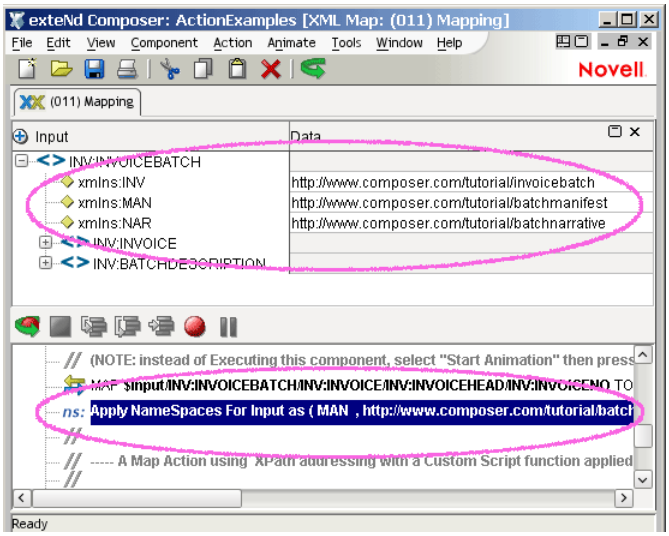
By default, Composer performs Namespace resolution. There are however, two ways to prevent Namespace resolution in Composer. The first technique is to use the `setSkipNameSpaces()` method for a Part as in `Input.setSkipNameSpaces(true).` The second technique is to add an Apply Namespaces action and check **Ignore Namespaces** when document is used in a Map Action Source control.

When Namespace resolution is performed, two additional conditions must be met in order for a Map action to work. The Prefixes used in the Map action and the Prefixes present in the document must: 1) be resolvable to Namespace URIs, and 2) the Namespace URIs must match. The first condition is a pre-requisite for the second.

The first condition requires that the Prefixes used in the Map Action Source (what you expect to receive) and the Prefixes used for elements in the runtime document (what you actually receive) must be expanded and resolvable to Namespace URIs. If either cannot be resolved, then the Map action fails. In order for a Map action to work, the expanded form of its Source specification Prefixes must match the expanded form of an element in the XML document being mapped (the second condition). The Map action Prefix is expanded by resolving it to a Namespace URI specified in the XML Template or in an Apply Namespaces action. Prefixes for the element in the XML document are expanded by resolving to a Namespace URI declared in the XML document (i.e. an **xmlns:someprefix="someURI"** attribute in the root element). If the expected Namespace URI of the Map action does not match the actual Namespace URI from the document, the Map action will fail.

Example: Assigning Namespace Declarations to Output Messages

When a new component is created and its Output Message is based on an XML Template containing Namespace declarations, Composer automatically adds an Apply Namespaces action to the Action Model. When the component executes, this action creates the appropriate root element, Namespace Prefix for the root element, and root element Namespace declaration attributes in the Output XML Message. Normally, this action is appropriately left at the start of the Action Model. In addition, the action allows you to add new Namespace declarations to the Output Message that are not declared in the XML Template. The following graphic shows how a Declare Namespaces action is defined for a typical Output message.



If a component or program that receives this component's Output is designed to work with the same Namespace but uses different Prefixes, you can use the Apply Namespaces action to add an alternate Namespace Prefix in the Output Message. Simply open the Apply Namespaces action and press the Add button. Copy the Namespace URI you wish to associate with another Prefix and paste it into the new line. Then specify the alternate Prefix.

NOTE: Note: When a Temp document is going to be used as the target of Map actions, you need to define a similar Apply Namespaces action for it. Since Temp documents can be both a Source and Target for a Map action, Composer does not know your intentions and so does not create the action automatically for you.

Example: Ignoring Namespaces

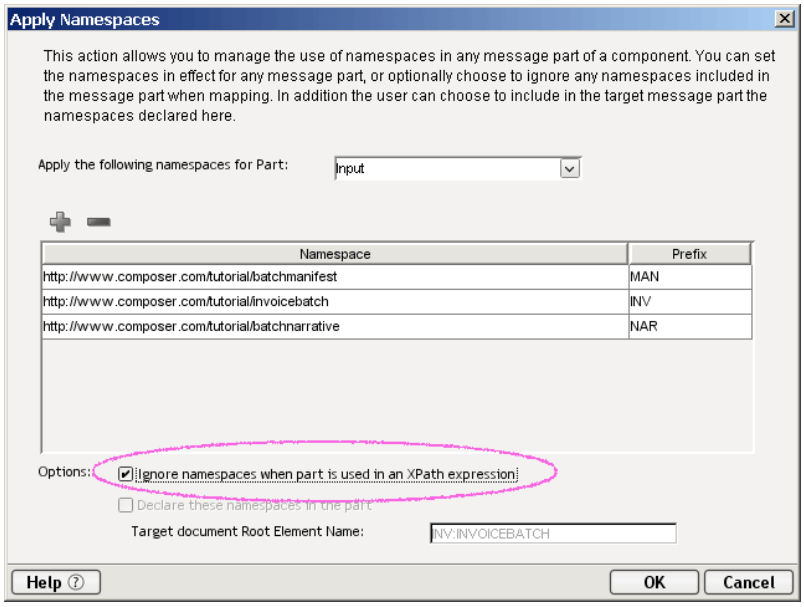
In some cases namespaces and their associated Prefixes are irrelevant to the mapping or transformational purposes of the component. Perhaps, a document has already been validated but needs to be re-structured before being inserted into a back end data store (e.g. a relational database via a JDBC component or a CICS transaction via a CICS/RPC component). In this case the Map actions are concerned only with re-structuring the Input XML message into a different hierarchy which would be much easier and quicker to design by referencing local names only in the document. In this case, an Apply Namespaces action can be added that ignores Namespaces altogether. This allows you to construct Map actions that omit any Namespace Prefixes in the Source XPath's you define. So instead of expressing the Source of a Map action as

```
INV:INVOICEBATCH/INV:INVOICE/INV:INVOICEHEAD/INV:INVOICENO
```

you can write:

```
INVOICEBATCH/INVOICE/INVOICEHEAD/INVOICENO
```

which is also more readable.



The Convert Copybook to XML Action

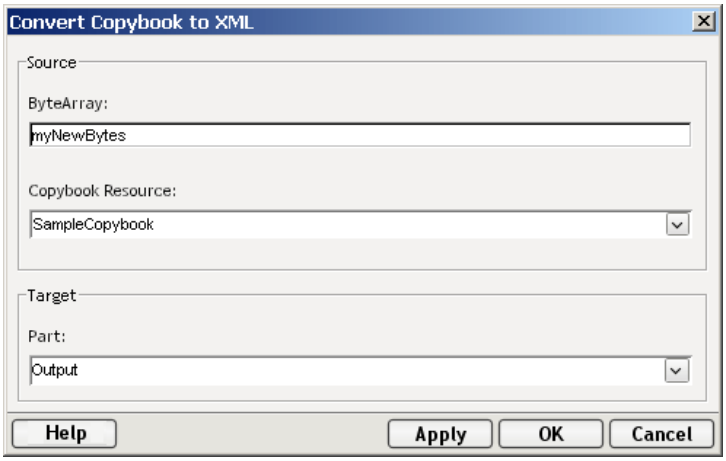
This action to converts a ByteArray into XML data using a COBOL Copybook Resource so as to map COBOL fields in the ByteArray to XML elements. The XML can then be used like any other XML inside the component.

➢ **To create a Convert Copybook to XML Action**

1 Open a component.

2 Select a line in the Action Model where you want to place a Convert Copybook to XML action.

3 From the **Action** menu, select **New Action>Advanced**, then **Convert Copybook to XML**. A dialog window appears:

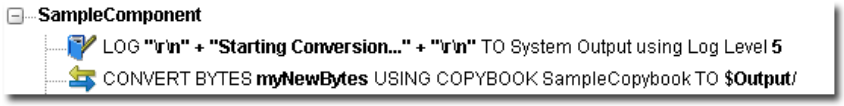


4 Under Source, type in the name of an existing **ByteArray** whose data you would like to convert to XML format.

5 Select a previously defined **Copybook Resource** (see “About Copybook Resources” on page 215).

6 Under Target, select an XML Message **Part** to be used to receive the converted ByteArray.

7 Click on **Apply** to see the results of your action, or click on **OK** to finish creating the new action and it it to your action model.



NOTE: Experienced CICS RPC users will recognize that this action performs the same function as the Auto Map Copybook feature available in the CICS RPC Component Editor. The only difference is that no Map actions are created for the user. In order for the mappings performed by the Convert action to work, the user must have a properly formatted XML document that accurately represents the structure of the Copybook. Creating an XML Sample for this is easy inside a CICS RPC component or JMS Component. Simply use Auto Map in the CICS RPC component to create an XML Template which can then be used as the target for this action. Refer to the *CICS RPC Component Editor User’s Guide* for more information on this topic.

The Convert XML to Copybook Action

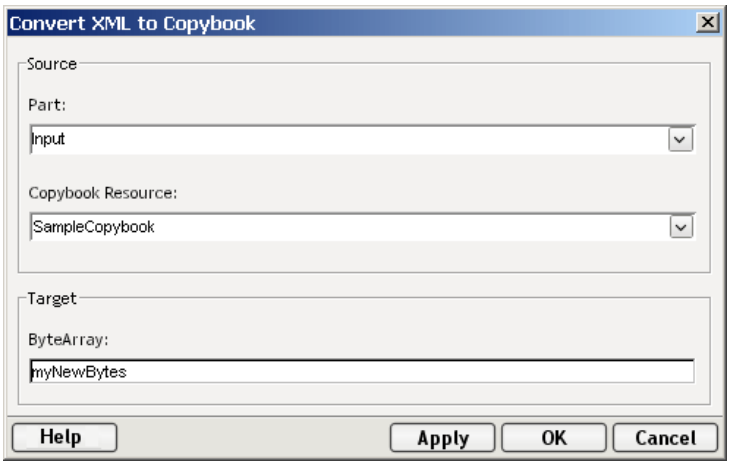
This action converts XML data into a ByteArray object using a COBOL Copybook Resource to properly map XML elements to COBOL fields in the ByteArray. The ByteArray can then be used directly by an ECI Execute action in a CICS RPC component or, perhaps, by a JMS Send action whose Body Message Type is Copybook (JMS bytes). The created ByteArray object can then be published globally using the Extended ECMAScript Component method named exportObject() making it reference-able by other components by its name.

➢ To create a Convert XML to Copybook Action

1 Open a component.

2 Select a line in the Action Model where you want to place a Convert XML to Copybook action.

3 From the **Action** menu, select **New Action>Advanced**, then **Convert XML to Copybook**. A dialog window appears:

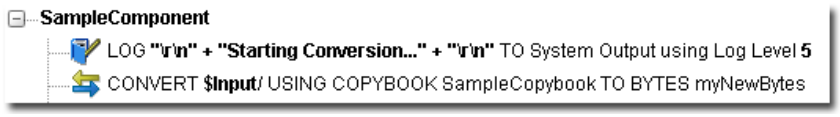


4 Under Source, select an existing Message **Part** whose XML data is to be converted into a ByteArray.

5 Select a previously defined **Copybook Resource** (see “About Copybook Resources” on page 215).

6 Under Target, type in name for the **ByteArray** to receive the converted data.

7 Click on **Apply** to see the results of your action, or click on **OK** to finish creating the new action and it to your action model.



NOTE: Experienced CICS RPC users will recognize that this action performs the same function as the Auto Map Copybook feature available in the CICS RPC Component Editor. The only difference is that no Map actions are created for the user. In order for the mappings performed by the Convert action to work, the user must have a properly formatted XML document that accurately represents the structure of the Copybook. Creating an XML Sample for this is easy inside a CICS RPC component or JMS Component. Simply use Auto Map in the CICS RPC component to create an XML Template which can then be used as the source for this action. Refer to the *CICS RPC Component Editor User’s Guide* for more information on this topic.

The Simultaneous Components Action

The Simultaneous Components Action allows you to execute two or more components simultaneously (which is to say, in their own separate threads of execution). This is an important capability to have in an XML integration application that relies on inquiries to legacy systems which might be relatively slow to respond. For example: Imagine that your service needs to retrieve information via CICS RPC and JDBC from two data sources. The CICS inquiry might have a round-turn time of five seconds and the JDBC inquiry might require four seconds. If the two inquiries are performed one after the other, the total time spent waiting for data would be nine seconds. But if both back-end systems can be queried at the same time, the total wait-time is cut to approximately five seconds. This is a significant performance improvement.

The Simultaneous Components Action places a “Simultaneous Components” header line in the action model, below which you can insert any number of Component (or other) actions.

In the above illustration, the action list under “Simultaneous Components” contains a call to a 3270 Component, a call to a JDBC Component, and a Send Mail action. The two Component actions will be spawned in separate threads. The Send Mail action will then be executed immediately (whether or not the 3270 and JDBC components have returned).

NOTE: You can include any type of Action (Map, Decision, etc.) in the list beneath a Simultaneous Components Action. But no action in the list should depend on return values from any Component actions, because Component Actions are not guaranteed to return before other actions in the block execute.

Downstream actions that are *outside* of the Simultaneous Components block *can* depend on return values from spawned components, because the Simultaneous Components action does not pass control to downstream actions until all spawned components have returned. Synchronization is guaranteed to occur, in other words, before execution continues beyond the Simultaneous Components block.

➢ **To create a Simultaneous Components action:**

1 Open a component.

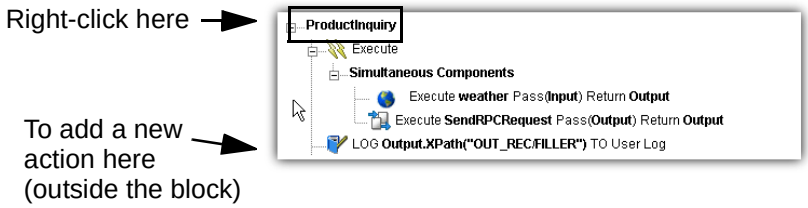
2 Select a line in the Action Model where you want to place a Simultaneous Components action.

3 From the **Action** menu, select **New Action>Advanced**, then **Simultaneous Components**. A Simultaneous Components header line appears in the action model (per the illustration above).

4 Place any number of actions below the header line. (Right-click on the header line and choose an action from the context menu, or Paste actions into the Simultaneous Components block.)

NOTE: No actions other than Components Actions will be spawned as new threads.

To place new actions outside of (downstream of) the Simultaneous Components block, right-click on the line above the Simultaneous Components header line, and choose a new action. The new action will be added below the Simultaneous Components block. See below.



The Throw Fault Action

The Throw Fault action allows you to write information to an XML message on failure of an action, perform any number of “Before Throw” actions, and finally halt execution of a component. Throw Fault is only executed when a condition that you specify is true. The Message Part that gets written when a Throw Fault action is executed is called a Fault document, and the XML within this message will also be contained in a global object called ERROR. For a discussion on Fault Parts, refer to “Creating a Fault Message Part” on page 116.

Throw Fault actions can be used in a number of different ways:

- **Using a Throw Fault Action by itself**. You can easily specify a Fault Condition and it's accompanying error message within the Throw Fault Action dialog. An example of this procedure is given below. When the action is executed, the Fault Condition is evaluated and if true the following occurs:
  - Any "Before Throw" actions you specify are executed. This can be very useful as a way to leave your application in a particular state before halting execution. You might want to, for example, send a mail message stating that the execution did not complete.
  - The contents of the Error Message are written to the Fault document in a node you specify, as well as to the global object ERROR.
  - The component execution is halted.
- **Using a Throw Fault Action within a Decision Expression in the Decision action.** You might want to specify your Fault Condition by entering it in the Decision Expression of a Decision Action. Then you put your Throw Fault statement in the True branch of the Decision action. Here you can either specify additional conditions in the Throw Fault dialog's Fault Condition or leave it blank and simply specify the Fault document to which the fault information should be written. When the action is executed and all your conditions are true, the Throw Fault action is executed as described above. If the Fault Condition in the Decision action or Throw Fault action is false, the next action in the action model is executed.
- **Using a Throw Fault inside a Try / On Fault action.** By putting either of the above methods inside the Execute branch of a Try / On Fault action (which is described in "The Try/On Fault Action" below), you prevent the component from halting execution and have an opportunity to respond or recover from the fault. You create your fault condition using one of the previous two methods inside the Execute branch of a Try / On Fault action after other actions whose output you want to test worked correctly. You can specify any number of unique faults so that your component can branch into several different directions depending on which fault actually occurs. This works in a similar fashion to a Switch action. When the Throw Fault action for the given fault fires, instead of halting execution of the component, control passes into the appropriate On Fault branch of the Try / On Fault action. Here you can specify other actions to remedy or respond to the error.

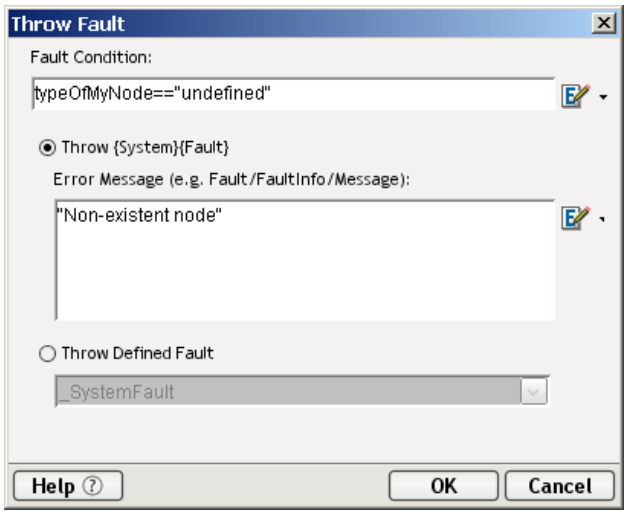
You can decide where it is appropriate to deal with error conditions and use Throw Fault accordingly. For instance, during runtime, if you want a component to stop running when an error condition is encountered, (and return control to the service in which it is running) use Throw Fault alone. The action throws an exception, which is displayed as a dialog box in Composer, and a stopped component on the application server.

On the other hand, suppose a service calls another component from within a Try/On Fault action (specifically under the **Try** branch). Inside the other component, a Decision action inspects some data in an XML document. If the data is valid, the component continues executing. If the data is not valid, the Throw Fault action executes,writing to the Fault document , and the component stops execution, returning control to the service. The Try/On Fault detects that a Throw Fault occurred and logic transfers to the appropriate On Fault branch of the Try/On Fault action. In the On Fault branch, you can process the Fault Message Part any way you like. You might, for example, write a message out to a Log file.

➢ **To add a Throw Fault action:**

1. Open a component.
2. Select a line in the Action Model where you want to place the Throw Fault action. The new action is inserted below the line you selected.

3 From the **Action** menu, select **New Action>Advanced,** then **Throw Fault**. The Throw Fault Action dialog box appears.



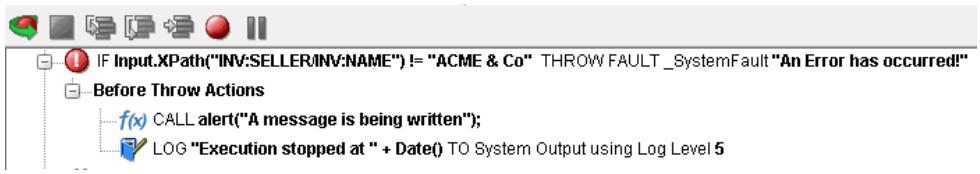
4 In the Fault Condition field, type a valid ECMAScript expression that, when true, causes the action to throw a fault. (You can also click the **Expression Builder** button and build an expression.)

5 Select **Throw {System}{Fault}** to write your error message to the _SystemFault document. By default, the message you type in the Error Message field will be placed in the Fault/FaultInfo/Message node of that document. Specify another node if desired. You also have access to the **ECMAScript Expression Builder** button so that you can build an expression.

6 Select **Throw Defined Fault** if you wish to select a Fault document that is one of the Message Parts you have associated with your component

7 Click **OK**.

The new Action is added to your model. Place any actions you wish to execute before the application halts in the Before Throw Actions area.

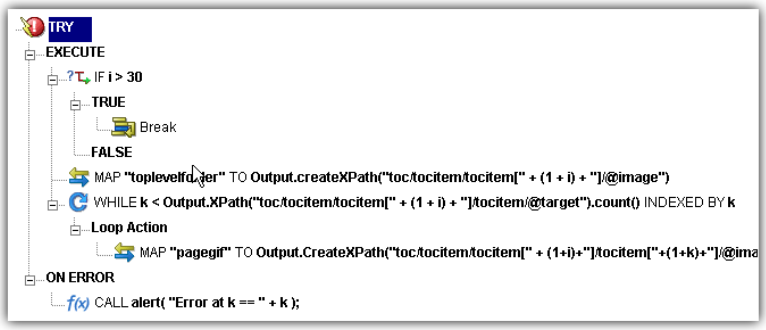


The Transaction Action

The Transaction action allows you to insert begin, commit, or rollback commands in your Action Model, thereby making it possible for you to exercise low-level control over transaction boundary demarcation within components that use transactions.

NOTE: This action is not available in Composer when installed as part of the Professional Edition suite.

Any Transaction actions that you place in your action list will result in the appropriate corresponding Java pass-throughs being generated in your service's application metadata. The details of how this occurs are beyond the scope of this discussion. See the Transaction Management chapter of the *Novell exteNd Composer User's Guide* for the Novell exteNd application server. (If you are using another application server, see the appropriate *exteNd User's Guide*.)



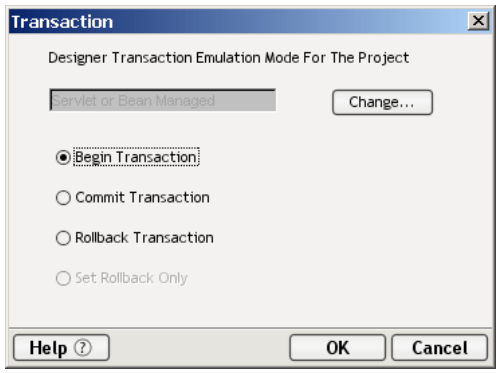
NOTE: Successful use of Transaction actions requires an in-depth understanding of Java transaction models. The services you create in exteNd Composer can be deployed using Servlet triggers or Enterprise Java Bean (EJB) triggers. The choice of deployment mode will have significant implications for transaction management.

➢ **To add a Transaction action:**

1 Open a component.

2 Select a line in the Action Model where you want to place a Transaction action. The new action will be inserted below the line you've selected.

3 From the **Action** menu, select **New Action>Advanced**, then **Transaction**. The Transaction dialog appears.

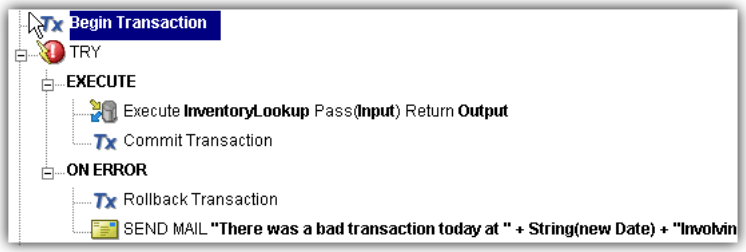


4 Select from one of the available transaction command types.

NOTE: Radio buttons are enabled or greyed out depending on which Transaction Mode you've selected in the **Designer** Tab of the **Preferences** dialog available under the **Tools** menu. For example, in the above illustration, the first three radio buttons are enabled while the **Set Rollback Only** button is greyed out. This is because the current transaction emulation mode is Servlet or Bean Managed. The **Set Rollback Only** button is available only in the context of a Container-managed EJB deployment; it is not applicable to Servlet or Bean-managed EJB deployments. To change emulation modes (and cause a corresponding change in which radio buttons are enabled in the Transaction dialog), click the **Change** button.

5 Click **OK**.

The following illustration shows a pair of Transaction actions as they appear in the Action Model pane.



Once you have generated Transaction actions in your Action Model, you can test them by executing the component in Composer (or by stepping through the action list as part of an animation/debug session). Appropriate error messages will appear based on any problems that might exist with your use of Transaction commands in your Action Model. For example, if you have used two begin commands in your action list with no intervening commit, you will see a warning dialog based on the fact that nested transactions are not supported.

The Try/On Fault Action

The Try/On Fault action executes a set of actions when a fault occurs within the Execute branch of the Try/On Fault action. Any number of defined faults can be specified within the Execute branch. You can use the Try/On Fault action to trap anticipated errors and run other actions to remedy or report on the fault. For instance, you can use Try/On Fault to respond to an XML Interchange action that fails to find a file.

When you add a Try/On Fault action, a dialog appears from which you select a number of pre-defined Fault Part Names. These are the Fault Messages you defined when you set up your component. Several lines are then added to the Action Model pane: the beginning of the Try action, the Execute branch, a branch for each Fault you specified and an “All other Faults” branch. When you are aware of potential faults an action can produce, you put those actions under the **Execute** branch. You then put error handling actions under each **On Fault** branch to handle unique situations. If a fault does occur, the actions under the **On Fault** branch execute.

Following the example given previously, if you anticipate a fault with the XML I/O action, you put the action under the Execute branch. Under one On Fault branch, you might add another XML I/O action that attempts to read the file from an alternate location. Under another On Fault branch, you might add another XML I/O action that looks for a file with a different extension.

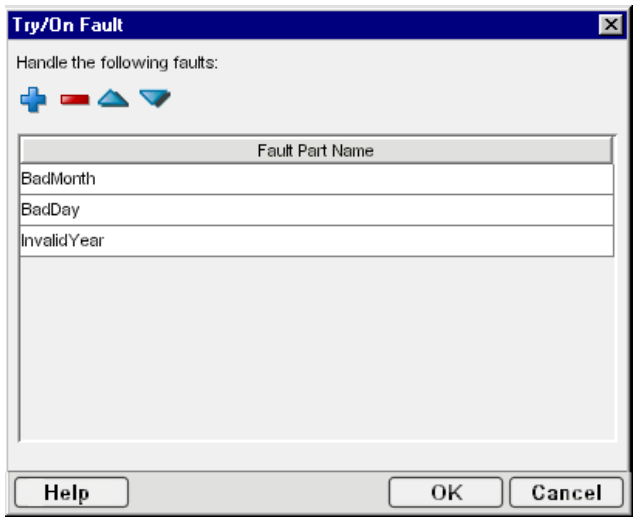
➢ **To add a Try/On Fault action:**

1 Open a component.

2 Select a line in the Action Model where you want to place the Try/On Fault action. The new action is inserted below the line you selected.

3 From the **Action** menu, select **New Action>Advanced,** then **Try/On Fault.**

4 The Try On Fault Dialog appears:



5 Use the blue + icon to add Fault Part Names you have previously associated with your component. Use the red - icon to remove them. Use the up and down arrows to change the order of the faults.

NOTE: If you do not define any custom Fault Parts, corrective actions can be placed in the default "All Other Faults" branch of the Try/On Fault action.

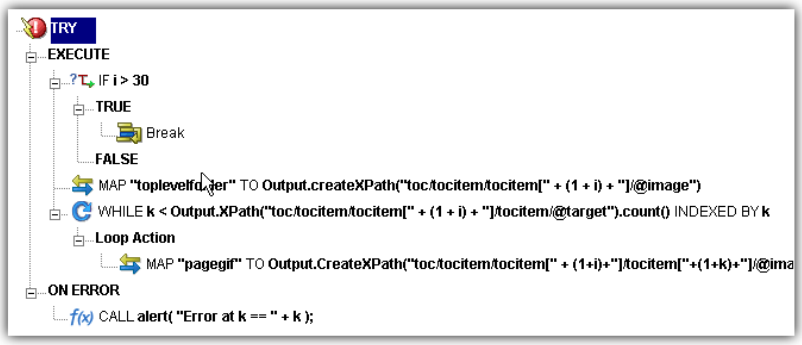
6 Click **OK** when you have finished defining your Fault Parts.

7 The Try On Fault action icon, with an **Execute**, one or more **On Fault** Branches, and an **All Other Faults** branch appears in the Action Model pane.

8 Add any actions that might cause potential errors under the **Execute** branch.

9 Add actions that resolve the error under the **On Fault** branch.

The following illustration shows a complete Try/On Fault action in the Action Model.



NOTE: It is good programming practice to use Try/On Fault actions liberally throughout your action model.

The XForm Process Action

NOTE: This action is not available in Composer when installed as part of the Professional Edition suite. For XForm-related functionality in Novell exteNd Professional Edition, you will use exteNd Director. See the Director documentation for details.

The XForm Process action allows you to specify an XForm document and subject it to various kinds of preprocessing before mapping it to output. Before using this action, you would typically already have created an Form Resource (see the chapter on Resources) in the current project, or you would (alternatively) be using an XForm as your component or service's Input message part. A typical scenario would be one in which a JSP, in response to a user request, kicks off a Composer service to handle a forms session. The key responsibility of the service would be to serve out the appropriate XForm.

In normal usage, an XForm is not transmitted to the user (the client) in its raw state, because an XForm is not renderable directly and embodies few assumptions as to what the final "rendered form" will look like. The same XForm may have an entirely different appearance on a desktop PC than it has on a palm device, for example. The decision of how to final-encode the form for presentation to the user is done at runtime, and the transformation from raw XForm to, say, XHTML must be handled at the server level since web browsers and client devices have no native support for XForm-rendering.

A typical roundtrip scenario might look like this:

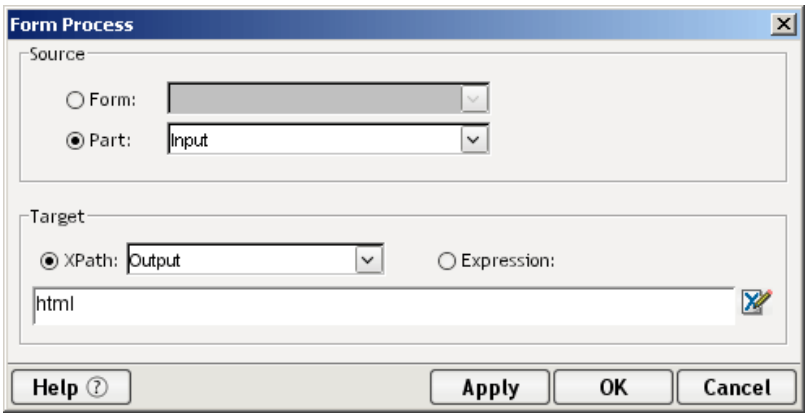
1. A customer goes to Company A's web site and decides to place an order. He or she clicks the "Order Now" button on the web page.
2. The button click results in a redirect to a URL that triggers a Composer service. Some user params (perhaps the user's first and last name, pulled from a cookie) are passed on the end of the URL.
3. The service calls a component that invokes an XForm Process action.
4. The XForm Process action:
   - Retrieves the proper Order Form from an Form Resource
   - Maps the user's name to the appropriate "instance data" locations in the form
   - Sends the form to the Novell exteNd XForm Processor to be converted to the appropriate output format (whether XHTML, SMIL, WML, or whatever)
   - And finally, appends, copies, or otherwise maps the transcoded document produced by the XForm Processor to a suitable Output message part
5. When the service has finished executing any additional business logic that might be dictated by the request, it serves the output document (containing the transcoded, prepopulated form) back to the user.
6. The user fills out the form and clicks the Submit button, triggering a redirect or another XForm session, or whatever action is necessary.

NOTE: This discussion is not meant to be a primer on XForms. For more information on XForm technology see **http://www.w3.org/MarkUp/Forms/**. For further discussion of the XForm capabilities provided by exteNd, see the documentation for exteNd Director.

➢ **To create an XForm Process action:**

1. Open a component (if necessary).
2. Select a line in the Action Model where you want to place the XForm Process action. The new action will be inserted below the line you selected.

3 From the **Action** menu, select **New Action > Advanced,** then **XForm Process**. A dialog appears.



4 In the upper portion of the dialog, choose one of the radio buttons:

- Choose the **Form** radio button if you want to specify an existing Form Resource as the source document. The pulldown menu will be prepopulated with the names of any Form Resources in your project. Select the Form Resource of interest.
- Choose the **Part** radio button if the XForm you want to use is already loaded into a message part (e.g., Input, Input1, Temp). In this case, choose from among the message parts shown in the pulldown menu, and in the text field just below, enter the XPath expression that points to the root of the XForm.

5 Under **Target**, specify (via either **XPath** or an **Expression**) the DOM node that will be the root of your XForm. (The example shown in the above illustration represents a typical use case where the target message part is Output, and the root node of the output document is **<html>**.)

6 Optionally click the **Apply** button to execute the action.

7 Click **OK** to dismiss the dialog. The new action is added to your action model.

The XSLT Transform Action

The XSLT Transform action takes a DOM and an XSL stylesheet you specify as input and sends the output to another DOM in the component. This process is also referred to as Server Side XSL Processing.

To create XSL output you need to specify three parameters of the action. The Source Document Expression is a valid ECMAScript expression that results in the name of a DOM or document handle (such as Input). The XSL URL expression is a valid ECMAScript expression that points to an XSL Stylesheet. This parameter is optional if the DOM already has an XSL Processing Instruction that specifies an XSL Stylesheet. If an XSL Stylesheet is not specified in the DOM, then you must specify this parameter. If you specify this parameter, and the DOM also has an XSL Stylesheet processing instruction, then your parameter will override it.

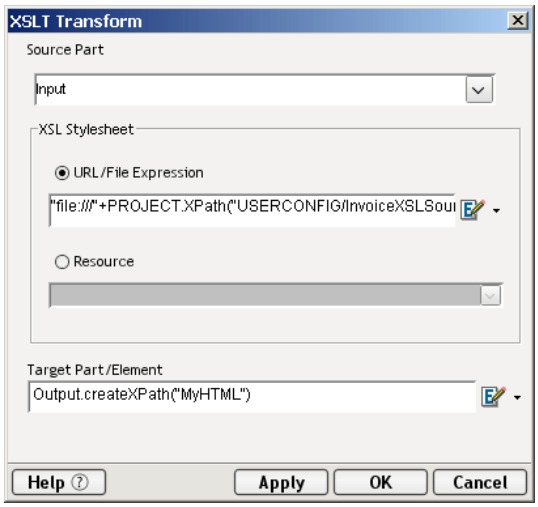
The Target Document/Element Expression specifies which DOM is to receive the results of the XSL processing.

➢ **To add a XSLT Transform action:**

1 Open a component.

2 Select a line in the Action Model where you want to place the XSLT Transform action. The new action is inserted below the line you selected.

3 From the **Action** menu, select **New Action>Advanced,** then **XSLT Transform**. The XSL Process dialog box appears.



4 Type the name of the **Source Document** you want rendered, or click the **Expression Builder** button and create an ECMAScript expression that resolves to a valid Part.

5 Type the name of the XSL stylesheet you want to use for transforming in the **XSL URL Expr** field, or click the **Expression Builder** button and create an ECMAScript expression that points to a valid stylesheet.

6 Type the name of the **Target Part/Element** you want to use, or click the **Expression Builder** button and create an ECMAScript script expression that specifies a Part.

7 Click **OK**.

The following illustration shows a complete XSLT Transform action in the Action Model.

Data Exchange Actions

This submenu contains actions concerned with the reading and writing of files and the interchange of data in web services and in XML.

Data Exchange Actions	Description
Composer Resource	Allows you to read in an XML or XSL resource
URL/File Read	Allows a file format that is not XML to be read into Composer.
URL/File Write	Allows a file to be written into a format other than XML.
WS Interchange	Executes a Web Service using messages and operations defined in a WSDL resource.

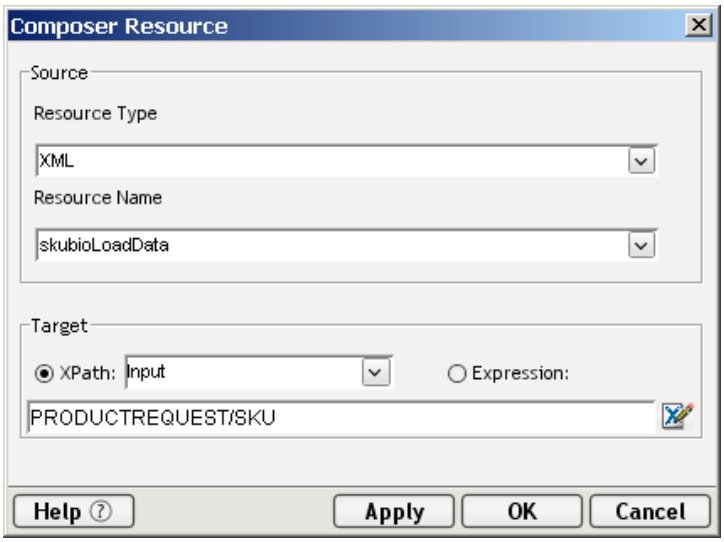
Data Exchange Actions	Description
XML Interchange	Reads external XML documents into the component's DOM or writes the component's DOM to an external XML document. Read/write methods include: Get, Put, Post, and Post with Response using the File, FTP, HTTP, and HTTPS protocols.

The Composer Resource Action

The Composer Resource Data Exchange Action allows you to load an XML or XSL resource into a Message Part.

➢ **To create a new Composer Resource action:**

1 From the Action menu, select **New Action>Data Exchange>Composer Resource**. The following dialog appears.



2 Under Source, select a **Resource Type**. The available choices are XML or XSL.

3 Select a **Resource Name**. You must already have added the XSL or XML file as a Resource in order for it to appear in this list. Refer to Chapter 9, “Resources” for instructions on how to accomplish this.

4 Under Target, use XPath to select a Part to contain the results of your XML or XSL, or click on Expression to enter the ECMAScript Expression Builder.

NOTE: Composer Resources will add the text of an XML or XSL document to a Part you specify. It can then be manipulated like any other Part. However, it will remain read-only.

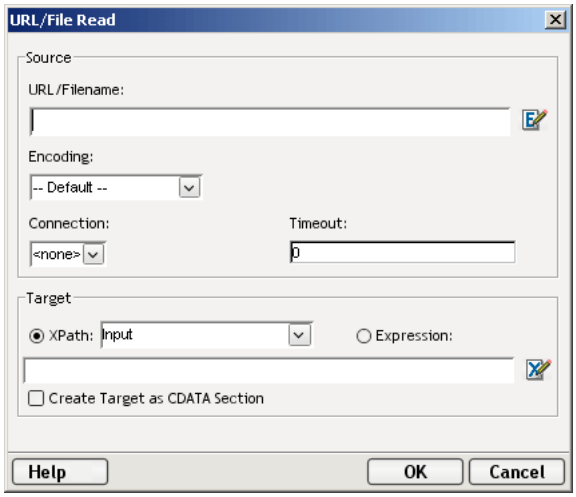
5 Click **OK** to add the Composer Resource Action to your Model.

URL/File Read

If a file is in a format other than XML, use this action to read the file into an XPath location.

➢ **To create a new URL/File Read action:**

1 From the Action menu, select **New Action>Data Exchange>URL/File Read**. The following dialog appears.



2 In the **Source File** portion of the screen, enter the file's URL. Since this is an ECMAScript expression, a URL string *must be enclosed in quotation marks*.

3 If applicable for the file format, select an **Encoding** algorithm from the dropdown menu.

NOTE: One common use case is shown above. The file in question might be binary, in which case it would be appropriate to select "Binary to Base64" from the dropdown. The appropriate *decoding* method can be specified in the URL/File Write action (below).

4 Select a **Connection Name**. Any HTTP, HTTPS and FTP connections resources you have created will appear in this list.

5 Specify a Connection **Timeout** value (in seconds), or leave as zero. Whatever value you place here will override any value specified in your connection resource.

6 In the Target File portion of the screen, select **XPath>Input** and enter the XPath destination of the file contents. You can also select Expression by clicking on the radio button. Doubleclick the Expression icon at right to bring up the Expression Builder, if desired.

7 Click the **Create Target as CDATA Section** checkbox if you want the contents of the file wrapped in a CDATA section. (This is not necessary for binary files that are to be encoded as Base64 per the above example.) This allows characters such as the angle brackets (< >) to be used inside an XML document without being interpreted as part of a start or end tag.

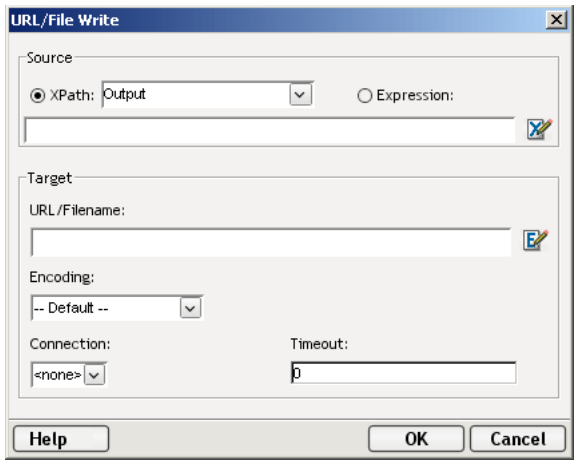
URL/File Write

If a file needs to be in a format other than XML, use this action to write the file from a DOM or message part.

NOTE: This action is, in every respect, the functional complement of the URL/File Read action described above.

➢ **To create a new URL/File Write action:**

1 From the Action menu, select **New Action>Data Exchange>URL/File Write**. The following dialog appears.



2 In the Source File portion of the screen, Select Source **XPath>Output**.

3 Enter the XPath containing the file content. (Alternatively, select the Expression radio button and enter an ECMAScript expression that specifies the location of the file contents.)

4 In the Target portion of the screen, enter the URL where the file is to be stored.

5 If applicable for the file format, select from the **Encoding** list box to specify a decoding before the file is written.

6 Select a **Connection Name**. Any HTTP, HTTPS and FTP connections resources you have created will appear in this list.

7 Specify a Connection **Timeout** value (in seconds), or leave as zero. Whatever value you place here will override any value specified in your connection resource.

The Web Service (WS) Interchange Action

In most cases, you will use Composer to build *consumable* services, but in some situations, you may have a need for your service to act as a *consumer* of other services.

The Web Service Interchange action allows your component to invoke a Web Service according to calling conventions specified in a WSDL Resource. (See “About WSDL Resources” on page 240 for more information about WSDL Resources.) You will use this action in scenarios that might require your component or service to act as a *client* in a web-service interaction involving a remote service.

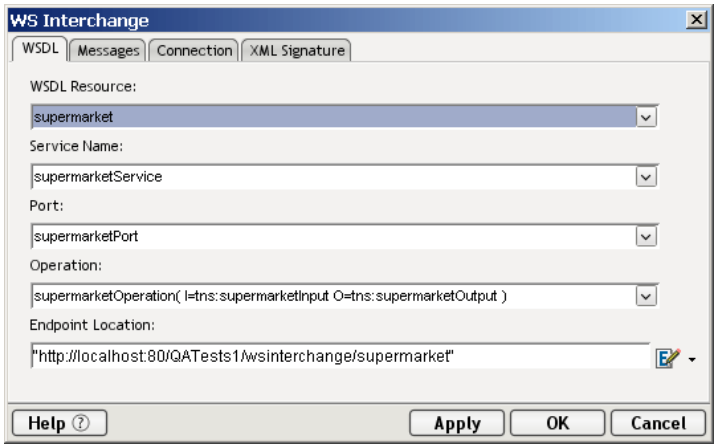
Note that before you can create a Web Service Interchange action, you *must* have a WSDL Resource that describes the service.

➢ **To create a Web Service (WS) Interchange action:**

1 Open a component.

2 Select a line in the Action Model where you want to place the Web Service Interchange action. The new action is inserted below the line you selected.

3 From the **Action** menu, select **New Action/Data Exchange,** then **WS Interchange**. The Web Service Interchange dialog appears.

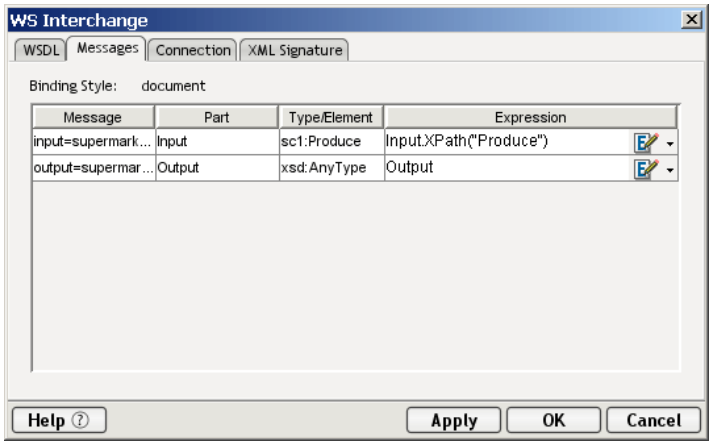


4 Choose the desired **WSDL Resource**, **Service Name** (if applicable), **Port**, and **Operation** from the dropdown menus provided. (These menus will be prepopulated with choices taken from the information in your existing WSDL Resources. For information, refer to the section on WSDL Resources in Chapter 9, “Resources”)

5 Enter the **Endpoint Location** (usually a URL pointing at a servlet) for the Web Service you wish to use, wrapped in quotation marks. (Alternatively, enter an ECMAScript expression that will evaluate to an Endpoint Location at runtime.)

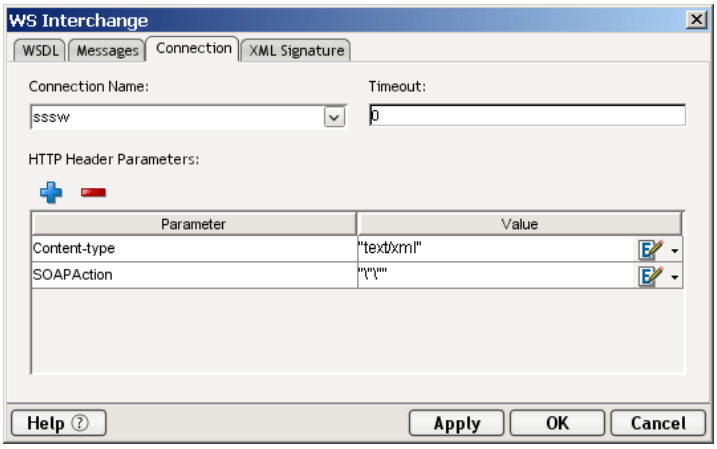
NOTE: This is the only field on the WSDL tab of the dialog that you should have to fill out by hand.

6 Click the **Messages** tab to bring up the following panel:



7 Specify the input and output messages for the particular service you are going to invoke. The **Message**, **Part**, and **Type/Element** fields will be prepopulated. Under **Expression**, enter the ECMAScript expression that describes the source and target for each message. Usually, this will be an expression that specifies an XPath location in an Input Part or Output Part. Click the Expression Builder icon at the far right to go to the Expression Builder dialog, where you can easily build the appropriate expression(s) via point-and-click.

8 Click the **Connection** tab to bring up the next panel:



9 Choose an HTTP Connection Resource (as needed) from the **Connection Name** dropdown menu.

NOTE: For ordinary HTTP connections, you can specify <none> here. The intent of this field is to let you connect via HTTPS to a secure site using the user ID and password information stored in an HTTP Connection Resource.

10 Specify a Connection **Timeout** value (in seconds), or leave as zero. Whatever value you place here will override any value specified in your connection resource.

11 The **Parameter** and **Value** fields in this dialog should already be populated, based on the Operation and Message information given in other tabs of the dialog. If the Value fields are empty, enter appropriate strings or expressions for the type of SOAP action and/or the content type (MIME type) of the exchange.

12 If you wish to specify additional HTTP header information, click the **Plus sign** above the combo box to add new HTTP parameter fields.

13 Click the **XML Signature** tab to bring up the next panel: (Optional)

14 Use the prepopulated pulldown menu to select an existing Certificate Resource in your project. (See “About Certificate Resources” for details on how to create this kind of resource.)

15 Check the box if you would like to **Validate the XML Signature** on the way out.

16 Click **Apply** to test the Web Service action in real time, or click **OK** to dismiss the dialog.

The XML Interchange Action

The XML Interchange action reads external XML documents into a component's DOM and writes data from a component's DOM out as XML files. There are four types of XML Interchange actions:

- GET
- PUT
- POST
- POST with Response

When using the **Get** interchange, fill in the "Interchange URL Expression" field with a URL that points to the XML document you want to bring into the component. If you have created an HTTP or FTP Authentication connection resource, you can specify it under "Connection Name." Otherwise, the connection information would need to be embedded in the URL. In the "Response Part" field, you will specify a DOM which is to receive the XML. If the DOM name you specify does not exist, it will be created.

When using the **Put** interchange, fill in the "Interchange URL Expression" with a URL that points to the location to which you want to write the XML document. Select a "Connection Name" from the list if you have already created an HTTP or FTP Authentication connection resource. Otherwise, the connection information will need to be embedded in the URL. For "Request Part", you will specify the name of a DOM in your component to send its data as XML.

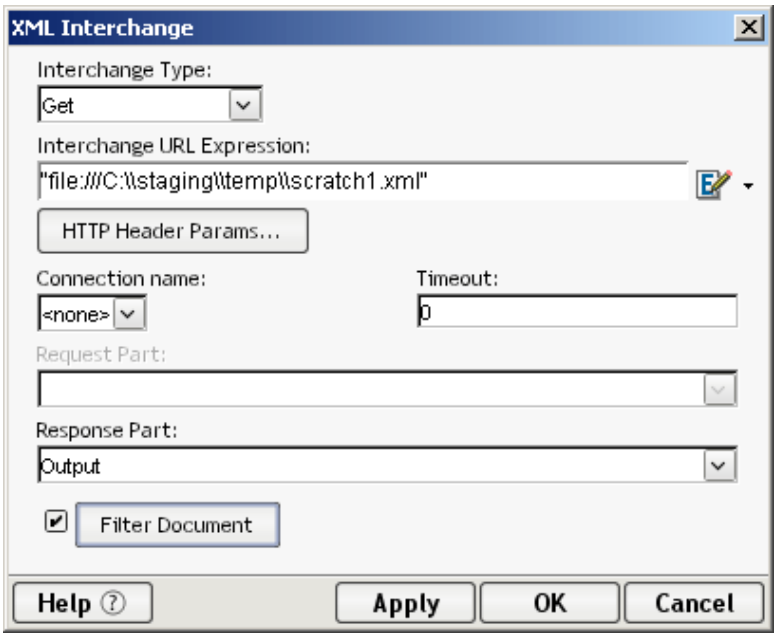
When using the **Post** interchange, fill in the "Interchange URL Expression" with a URL that points to the location to which you want to write the XML document. Select a "Connection Name" from the list if you have already created an HTTP or FTP Authentication connection resource. Otherwise, the connection information will need to be embedded in the URL. For "Request Part", you will specify the name of a DOM in your component to send its data as XML.

When using the **Post with Response** interchange, you supply the same parameters as for Post, with one additional parameter. You must also specify a "Response Part" DOM to receive the Response XML document from the Post with Response action. The difference between the two interchanges is that Post with Response expects a response XML object back from the origin server.

➢ **To add an XML Interchange action:**

1. Open a Component.
2. Select a line in the Action Model where you want to place the XML Interchange action. The new action is inserted below the line you selected.

3 From the **Action** menu, select **New Action/Data Exchange** then **XML Interchange**. The XML Interchange Action dialog box appears.



4 Select an **Interchange Type**.

5 In the **Interchange URL Expression** field, type an expression that defines a fully qualified URL for an XML document using any of the following supported protocols:

- file
- ftp
- http
- https

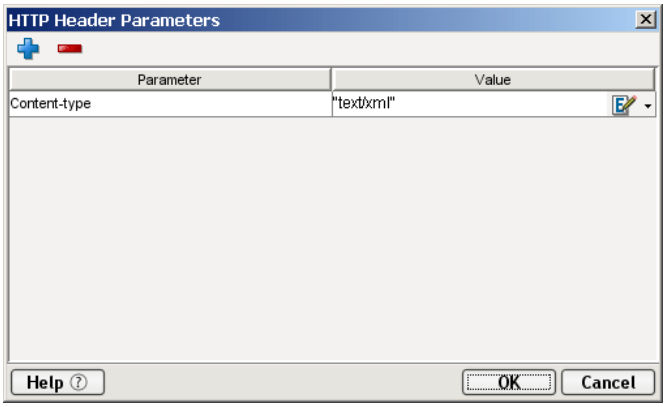
Depending on the Interchange Type selected, this URL is the source or the destination of the XML file for the XML Interchange action. For example:

file:///g:/xmldata/invoicebatch1.xml

ftp://accounting:password@123.456.789.987:21/invoices/inv1.xml

Since this is an ECMAScript expression, a URL string *must be enclosed in quotation marks*.

6 Optionally click the **HTTP Header Parameters** button. The HTTP Header Parameters dialog appears.



7 Click the plus (+) icon to add new header parameters. Enter a **Parameter** name and the desired corresponding **Value**. Common HTTP header parameters include “Content-Type,” “Content-Length,” and “Keep-Alive.” You can add any number of Parameter-Value pairs in this dialog.

8 Click **OK** to close the HTTP Header Parameters dialog. The XML Interchange dialog reappears.

9 Select a **Connection Name**. Any HTTP and FTP connections resources you have created will appear in this list.

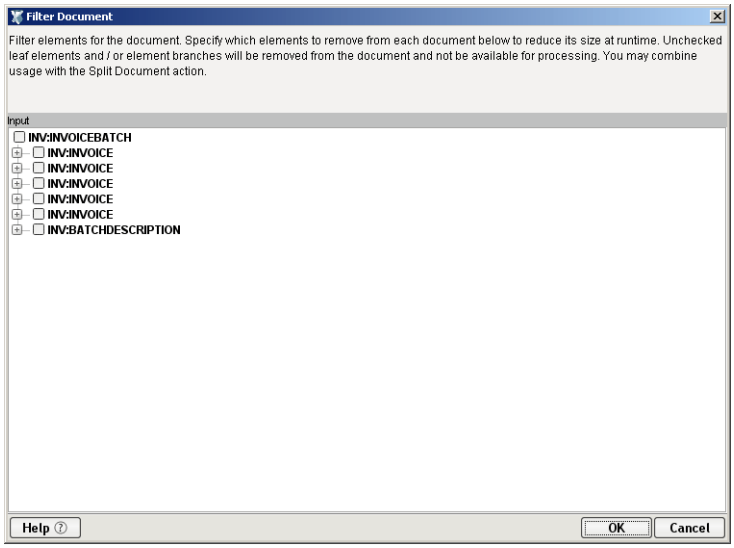
10 Specify a Connection **Timeout** value (in seconds), or leave as zero. Whatever value you place here will override any value specified in your connection resource.

NOTE: A value of zero means that no time limit is placed on the connection, unless you are using an HTTP Connection Resource (which is optional for non-authenticated connections). If a timeout value is specified in that connection resource, it will be used.

11 In the **Request Part** field, specify the name of a DOM in your component to send its data as XML. Request Part is used for **Put**, **Post** and **Post with Response** Interchange types.

12 In the **Response Part** Field, specify the name of the DOM tree that will receive XML. **Response Part** is used for Get and Post with Response.

13 Optionally check the checkbox next to the **Filter Document** pushbutton (thereby enabling it). If document filtering (see discussion below) is desired, click the pushbutton. A dialog will appear:



NOTE: The document shown in the dialog will be the one selected in Response Part in the XML Interchange dialog.

The purpose of this dialog is to allow you to specify individual nodes that are to be retained (rather than stripped off) the incoming XML document in real time for purposes of improving performance and reducing RAM overhead.

Check the checkbox next to the nodes you want to *keep* in the document. Unchecked nodes will be stripped off (discarded) prior to parsing the DOM. (See additional discussion in the section following this one.)

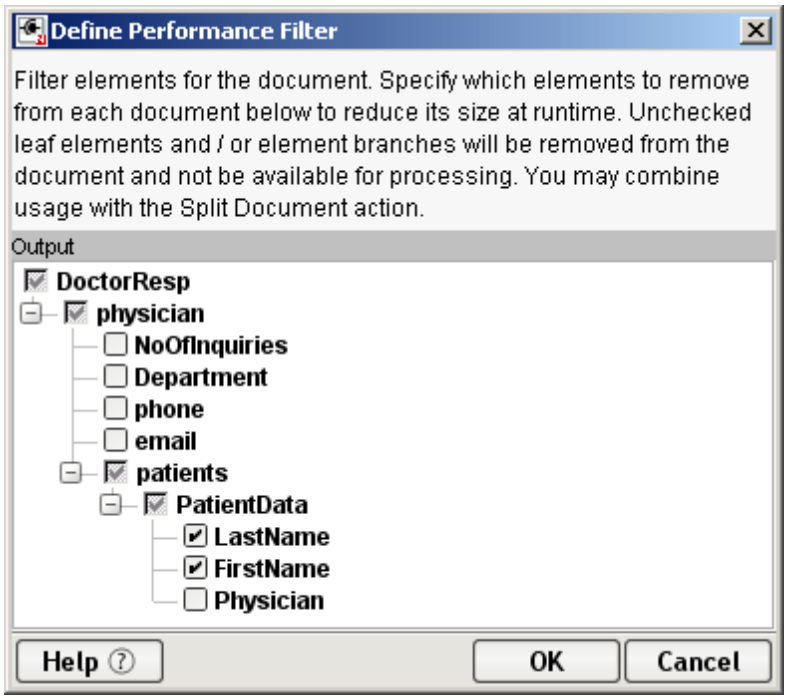
When you have selected nodes that you wish to be kept, click **OK** to dismiss the dialog.

14 Click **OK**. Alternatively, you can press the Apply button to see the affect of the XML Interchange action without closing the dialog. This allows you to make repetitive edits to a XML Interchange action and quickly see the results.

Performance Enhancement Using “Filter Document”

The **Filter Document** button in the XML Interchange dialog (further above) offers the potential for greatly improved performance when processing large incoming documents. It also offers potential benefits in terms of memory conservation, since a filtered document will require less memory.

The **Filter Document** button brings up a resizable dialog containing a tree view of the document in question.



For XML Interchange actions, the document shown in this dialog will depend on the interchange mode (GET versus POST with Response) as well as the target message part you've selected in the combo boxes provided. (Note that you cannot get to this dialog if PUT or POST have been selected, since in those cases there will be no incoming document; only an outgoing one.) In the tree view display, every element of the document will have a checkbox next to it. Any elements that you check will be kept when the document is DOM-parsed for use in your component. Any boxes that are unchecked will result in the associated elements (and their attributes) being discarded, so that the parsed DOM is smaller than it would otherwise be.

In the above illustration, the incoming document, with root node **DoctorResp**, will have a **/physician** node with a **/patients** node under it, and the **/patients** element, in turn, will have a **PatientData** element under it. Likewise, the latter will have child nodes **LastName** and **FirstName**. But since **Physician** is *not* checked, the incoming document will not have anything under the XPath:

```
DoctorResp/physician/patients/PatientData/Physician
```

Similarly, there will not be anything under **/physician/NoOfInquiries**, **/Department**, etc., because those nodes were not checked.

It's quite common to encounter scenarios in which only a few nodes or XPath locations in a given input document are of interest to a particular component or service. When this is the case, it makes sense to use the Filter Document dialog to strip away unneeded portions of the input document. Careful use of document filtering will allow you to create services that process documents efficiently and quickly, with minimal RAM impact.

NOTE: You can apply document filtering (using the above dialog) to *any* input document for any kind of service (not just documents arriving via the XML Interchange action). See the discussion in Chapter 6, "Creating an XML Map Component", for further information on how to filter Input documents.

Repeat Actions

This submenu contains actions that implement looping and loop-control constructs.

Repeat Actions	Description
Break	Stops execution of a Repeat for Element, Repeat for Group, or Repeat While loop and continues execution with the next action outside the loop.
Continue	Stops execution of the current Loop iteration in a Repeat for Element, Repeat for Group, or Repeat While loop, and continues at the top of the same loop with the next iteration.
Declare Group	Allows you to create and name a group based on an element that occurs multiple times. Groups are used in the Repeat for Group action.
Repeat for Element	Repeats one or more actions for each occurrence of a specified element in your DOM tree. The Repeat For Element action allows you to create a loop based on an element that occurs multiple times.
Repeat for Group	Repeats one or more actions for each member of a group. A Repeat For Group action allows you to re-structure your data and calculate aggregates on your data.
Split Document	Allows a service or component to read (and process) a large input document in sections, rather than all at once. This can be an important strategy for reducing machine-resource requirements at runtime. It can also result in faster throughput.
Repeat While	Repeats one or more actions by creating a loop. A While Repeat action allows you to base a processing loop on any valid ECMAScript expression.

The Break Action

The Break Action stops the execution of a Repeat for Element, Repeat for Group, or Repeat While loop. The Action Model continues execution with the next action *outside* the loop.

The use of a Break is appropriate when, for example, you are using a loop to search a node list for one particular item. When the target item is found, there is no need to continue iterating; hence a Break can be used to terminate the loop immediately.

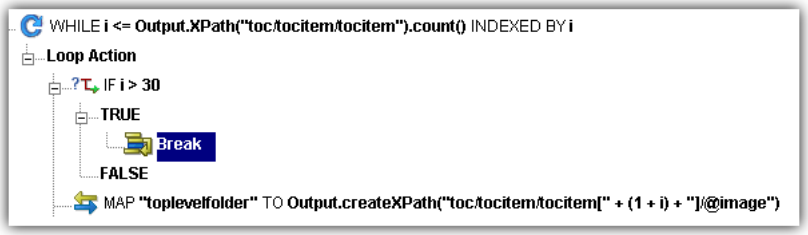
NOTE: A Break action will typically occur in one branch of a Decision action (within a loop). You'll place the Break action in either the True or False branch of the Decision action, as appropriate.

➢ **To add a Break action:**

1 Open a component that contains a Repeat action you wish to modify to include a Break action.

2 Select a position inside the loop where you wish to place the Break action. Generally, this will be in one leg or the other of a Decision action (as shown below).

3 From the Action menu, select **New Action>Repeat** then **Break**. The Break action appears immediately in the action model. (There is no setup dialog.) See below.



The Continue Action

The Continue action causes execution of the current iteration of a Repeat for Element, Repeat for Group, or Repeat While loop to stop and execution to begin at the top of the loop, with the next iteration. The Continue action provides a way to short-circuit downstream actions inside the loop while allowing the loop to continue on to the next iteration.

A Continue action is appropriate in a situation where, for example, one item in a list should be skipped over some reason, yet execution of the loop must continue.

NOTE: A Continue action will typically occur in one branch of a Decision action (within a loop). You'll place the Continue action in either the True or False branch of the Decision action, as appropriate.

➢ **To add a Continue action:**

1 Open a component that contains a Continue action you wish to modify to include a Continue action.

2 Select a position inside the Loop actions where you wish to place the Continue action. This will generally be inside one fork or the other of a Decision action; see illustration below.

3 From the Action menu, select **New Action>Repeat>Continue**. A Continue action appears in the action model.

The Declare Group Action

The Declare Group action allows you to create two special lists, each in reference to a DOM. These group lists can then be used as the basis for a loop in the Repeat for Group action. To create the lists, you supply a Group Name and specify an XPath. Composer then creates the lists as follows: a Group list is created that contains one entry for each unique value found among all the elements that match the XPath. The Group list is referred to by the Group Name you supply. Then a Detail list is created for each unique entry in the Group list that contains as many entries as there are members in the Group (i.e., a non-unique list). The Detail list is referred to by the Group Name you supply post-fixed with the label "(Detail)."

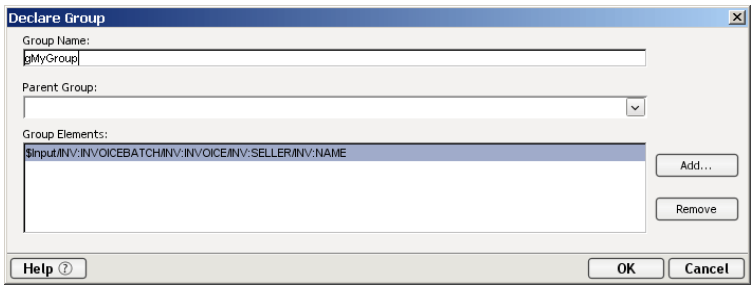
Grouping allows you to select a repeating element in your Input DOM and create fewer elements based on the unique values across all instances (siblings) of that repeating element. So instead of having multiple elements, you end up with one element for each unique element value in your Output DOM.

➢ **To add a Declare Group action:**

1 Open a component.

2 Select a line in the Action Model where you want to place the Declare Group action. The new action is inserted below the line you selected.

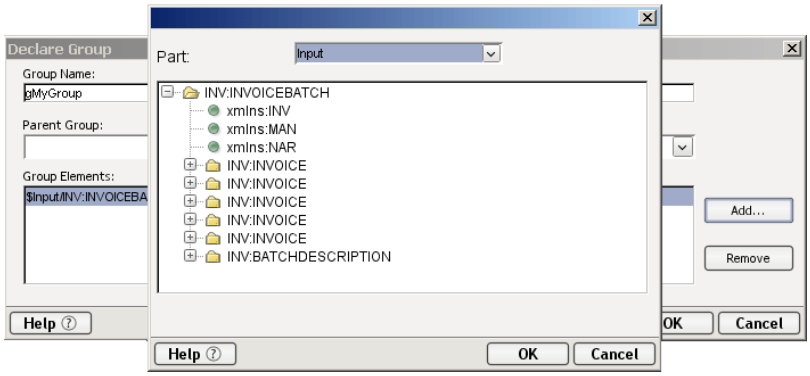
3 From the **Action** menu, select **New Action>Repeat** then **Declare Group**. The Declare Group dialog box appears.



4 Type a name for the group.

5 Optionally, select a parent group. This is used if you want to create multiple group levels.

6 Click **Add**. The Add Element dialog box appears.



7 Select a Part name and an element.

8 Click **OK**.

9 Repeat steps 6 through 8 to add more elements to the group.

10 Click **Remove** to delete elements from the group.

11 When you have all the elements you want in the group, click **OK**.

NOTE: An example of this can be found in the Action Examples sample project installed on your computer.

The Repeat For Element Action

The Repeat action creates looping structures within an Action Model. Loops give you the ability to repeat a set of one or more actions. There are three types of loops: Repeat For Element, Repeat For Group, and Repeat While.

XML allows multiple instances of an element in a document (analogous to multiple records in a database table). The number of instances can vary from document to document and is defined in the Document Schema (DTD or XML Schema). For instance, you might receive an XML document containing lineitems for an invoice on a daily basis. Each day the XML document has a different number of lineitems. Not knowing how many instances of “lineitem” are in the XML document poses a problem if you want to transfer these item numbers from the input XML document to an output XML document programatically. The Repeat For Element action solves this problem.

The Repeat For Element action allows you to mark an element that occurs multiple times. The action then sets up a processing loop that executes one or more actions for each instance of the marked element until no more instances exist. In the example above, the processing loop would contain a single Map action to transfer the lineitem number and this action would be repeated until all lineitems had been mapped.

The Repeat for Element action also uses the concept of an alias. An alias performs two functions. It is an alternate name or shorthand for the marked repeating element, which saves you the work of re-specifying long XPath expressions. In some cases, the repeating element may be several levels down in the document hierarchy. When you create Map actions in the Repeat loop that transfer child elements of the marked element, using the alias is quicker than re-typing a long XPath expression. An alias is also an indicator to Map actions within the Repeat loop to use the next instance of the repeating element each time the loop processes. A Map action within a Repeat for Element loop that does not use the alias always refers to the first instance of the element in the source Part.

NOTE: Hovering the mouse over a Repeat alias in the Map dialog will display a tool tip showing the XPath represented by the alias.

The Repeat For Element action allows you to process more than one action within the loop. In the simplest case, the repeat loop might only contain one Map action that transfers the value of the current element instance from the input Part to the output Part. You can also define multiple actions in the processing loop, for example: a Map action to transfer the current value and a Log action that writes to a file, creating an audit of each transfer.

➢ **To use a Repeat For Element processing loop:**

1 Create a Repeat For Element action.

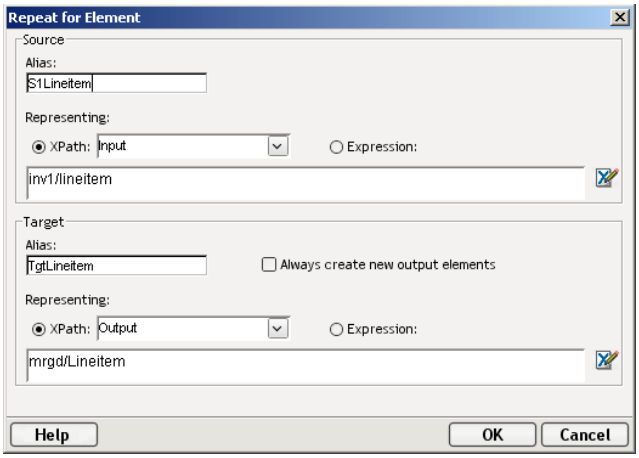
2 Create actions (Map, Log, Decision, etc) within the Repeat For Element processing loop.

➢ **To create a Repeat For Element action:**

1 Select the first instance of a repeating element in an XML Document tree.

2 Right-click on the repeating element in the Part or, from the **Action** menu, select **New Action>Repeat**, then **Repeat for Element**.

The Repeat for Element dialog appears.



3 Begin by identifying your **Source**.

- Type in the **Alias** field. A good naming convention for an alias is to use the element name with a prefix indicating sources or target and the type of repeat action such as “S1Lineitem.”
- Enter an XPath expression, or click the **Expression Builder** button and build an XPath expression for the repeating element. The example above show an XPath which points to a root node of “inv1” and it’s child node “lineitem.”

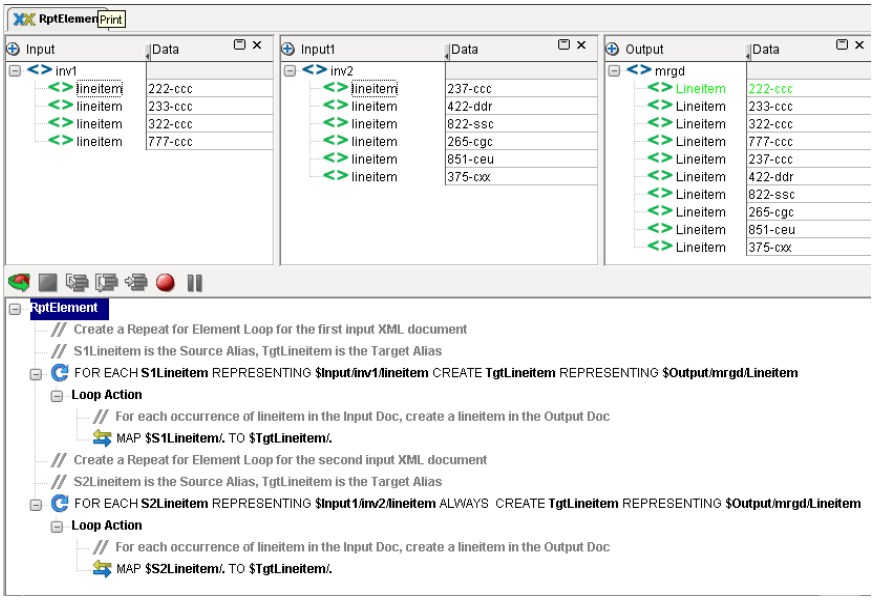
4 The next step is to identify your **Target**.

- Create another **Alias**, such as “TgtLineitem.”
- Use the checkbox to indicate if you want to **Always create new output elements**. This box would be used in situations where you had multiple input documents containing similar node structures which you wanted to merge into a single Output DOM with common node names. Refer to the Action Model following the final step of this procedure for an example.
- Enter an XPath expression, or click the **Expression Builder** button and build an XPath expression for the repeating element. In this case, we are using “mrgd/Lineitem.”

5 Click **OK**. Your Repeat for Element loop is added to the Action Model.

6 Highlight **Loop Actions** to begin adding Map actions or whatever other actions are necessary for your component.

The following illustration shows an Action Model for a component containing two Repeat For Element actions and the input and output XML documents that are used by the component. This model contains two Repeat For Element groups because the user has two very similar input DOMs containing an unspecified number of lineitems. Map actions are used within the processing loop to transfer the lineitems from the two input DOMs to the single output DOM.



The Repeat for Group Action

The format of an XML document that you receive is not always the format that meets the requirements of your business process. For instance, an XML document might contain invoices from different sellers. The data is received as individual invoices, but in the context of a business-to-business transaction, you might need to summarize the data and send the summary data to a manager, and at the same time, send the invoice data to the Accounts Payable department.

A Repeat for Group action allows you to re-structure your data and/or establish a framework to calculate aggregates on your data. Grouping allows you to select a repeating element in your input Part and create fewer elements based on the unique values across all instances (siblings) of that repeating element. Instead of multiple seller elements across the invoices (some with the same seller value), you end up with one element for each unique seller value in the output Part.

The Repeat For Group action sets up a processing loop based on one of two lists created by the Declare Group action. The loop executes as many times as there are entries in the list you use (either the Group list or Group (Detail) list). In the above example, if you use the Group list, once you have one element per seller, you can add Map actions to the processing loop to calculate how many invoices each seller had. You can also list the individual invoice numbers beneath each seller. By combining a Repeat for Group with Map commands, you can create a new XML document whose structure and data are different from the original.

In a way similar to the Repeat for Element action, a Repeat for Group action also uses the concept of an alias. The values for Source Group used in the Repeat for Group dialog are the list names created by the Declare Group action. The list names perform two functions. They are an alternate name or short-hand for the XPath source of any Map actions within the loop. This saves you the work of re-specifying long XPath expressions. The group list name when used in place of a DOM name in a Map action source, is also an indicator to the Map action within the Repeat loop to use the next instance in the group list each time the loop processes. A Map action within a Repeat for Group loop that does not use the group name always refers to the first instance of the element in the source Part.

The target aliases created in the Repeat for Group action also serve two functions. They are an alternate name or short-hand for the XPath target of any Map actions within the loop. This saves you the work of re-specifying long XPath expressions. The target alias when used in place of a Part name is also an indicator to Map actions within the Repeat loop to create a new instance of the Source in the target Message Part. A Map action within a Repeat for Group loop that does not use a target alias always overwrites the first instance created in the target Message Part with subsequent instances from the Source group list.

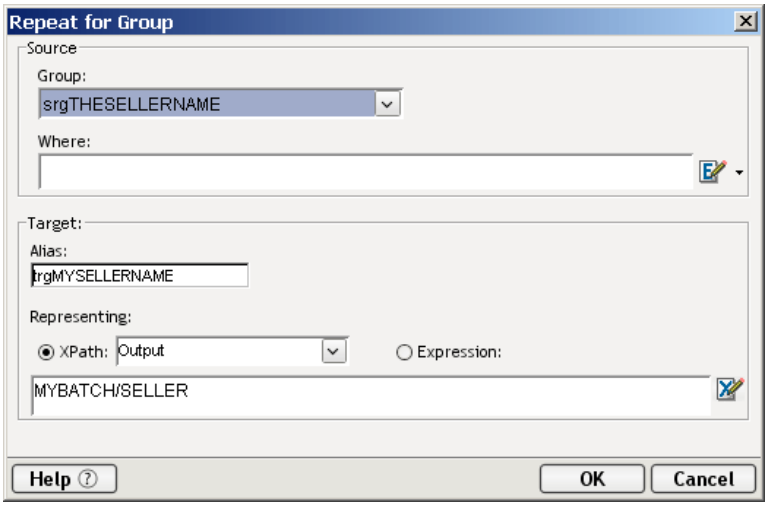
To create a Repeat for Group action, you need to complete these three tasks:

- Create a Declare Group action to create the group lists.
- Create a Repeat for Group action specifying which group list to use.
- Create Map actions inside the loop.

➢ **To add a Repeat for Group action:**

1 Open a component.

2 From the **Action** menu, select **New Action>Repeat,** then **Repeat for Group**. The Repeat for Group dialog box displays.



3 Under Source, select a **Group** name on which to base the Repeat for Group action loop.

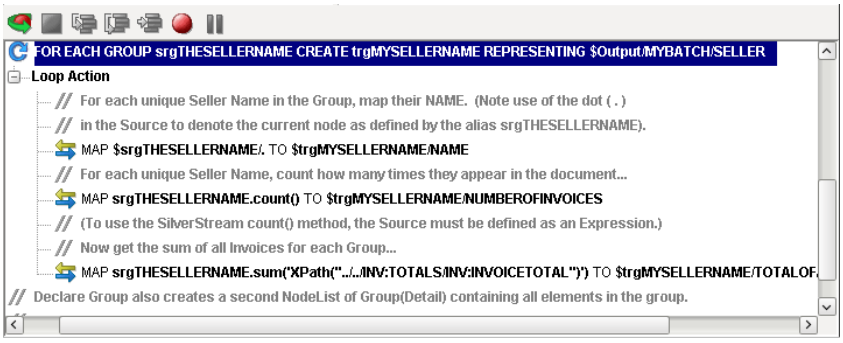
4 Optionally, type in a **Where** clause to filter the group list, or click the **Expression Builder** button and create a Where expression.

5 Under Target, you can optionally create an **Alias** name to be used by Map actions in their target expressions.

6 Create an XPath or Expression to be represented by the alias.

7 Click **OK**.

The following illustration shows a complete Repeat For Group action in the Action Model pane.



The Repeat While Action

The Repeat While action repeats one or more actions as long as a condition that you specify remains true. For instance, you can create a variable that contains the total sales from line items within invoices. You can then create a Repeat While action that reads invoices, totals the line items, and stops when the line item total reaches a certain amount.

The target alias created in the Repeat While action serves two functions. It is an alternate name or short-hand for the XPath target of any Map actions within the loop. This saves you the work of re-specifying long XPath expressions. The target alias when used in place of a DOM name in a Map action is also an indicator to Map actions within the Repeat loop to create a new instance of the Source in the target DOM. A Map action within a Repeat for Group loop that does not use a target alias always overwrites the first instance created in the target DOM with subsequent instances from the Source.

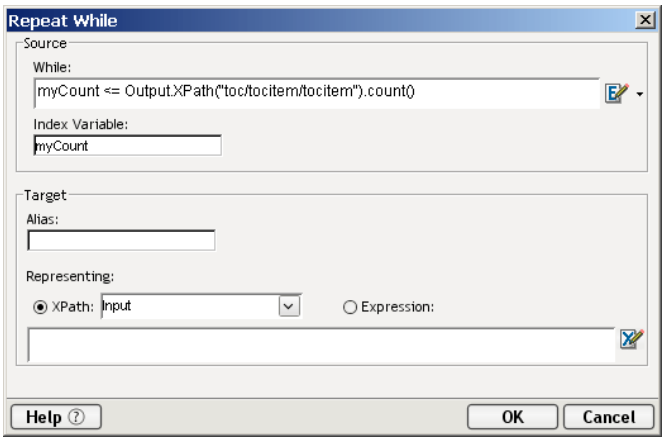
NOTE: Unlike the Repeat for Element and Repeat for Group, the Repeat While does not have to be based on data in a DOM tree. The loop can operate independently of data in the DOM tree.

➢ **To add a Repeat While action:**

1 Open a component.

2 Select a line in the Action Model where you want to place the **Repeat While** action. The new action is inserted below the line you select.

3 From the **Action** menu, select **New Action>Repeat,** then **Repeat While**. The Repeat While dialog box appears.



4 Under Source, type an expression to test the While loop, or click the **Expression Builder** button and build an expression.

5 Type a name for a variable that keeps track of the condition of the loop.

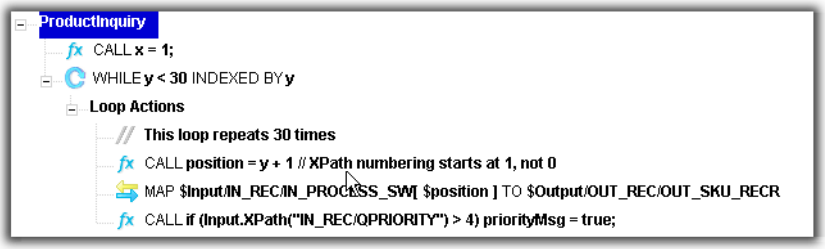
6 If you know the alias for the **Target** element, type in the **Alias** field.

7 If you do not know the alias, select either XPath and a Part element, or Expression and type in a valid expression.

8 Enter a criteria statement, or click the **Expression Builder** button and build an expression.

9 Click **OK**.

The following illustration shows a complete Repeat While action in the Action Model pane.



The Split Document Action

When a service receives an input document, Composer's default behavior is to read the entire document into memory at once, then parse it into a DOM. Message Parts (Input, Input1, Temp, etc.) are then passed between components—or from service xObjects to components—as self-contained DOMs. This approach is appropriate for most services. But in some circumstances, such as when a service routinely encounters large documents, machine memory and parsing overhead become significant issues. In such situations, it can make more sense to process large documents in *pieces*.

The Split Document action is a special-purpose action designed to enable piecewise processing of large XML documents. With the Split Document action, input documents are treated as *streams*. A stream can be consumed in arbitrarily defined chunks; the chunks, in turn, can be processed serially. The net result is a much reduced demand on system RAM, and (potentially) higher throughput from reduced DOM-handling overhead.

You should consider using the Split Document action when:

- Your service can be expected to encounter large input documents (200+ Kbytes) containing repeating elements, *or*
- Your service runs out of memory on the server due to large DOM sizes, *or*
- Your deployed service has a performance bottleneck that you think may be related to DOM parsing, *or*
- The nature of your business logic (and/or your input documents) is such that it would be more natural to process the data streamwise, in *chunks*, than to create and debug a single large Repeat For Element loop.

Limitations of Stream-Based Document Processing

The Split Document action is subject to some important caveats. The most obvious limitation is that the document in question should be piecewise-processable; which is to say, it should contain *repeating elements* (identifiable split points where the document can be separated into chunks). The split points are defined in terms of an XPath expression representing the type of node on which to break. (While it is technically possible to split a non-repeating XML Document into two parts using the Split Document action, this would be an abnormal use case and is not recommended.)

It's important to understand that because the document is encountered in chunks, and because each chunk is released from memory (goes out of scope) after it is processed, any business logic that has to "know about" data in downstream parts of the document (such as a footer section) while processing upstream parts can't be expected to work. In general, any dependencies that span "document chunks" will, at the very least, require custom workarounds involving operations that "keep track of" document characteristics as processing occurs.

NOTE: If global knowledge of document statistics is required—or if it is necessary to use aggregate-oriented XPath methods like count(), last(), etc.—then stream-based processing using the Split Document action is not appropriate, because the entire document needs to be read into memory.

How the Split Document Action Works

The Split Document action should be used in the top-level Service xObject that wrappers all of your service's components. It also should be the first "DOM-processing" action in that service's action model. That is to say, no other action preceding the Split Document action should reference the Message Part (typically Input) that will be split.

The first action in the action model that references a document determines how that document will be handled. If the first action to reference Input is a Map action (or other non-Split-Document action), then Input will be treated as a single, monolithic DOM. If, on the other hand, the first action to reference Input is a Split Document action, the source document for Input will be treated as a *stream*. At that point, no self-contained "DOM version" of the streamed document will ever be available, for the life of that service instance.

NOTE: Within a given service, a particular document can be processed either as a DOM *or* a stream, but not both. However, if an Input document is processed in stream fashion, only that document is handled that way. Other documents (Temp, Output, etc.) will be subject to the normal DOM parsing.

The Split Document action requires you to specify an XPath expression representing the type of document element on which to split. Consider the following hypothetical XML document, representing a batch of invoices.

```
<DATA>
<PrologInfo/>
        <BatchDate/>
<InvoiceBatch>
    <Invoice/>
        <Line Item/>
        <Line Item/>
```

```
    <Invoice/>
        <Line Item/>
        <Line Item/>
    <Invoice/>
        <Line Item/>
        <Line Item/>
    <Invoice/>
        <Line Item/>
        <Line Item/>
    <Invoice/>
        <Line Item/>
        <Line Item/>
</InvoiceBatch>
<SummaryLog/>
     <NumberOf<Invoices/>
</DATA>
```

The natural “split point” for this document might be an XPath of

```
DATA/InvoiceBatch/Invoice
```

Using this XPath with a Split Document action, the above document would be read in the following chunks:

```
<PrologInfo/>
        <BatchDate/>
<InvoiceBatch/>
     <Invoice/>
        <Line Item/>
        <Line Item/>
```

followed by:

```
<InvoiceBatch/>
     <Invoice/>
        <Line Item/>
        <Line Item/>


<InvoiceBatch/>
     <Invoice/>
        <Line Item/>
        <Line Item/>


<InvoiceBatch/>
     <Invoice/>
        <Line Item/>
        <Line Item/>


<InvoiceBatch/>
     <Invoice/>
        <Line Item/>
        <Line Item/>
<SummaryLog/>
     <NumberOf<Invoices/>
```

There would be five chunks, total. The first and last chunks would be “special” in the sense that they contain header and trailer (or prolog/epilog) data in addition to the Invoice data. There is nothing special about how they were created, however. The document was simply split wherever `DATA/InvoiceBatch/Invoice` occurred.

NOTE: Each time a split occurs, the chunk that gets created contains *the entire subtree under the parsing node*. If the chunk is the first chunk in a document that contains prolog information *before* the first parsable node, then the first chunk will contain all of the document (including prolog) up to and including the first parsable node and its children. Similarly, if the document has information following the last parse-tree, anything trailer-nodes will travel with the chunk.

Controlling the Size of Chunks

If the document in the foregoing example had contained *thousands* of invoices, splitting it into one-invoice chunks probably would not be wise. (Component-calling overhead could be expected to result in a performance hit.) For efficiency, it would be better to break the document into larger-sized pieces. The Split Document action allows you to do exactly that. You can override the default "strict parsing" behavior shown above by specifying a value *greater than one* in the "Occurrences per split" portion of the Split Document dialog. (See further below.) This way, in a document containing a thousand invoices, one could split on every ten or every hundred invoices. It would be up to the invoice-handling component (the component to which "chunks" are passed) to loop through individual invoices at the action-model level.

Suppose the document in the previous example were processed with a Split Component action in which the "Occurrences per split" parameter is set to 2. The resulting chunks would look like:

```
<PrologInfo/>
        <BatchDate/>
<InvoiceBatch>
    <Invoice/>
        <Line Item/>
        <Line Item/>
    <Invoice/>
        <Line Item/>
        <Line Item/>
```

followed by

```
<InvoiceBatch/>
     <Invoice/>
        <Line Item/>
        <Line Item/>
<InvoiceBatch/>
     <Invoice/>
        <Line Item/>
        <Line Item/>
```

followed by

```
<InvoiceBatch/>
     <Invoice/>
        <Line Item/>
        <Line Item/>
<SummaryLog/>
     <NumberOf<Invoices/>
```

Notice that once again, header elements come as part of the intial chunk, while footer elements are contained in the final chunk. The first two chunks contain two invoices each. The final chunk contains just one, since 5-modulo-2 is one. The final chunk, in other words, contains the "remainder" (or leftover pieces) from the splitting operation. This means that the Repeat loop in your chunk handler will need to be able to deal gracefully with situations where a chunk contains less than the expected number of pieces. One way to do this is to base the loop's termination condition not on a fixed number, like 2 or 10 or 100 (representing the "Occurrences per split" value), but on an actual count of the number of target nodes contained in the incoming chunk.

For example, the following ECMAScript expression would tell you how many <Invoice> elements are in a given chunk, in the previous example:

```
Input.XPath('InvoiceBatch/Invoice').length
```

You can safely continue iterating until the loop counter variable reaches the amount returned by this expression.

Loop Control and the Split Document Action

The Split Document action is itself a looping action: Composer places a “Loop Action” block under the Split Document line in the action model automatically. You will probably put a Component action within the Loop Action block, along with pre- and post-processing logic for chunks, exception-handling code, etc.

Loop termination is handled automatically, in the sense that you do not have to declare a counter variable (nor specify a termination condition). Composer simply performs the appropriate number of stream-reads and splits, and stops when there are no more “chunks” in the stream.

You can terminate the loop prematurely (or continue on to the next iteration at any point in the loop) by, for example, placing a Break (or Continue) action in the True branch of a Decision action. More sophisticated loop control can be achieved using Try/On Fault (see “The Try/On Fault Action” above) in the service and Throw Fault in the chunk-handler component, or by analyzing a custom Output doc returned by the chunk handler, etc.

Chunks as Documents

Typically, you will place a Split Document action in a service that calls a chunk-handling component (which might be an XML Map component, a JDBC component, or any other component type). The service will call the component via a Component action. The component will operate on the chunk’s contents, using whatever business logic is necessary. The component may or may not hand an Output document back to the original service; and the service itself may or may not construct an Output document for the benefit of the invoker.

The service containing the Split Document action will typically be splitting the `Input` message part. The Component action (calling the chunk handler) will in turn specify “Input” as the input to the handler component. In effect, a chunk becomes a DOM (a first-class document) in its own right. Any of the normal DOM operations can be performed against it. It can be passed component-to-component, written to disk, appended to other DOMs, mapped into or out of, etc.

Special Considerations for Animation and Debugging

When you open a service containing a Split Document action, the Input document window will initially be empty. (Ordinarily, you would expect to see your Input template document in tree view.) As you step through the action model in animation mode, the document window will populate as soon as you execute the Split Document action. The window will show the first “chunk” of the input stream, based on a parsing of the Input template. If you continue to use Step Into, each trip through the Loop Action block will re-load the Input window with the appropriate chunk from the input stream. At any time, during any of these iterations, you can Stop the animation and then perform drag-and-drop mapping of data from Input to other message parts (such as Temp or Output) as needed.

After the Split Document action is complete, the last chunk of the document will remain in the Input pane. Footer data can be mapped at that point (using drag-and-drag or an ordinary Map action) to Output, or otherwise processed, as desired.

NOTE: At no time will the entire Input template document be visible in the document window. Only pieces will be visible. If you need to see the entire document, open the appropriate Template itself, outside the component.

An important behavior to be aware of at design time is that the document-handling mode (stream versus DOM) is not set until you Save the component or service you're editing. In other words, if you add a Split Document action to an action model and immediate animate it (without Saving), the stream processing behavior will not be evident. You should Save the service or component after making any change to the action model that would change the document-handling behavior (stream vs. single DOM) of the service/component.

Another important principle to be aware of at design time is that if you happen to place an action that references Input *anywhere upstream of* the Split Document action in your action model (even if it's merely a Function or Log action used for debugging purposes), Input will be treated as a single large DOM at animation time. When you then Step Into the Split Document action, an exception occurs, because there is no stream. (The stream has already been fully consumed in order to create the DOM.) As mentioned earlier, the Split Document action *must* be the first action that references the document in question. Any other actions that reference that document must occur downstream of Split Document in the action model.

A final consideration to bear in mind is that although the Split Document action is designed to facilitate working with large documents, you should not actually use a large document as a sample at design time. Composer needs a large amount of memory at design time when large sample documents are loaded. This is true even though a particular service might use stream processing (via Split Document) to process the document. For design time, you should use a relatively small sample document—a reduced-size version of the "real thing," just large enough to prove out the action model. Use fullsize documents after deploying to the app server.

Creating the Split Document Action

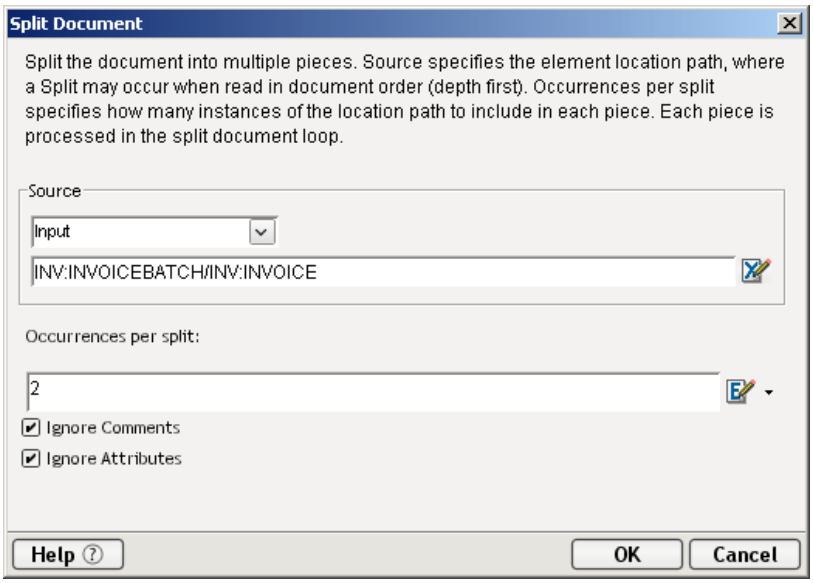
The following procedure steps you through the process of creating and using a Split Document action. It assumes that you have an Input sample document to work from, representing (in structure, if not in actuality) a large, splittable XML document. It also assumes that you have created a component to handle the processing of individual document chunks (a "chunk handler"), and a Web Service that calls that component.

➢ **To create a Split Document action:**

1. **Open** the service in which you plan to use a Split Document action, if it is not already open.
2. Place the cursor at the point in the action model where you intend to add the new action. (Highlight or select the line preceding the intended location.)

   NOTE: Be sure to heed the earlier warning about not placing the Split Document action after (downstream of) any existing action that references the document to be split (typically Input). If Input will be split, no action in the action model should reference Input unless the action in question comes *after* (downstream of) the Split Document action.

3 Either use the Action menu to create the new action, or right-mouse-click and choose **New Action > Repeat > Split Document...** from the context menu. A dialog appears.



4 Under **Source**, use the dropdown menu to select the message part (e.g., Input) representing the document to be split.

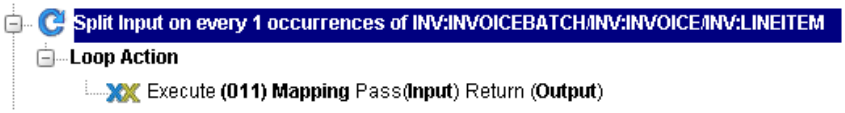
5 Also under **Source**, enter (in the text field provided) an XPath expression representing the node axis on which to split the document. (Click the small X-icon at the far right to bring up the XPath Expression Builder, if you'd like to have Composer help you build the expression in point-and-click fashion.)

6 Under **Occurrences per Split**, enter a positive integer representing the number of repeating pieces to include in a chunk. The default is one, meaning that Composer will split the document on *every* occurrence of the specified parsing node. To split on every third occurrence, enter 3. For every fourth occurrence, enter 4. And so on.

7 (Optional) Check the **Ignore Comments** checkbox if you would like XML comments to be automatically stripped from the input stream as the document is processed. This is a performance-enhancing option designed to speed the processing of (and reduce memory usage related to) documents that might contain large quantities of comments (possibly machine-generated).

8 (Optional) Check the **Ignore Attributes** checkbox if you would like Composer to discard attribute data while reading the input stream. Again, this is a potential performance-enhancer, meant to conserve memory and reduce processing overhead when dealing with large documents.

9 Click **OK**. A new action is added to the action model of the service.



Note that a "Loop Action" block appears automatically under the Split Document action.

10 Add a **Component Action** to the Loop Action block, so as to call the chunk handling component that will process individual pieces of the input doc. (See "The Component Action" for information on how to create and use this action.) In the above example, an XML Map Component called "Mapping" is called, with the service's Input passed as input to the component.

NOTE: Remember that at execution time, Input (in this case) actually represents a *piece* of the service's input doc.

11 Optionally add any other pre- or post-processing actions your service might need, in the Loop Action block.

12 **Save** your service.

IMPORTANT: Your Split Document action will not work (in animation mode) unless you have first Saved your service.

9 Resources

A resource is a reusable xObject that a component may need in order to carry out a task. For example, most XML integration applications communicate with a “back end” system of some sort; and to do this usually requires establishing a *connection* of some kind involving the specification of IP or JNDI addresses, ports, driver location, user ID and password, etc. This type of info can be stored in a reusable object and then accessed by a component at runtime. Resource xObjects accomplish this.

A Resource consists of the resource itself (whether a JPEG image, a JSP, an XSL stylesheet, or what have you) plus XML metadata about the resource, so that the characteristics of the resource are known to Composer Enterprise Server and to other runtime processes. At deployment time, all of your project’s resource are packaged into the deployment archive (usually a JAR inside an EAR), and they become available on the application’s classpath.

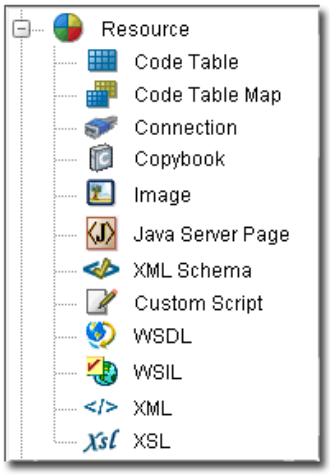
The core resource types available in Composer include those listed below. (Asterisks are shown next to resource types that are not available in the Professional Edition version of Composer.)

- Certificate
- Code Table
- Code Table Map
- Connection
- Copybook
- Custom Script
- DTD
- Form (XForm)*
- Image*
- JAR (Java archive)*
- JSP (Java Server Page)*
- WSDL
- WSIL
- XML*
- XSD
- XSL*

In addition to these core types, various Composer Connect products use additional resource types specific to the connector. For example, the EDI Connect allows you to specify EDI Document Metadata and EDI Interchange Metadata resources.

NOTE: The creation of Connect-specific resource types is explained in the documentation for the connector. Only core Composer resource types are discussed in the sections to follow.

Resource types have distinguishing icons (which are displayed in the Category Pane of Composer’s main navigation frame). The resource categories and their associated icons look like this:



Working with Resources

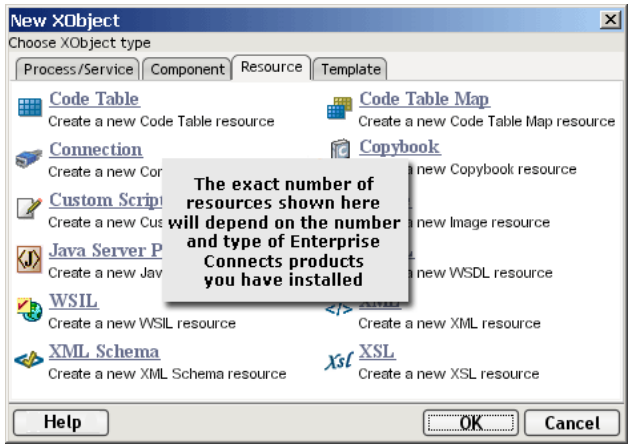
Custom-created resources of a given type appear in the Instance Pane when you select (highlight) a particular resource category (Code Table, Connection, etc.) in the Category Pane. Each resource instance is reusable by the various components and/or services in your current project (and can be imported into other projects as well).

At component creation time, the wizard for the component will prompt you for the name(s) of the resource(s) you would like to be used by the component. This means that resources need to be created *first*, so that components can use them.

All resources are created using the same basic procedure, indicated below.

➢ **To add a new Resource to a Project:**

1 From Composer's **File** menu, select **New,** then **xObject**. From the **Resource tab**, select the desired category of Resource. See below.



2 Once you select your resource type, a wizard will appear, prompting you for the name of your custom resource and other information pertinent to the type of resource being created. Fill in the information requested by the wizard.

3 On the final screen of the wizard, click **OK**. The new resource is added to the Resource Category in question, and its name is displayed in the Instance Pane.

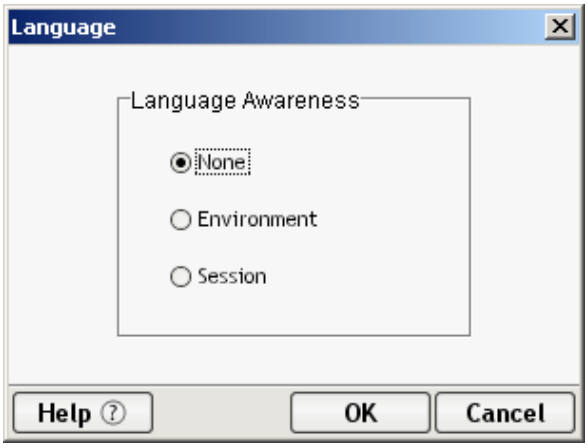
Support for Language Versioning of Resources

Composer supports a mechanism for dynamically selecting a language-appropriate version of an XML resource at runtime based on filename hints. The way it works is as follows. Suppose you have have two versions of an XML Resource file named **MyInvoice_en.xml** and **MyInvoice_jp.xml.** The "_en" file would contain Latin characters while the "_jp" file would have Japanese characters. (The language specifier is the two-character ISO-639 code.) At runtime, a Load Resource action can choose the appropriate language version of the XML resource in question, based on the language settings specified in design time.

To take advantage of this scheme, you must adhere to the following rules:

- You must use file-naming protocol mentioned earlier. (Namely: Every file name must end with an underscore followed by the two-character ISO-639 code for the language.)
- You must create one resource for each individual file. The applicable resource types are XSL, XML, and Form (i.e., XForm).
- You will specify a Language option when creating a Composer Resource action, or when specifying a stylesheet resource in a deployment object.

In Composer dialogs that offer a resource-picker dropdown list, such as the Composer Resource action, you will see a Language button. If you press that button, you will bring up the following dialog.



Choose one of the radio buttons:

- **None**: Applies no preference.
- **Environment:** Choose the language of the host machine.
- **Session:** Chooses the language specified in the servlet request.

Depending on which button you choose, the resource picker will update dynamically to show the list of available resource choices. Choosing **None** above means that every available resource (of all languages) will be displayed in the picklist. Choosing **Environment** or **Session** means the Resource Name list box will be populated with only the unique names of resources for the selected Resource Type *after stripping the language and locale suffixes from the file names*. In other words, the list is filtered according to the language-awareness preference chosen in the above dialog.

About Certificate Resources

Certificate Resources are used to hold Digital Signature information. Data that can be stored in this kind of resource include a public key x509 certificate, a corresponding private key, and a private key password.

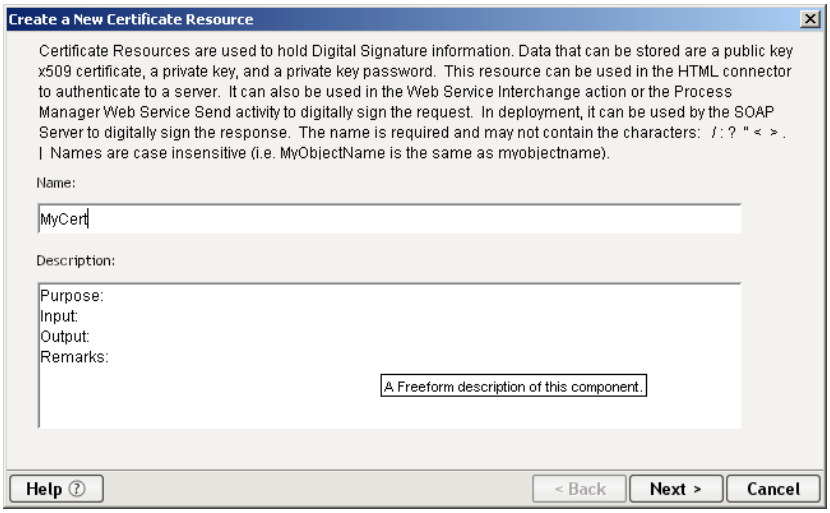
The Certificate Resource can be used in several ways:

- In the Web Service Interchange action, it can be used to digitally sign a request.
- In SOAP deployment, it can be used to signal the SOAP Server to digitally sign the outgoing response. (This option can be set in the options-panel UI for the SOAP HTTP trigger.)
- In the HTML Connect, it can be used to authenticate to a server.
- In the Process Manager, you can use a Certificate Resource to sign the outbound request of a Web Service Send activity.

NOTE: Since the Certificate Resource is a first-class Composer resource, it can be shared among components and services within a project.

➢ **To create a Certificate Resource:**

1 In Composer's navigator pane, right-click on **Certificate** (under Resource) and choose **New** from the context menu. A dialog will appear.



2 Enter a Name for the resource.

NOTE: The name is *required* and may not contain the characters: / : ? " < > . | Names are case-insensitive (i.e. MyObjectName is the same as myobjectname).

3 Click **Next**. A new dialog appears.

4 Use the **Browse** button to navigate your local drive or network to locate a suitable **Client Certificate** (x509).

5 Use the **Browse** button to navigate your local drive or network to locate a corresponding **Private Key** file. This key will be used for encryption of outbound digests and payloads. It will not be transmitted nor exposed to processes other than those residing in your Composer project.

6 Enter a **Private Key Password** (as applicable) so that your private key can be retrieved from the local keystore.

7 Click **Finish**. The resource is added to the navigator detail pane.

About Code Tables

In building your Composer applications you are often faced with the requirement to repetitively transform data you receive. Typical examples for this type of conversion include changing state codes (e.g., Alabama, Illinois) to regions for classification or accounting codes as they are moved between systems. Composer provides the capabilities to assist you with this type of conversion. For example, a "Code 1" for a bookstore may represent the fiction category, or a department store may use "Code M" to represent men's clothing.

If you were to design your application so that the output XML only included "Code 1" or "Code M," with no other description, the result could be cryptic and confusing. This is where a code table comes into play. A code table stores commonly used business code tables and works in conjunction with a code table Map to produce an output XML document that is more meaningful to the person or business process receiving the output. In the case of the bookstore, the input XML that included "Code 1," might be mapped using a code table to produce an output XML with a category "fiction."

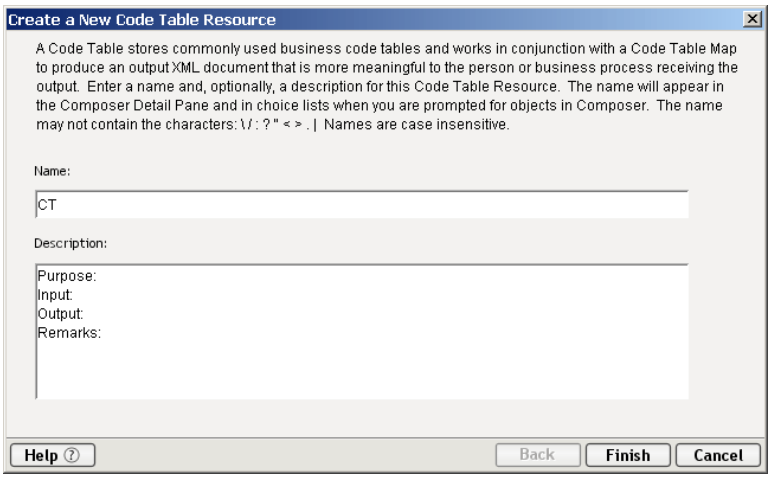
About the Code Table Editor

The Code Table Editor includes both menu options and a tool bar. In addition to the menu options, the Code Table Editor includes a tool bar with the following buttons:

Button	Description
	Save. Clicking this button saves changes to the open code table.
	Cut. Clicking this button removes the highlighted data from the Code Table Editor and puts in onto the Windows Clipboard.
	Copy. Clicking this button puts a copy of the highlighted data onto the Windows Clipboard.
	Paste. Clicking this button puts the contents of the Windows Clipboard at the position of the cursor, or replaces highlighted text.
	Delete. Clicking this button removes data from the currently active (or selected) cell of the Code Table Editor.
	Add Row. Clicking this button adds a new, blank row into the Code Table Editor.
	Delete Row. Clicking this button deletes the currently active (or selected) row from the Code Table Editor.

➢ **To create a code table:**

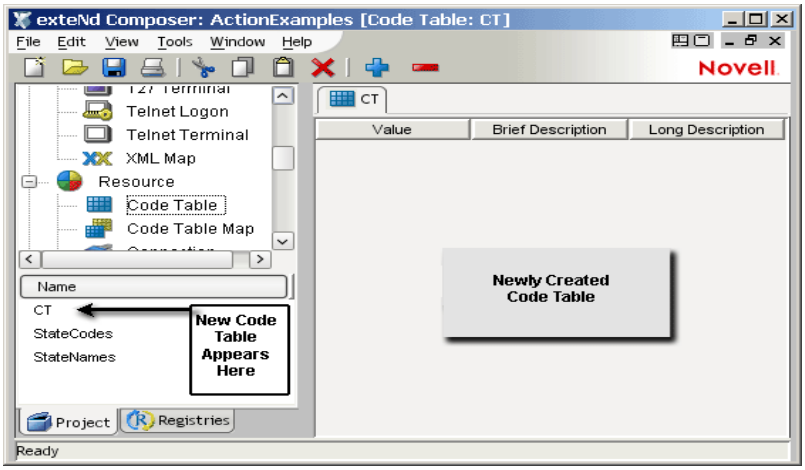
1 Select **File>New>xObject**. From the **Resource** tab, select **Code Table**. The Create a New Code Table xObject wizard appears.



2 Type in a **Name**.

3 Optionally, you may type in **Description** information.

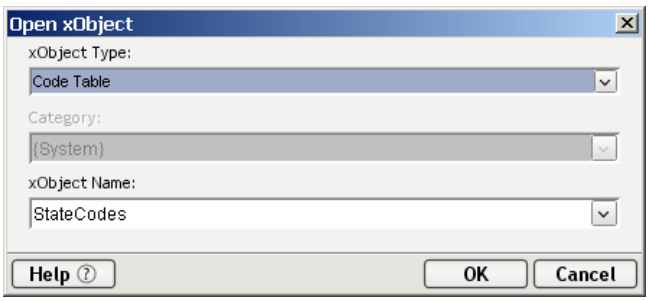
4 Click **Next**. The Code Table Editor appears with the name of your empty code table in the title bar.



When you close the Code Table Editor, the name of your new code table appears in the Resource category of the Composer window, under Code Table, as shown.

➢ **To open a code table:**

1 Select **File>Open**. The Open xObject dialog appears.



2 Select **Code Table** from the **xObject Type** dropdown list.

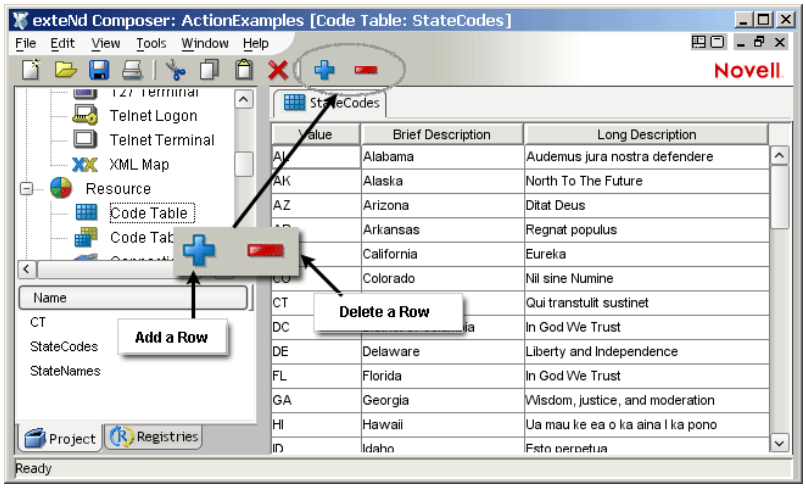
3 Select the code table you wish to open from the **xObject** dropdown list.

4 Click **OK**. The code table you selected opens in the Code Table Editor.

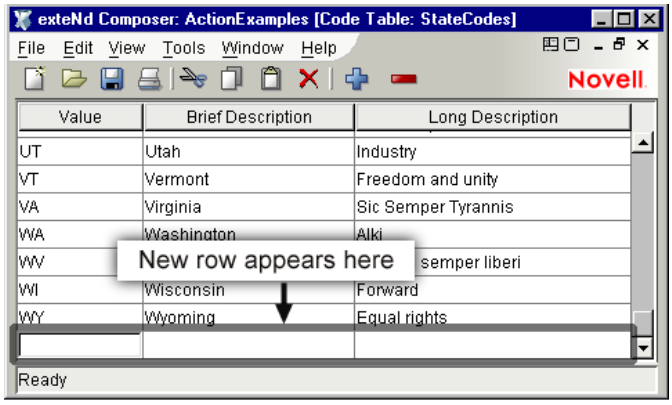
NOTE: Optionally, you may select Code Table in the category pane of the Composer window and doubleclick a code table from the detail pane.

➢ **To add data to a code table:**

1 Open the code table to which you'd like to add data. The code table you open appears in the Code Table Editor.



2 Click on the **Add Row** button. A blank row appears in the Code Table Editor window.



3 Click in the cell where you want to add data.

4 Type in the new data:

- In the **Value** field, type in the element data from the XML sample you are using
- In the **Brief Description** field, type a short description.
- In the **Long Description** field, type the full description.

5 Repeat steps 3 and 4 until you've added all your data.

6 Select **File>Save**, or click the **Save** button.

NOTE: Alternate and faster ways to enter data are to copy data from a spread sheet and paste it into the code table. Make sure your selection contains three columns. The first column must contain data; the second and third columns are optional. Open the spreadsheet, copy the three columns and as many rows as needed. Open the code table and immediately press the Paste button. You can also copy data from tables in a Microsoft Word® document using the same technique.

➢ **To edit a code table:**

1 Open the code table you'd like to edit.

2 Highlight the data you'd like to edit.

3 Use the **Edit** menu or the Code Table Editor tool bar buttons to cut, paste, or copy your selected data.

4 Click the **Save** button when you are done editing.

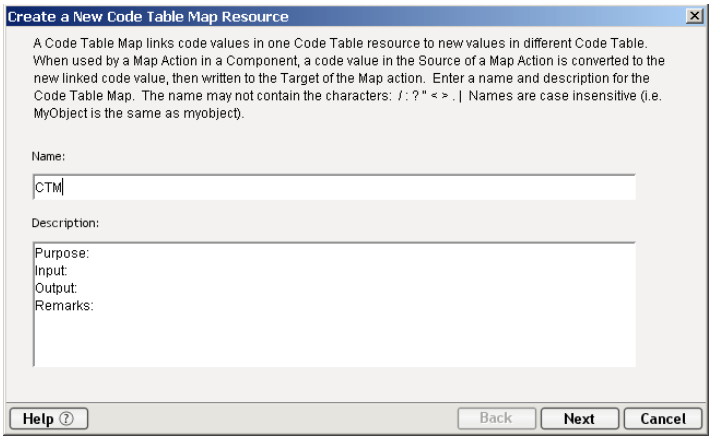
About Code Table Maps

A *code table map* is a resource used to automatically transform one set of codes into another set of codes. These maps are useful in translating and exchanging data between XML samples within a component. For example, one company may use numeric codes to store a status field while another uses alphabetic codes.

NOTE: You must create two individual code tables before you can create a code table map, since a code table cannot map to itself (See "About Code Tables" on page 201).

➢ **To create a code table map:**

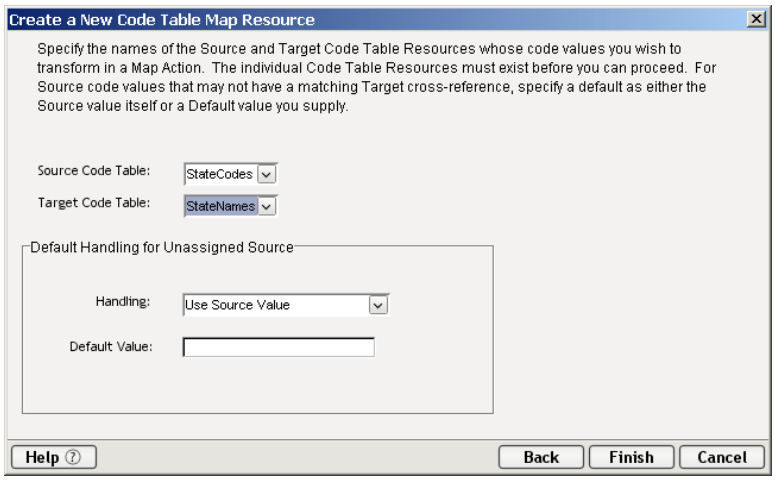
1 Select **File>New>xObject.** From the **Resource** tab, select **Code Table Map**. The Create a new Code Table Map xObject wizard appears.



2 Type in a **Name**.

3 Optionally, you my type in **Description** information.

4 Click **Next**. The second page of the Create a New Code Table Map xObject appears.



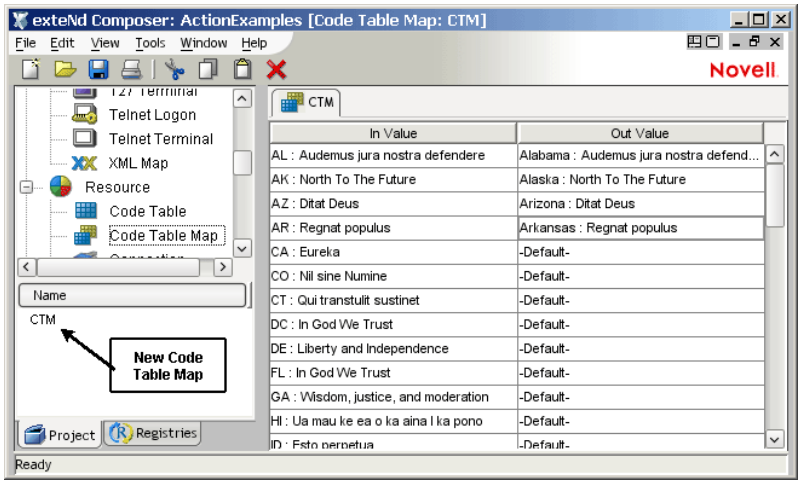
5 Select an **Input Code Table.** (These codes represent the data content as it will be received into a component.)

6 Select an **Output Code Table**. (These codes represent the desired code values.)

7 Select a **Handling** method. This feature allows you to instruct Composer on how to deal with values from the input Code Table that have no corresponding value in the output Code Table to which you are mapping. For example, if there are six values in Code Table 1 and only five values in Code Table 2, you must let Composer know how to deal with the additional value. You have two choices:

- **Use Source Value**—This choice simply uses the input value as the output value. For example, an input of “Warehouse1” would simply map to an output value of “Warehouse1.”
- **Use Default Value**—This choice would default to the value you set in the **Default Value** field. For example, you may enter “Not Applicable” in the **Default Value** field.

8 Click **Finish**. The newly-created code table map appears.



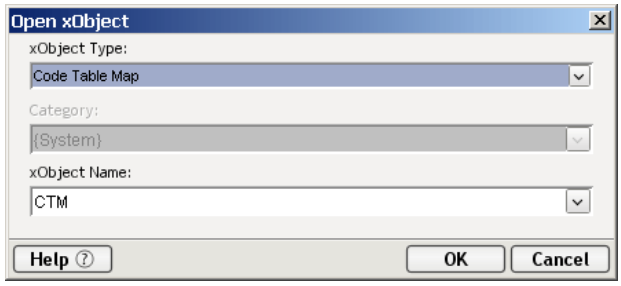
When you close the Code Table Map Editor, the newly-created code table map appears in the Resource category of the Instance Pane, under Code Table Maps, as shown above.

Mapping the Code Tables

Once you've selected the Input and Output Code Tables, you need to map the values. The code table map initially displays the **In Value** mapped to a Default setting in the **Out Value** field. The In Value is grayed out, since it cannot be edited. Once you click in a Default field, a dropdown list allows you to map the In Value to any one of the values in the Out Value field. This enables you to map more than one In Value to the same Out Value.

➢ **To open a code table map:**

1 Select **File>Open**. The Open xObject dialog appears.



2 Select **Code Table Map** from the **xObject Type** dropdown list.

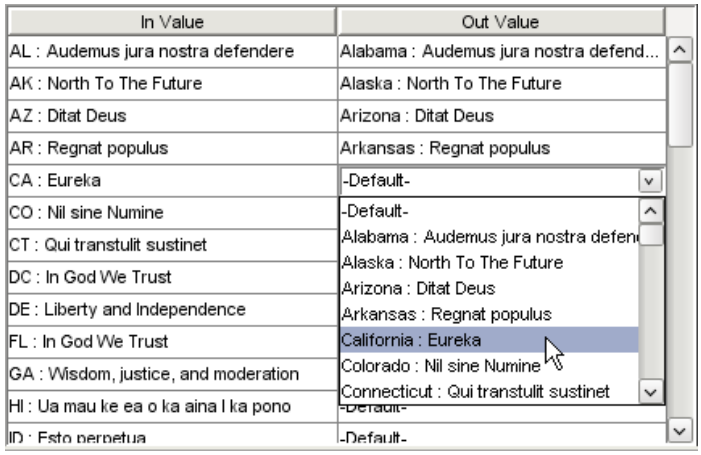
3 Select the code table map you'd like to open from the **xObject** dropdown list.

4 Click **OK**. The code table map you've selected opens.

➢ **To map values in the code table map:**

1 Open the code table map in which you'd like to map values.

2 Click the **Out Value** field in the first record. A dropdown list with all the available values from the Output Code Table appears.



3 Select the desired value from the dropdown list.

4 Repeat Steps 2 and 3 for all records.

5 Select **File>Save** or click the **Save** button on the tool bar.

➢ **To edit a code table map:**

1 Open the code table map to which you'd like to make edits (See "To open a Code Table Map" above).

2 Click inside the **Out Value** cell to which you'd like to make edits.

3 Select the new value from the dropdown list.

4 Select **File** then **Save** or click the **Save** button on the tool bar.

Using a Code Table Map

Once you've created a code table map, you use it as you build components. For example, in the XML Map component editor, you could create an action that would map an element from an input DOM via a code table map to an output DOM. The action might look like this:

By using the code table map in the Map action, you not only transfer the input data, but also transform it before placing it in the output.

See Chapter 7, "Basic Actions" for more information.

About Connections

A *Connection Resource* is a reusable object that wrappers connection-related information: typically an IP address, port number, and authentication credentials in the simplest case. The Connection Resource also stores critical information about driver names and/or JNDI names, LDAP distinguished names, time-out and retry settings, code pages, and/or whatever endpoint specifications might be needed to set up a connection with a given type of data store or stream.

Connection resources are needed not only for various data sources (such as database connections) but also for the URL File/Read, URL File/Write and XML Interchange actions—three of Composer's core actions. These Data Exchange actions allow you to transfer XML and non-XML documents via HTTP, HTTPS or FTP. The FTP Authentication Resource allows you to perform a simple FTP login and specify a connection time-out. The HTTP Connection Resource stores user-authentication and security information needed to set up an HTTPS session.

Some of the user-supplied information in a Connection Resource can be bound dynamically at runtime through the use of ECMAScript expressions. (See discussion below.) Not every piece of user info in the Connection Resource need be static.

Because Connection Resources specify detailed access information for the data stream or endpoint in question, you will generally need to create one Connection Resource for every type of data source that your component or service will use. For example, if your application requires you to interact with a database as well as a directory, you will need at least one JDBC Connection Resource and one LDAP Connection Resource. (You do not necessarily need one resource for each data store or system, however, since connection parameter values can be bound dynamically via ECMAScript; see the next section.)

It's worth noting that Connection Resources may be reused by multiple components. They are true *resources*.

About Constant vs. Expression Driven Connections

You can specify Connection parameter values in one of two ways: as constants or as expressions.

A *constant* based parameter uses the literal value you provide every time the Connection is used. An *expression* based parameter allows you to specify the value using an ECMAScript expression, which means the value might be different each time the connection is used at runtime. This late binding allows for flexible runtime behavior, since connection parameters can be determined using business logic or looked up from a backing store (including, potentially, an LDAP directory).

One simple use of an expression driven parameter in an HTTP Connection would be to define the User ID and Password as PROJECT Variables (e.g. PROJECT.XPATH("USERCONFIG/MyDeployUser"). At runtime, your service can populate these variables with values that are calculated or looked up on-the-fly (or derived from the service's input data); any connection resources used by any of your components can then obtain these values from the PROJECT variables at instantiation time (via ECMAScript).

A more sophisticated use of expression-driven parameters would be a case in which user credentials are looked up in a directory using LDAP queries. The procedure for doing this is described in the section called “Using LDAP to Obtain Connection Parameters” further below.

Setting Up an Expression-Driven Connection

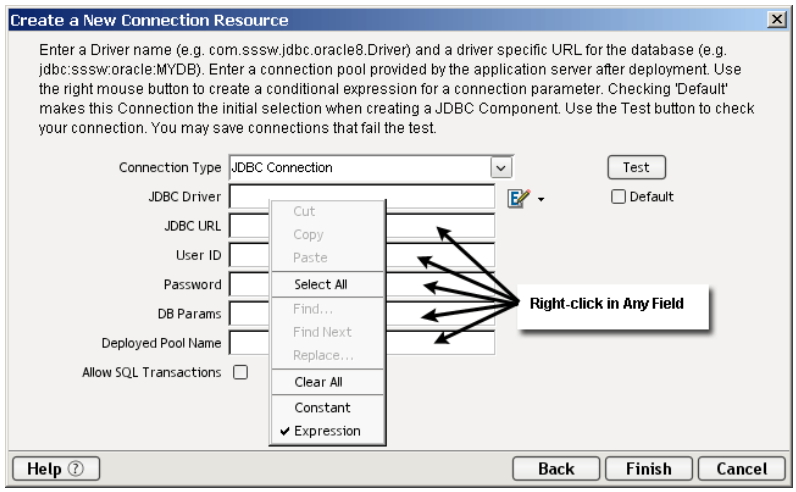
Any parameter in a Connection Resource (not only User ID and Password, but IP address, port number, etc.) can be expression-driven. The steps for setting this up are outlined below.

➢ **To switch a parameter from Constant to Expression driven:**

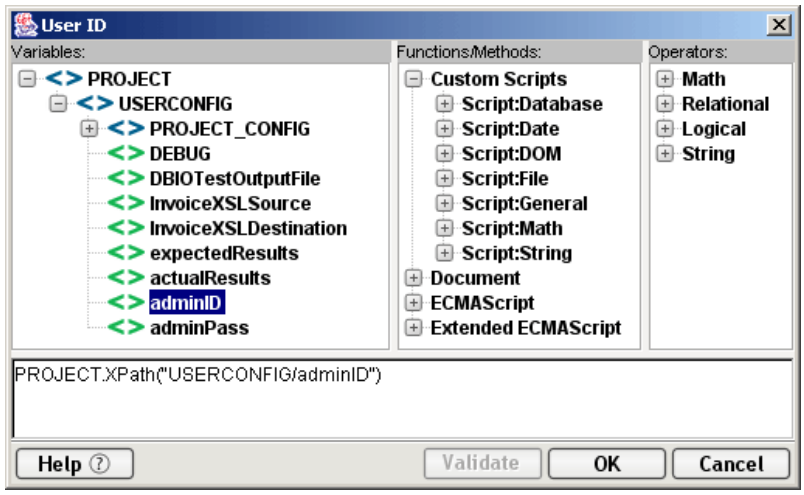
1 Click the RMB in the parameter field you are interested in changing.

2 Select **Expression** from the context menu. A flyout menu control will appear to the far right of the field, containing two buttons: an ECMA Expression button and an LDAP Expression button. If you will be looking up the connection parameter value from a directory, select the LDAP button. Otherwise, accept the ECMA Expression button (which is the default).

NOTE: For information on how to use the LDAP Expression button, see the section called “Using LDAP to Obtain Connection Parameters” further below.



3 Click the **ECMA Expression** button. The expression editor appears:



4 Use the expression editor to build an ECMAScript expression that will evaluate to a valid parameter value at runtime. (Note that most of the nodes in the various picktrees will autogenerate ECMAScript code for you if double clicked.) In the above example, a project variable (“adminID”) is consulted for the UserID value in a connection.

5 Dismiss the expression editor (click **OK**) to return to the connection-resource dialog.

6 Repeat the above steps as necessary for any other parameter fields to which you wish to apply expressions.

Using LDAP to Obtain Connection Parameters

User names and passwords are often stored in a directory (such as Novell eDirectory). The ability to look up user data at runtime is important in any application that has access control requirements. Composer allows you to leverage LDAP for this purpose. No matter what kind of back-end system you're connecting to, you can set up your Connection Resource in such a way that any or all of the connection parameters are obtained via directory lookup at runtime (with or without extra business logic to fine-tune connection particulars).

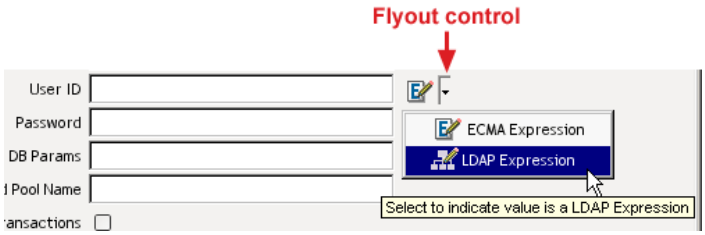
In order to obtain connection parameters by LDAP query, you must first create (or already have in your project) an LDAP Connection Resource. This resource tells Composer which directory to use, which port to go out on, the Base DN to use, etc., so that a connection (secure or non-secure) can be established to the target directory. (Detailed information on how to create an LDAP Connection Resource is contained in the *LDAP Connect User's Guide*.)

Once you have an LDAP Connection Resource, you can set up LDAP-driven connection parameters for any Connection Resource, using the technique outlined below.

➢ **To bind a connection parameter to a directory lookup:**

1 In the "Create a New Connection Resource" dialog, *right-mouse-click* inside the text field to which you wish to assign a directory-lookup value. A context menu will appear.

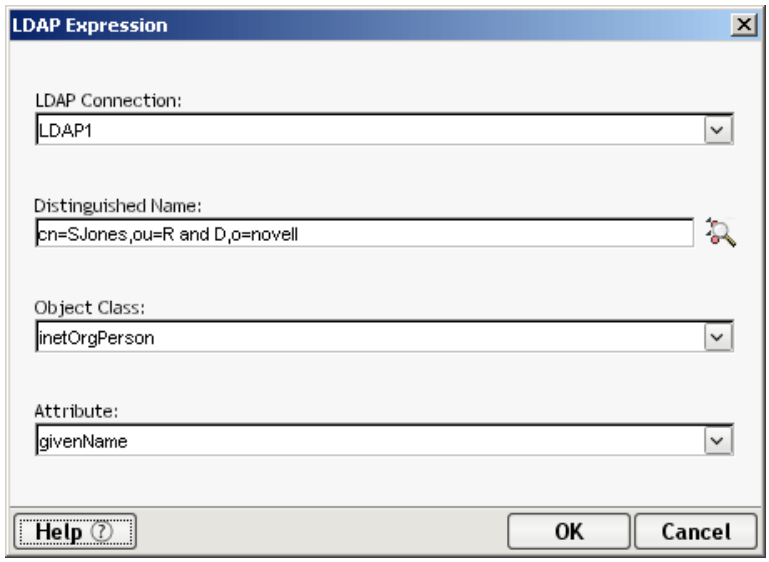
2 Choose **Expression** from the menu. A flyout control (note the small triangle) will appear at the far right of the text field in question.



3 Click the **flyout** and choose **LDAP Expression**. The button next to the flyout changes to the LDAP Expression Editor button.

4 Click the **LDAP Expression Editor button**. The LDAP Expression Editor window appears.

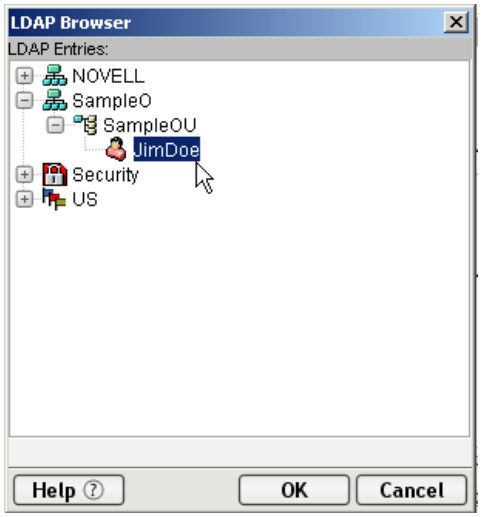
NOTE: You will get an error dialog at this point if you do not have at least one LDAP Connection Resource in your project.



5 At the top of this dialog, choose the **LDAP Connection** you will use. (The dropdown menu is pre populated with the names of LDAP Connection Resources in your current project.)

6 Under **Distinguished Name**, enter the LDAP Distinguished Name of the user or entity for which you are looking up data. If you don't know the proper LDAP syntax, use the DN Editor (also known as the LDAP Browser):

- Click the small **DN-and-pencil icon** to the far right of the Distinguished Name text field. A new screen appears:



- Navigate the directory's tree view until you get to the node (object) that contains the information you need. Click the node to highlight it.
- Click **OK.** You're returned to the LDAP Expression dialog. Notice that the DN field of the LDAP Expression dialog now contains the properly formatted Distinguished Name for the object you intend to query.

7 Under "Object Classes," select the type of object appropriate to the search you want to use, if it is not already showing.

NOTE: This control will normally already be showing the name of the object that corresponds to the DN you specified in the previous step. You should not have to do anything.

8 Using the Attribute pulldown menu control, select the name of the attribute that contains the data you wish to look up. (This control is prepopulated with the names of all attributes defined on the object class chosen in the previous step.)

9 Click **OK** to return to the Connection Resource setup dialog. An appropriately formatted ECMAScript expression will be generated for the connection-param field in question. The expression uses the Composer extension method `getLDAPAttr( )`. At runtime, the connection parameter will be determined dynamically by LDAP lookup. You can test this capability at design time, of course, either by clicking the Test button in the connection setup dialog, or by running your component (the component that uses this connection) in animation mode.

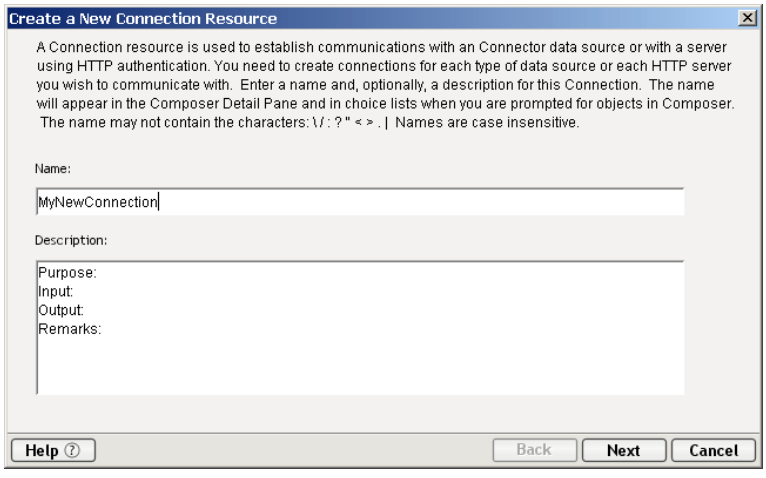
How to Create an HTTP Basic Authentication Connection Resource

Composer supports many types of connection resources, including LDAP, JDBC, FTP Authentication, HTTP (in three flavors, depending on the type of authentication specified), and SMTP in the core product; plus 3270, 5250, Telnet, HP3000, CICS RPC, JMS, SAP, Tandem, Data General, and Unisys-terminal connectivity in the various Connect products. Since the basic procedure for creating connection resources is the same for most of these different types, the following example (using HTTP Basic Authentication as the resource type) can be considered typical.

NOTE: See the user guide for the Connect product in question if you would like detailed information on how to create a connection resource for a particular back-end system, device, or protocol. Also, see the discussion at "Mail via SMTP Simple Authentication" (further below) for detailed instructions on how to create a mail-server connection resource.

➢ **To create an HTTP Basic Authentication Connection Resource:**

1 Select **File>New>xObject**. From the **Resource** tab, select **Connection**. The Create a New Connection xObject wizard appears.

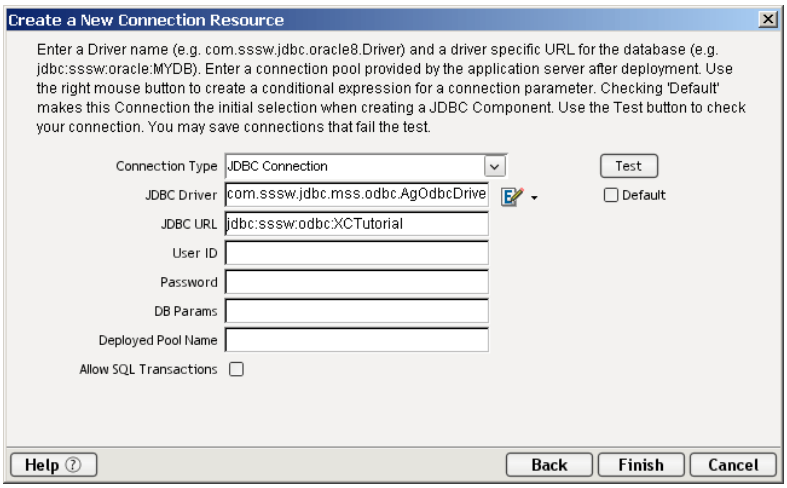


2 Type in a **Name**.

3 Optionally, type **Description** text.

4 Click **Next**. The second page of the Create a New Connection Resource wizard appears.

5 Select **HTTP Basic Authentication** from the drop down list. This will be used in conjunction with an XML interchange which uses HTTP connections.



6 Enter a **UserID** and **Password**. These are not actually submitted during the establishment of a connection. They are simply defined here (password is encrypted). The user will have access to UserID and Password variables from ECMAScript, allowing them to map UserID and Password as values into the screen. This way, no one ever sees the passwords.

7 Choose a **Client Certificate** by clicking on the **Browse** button and selecting the certificate file you want to use for this service connection.

8 Choose a **Client Private key** by clicking on the **Browse** button and selecting the client key file for security.

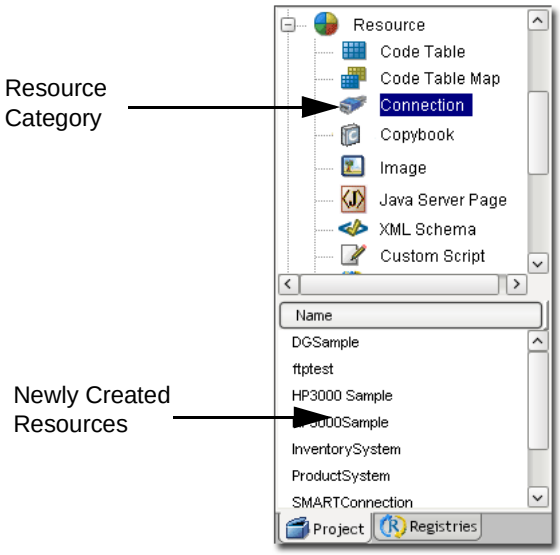
9 Enter the **Password** for the **Private key**. Private key is a another level of security for the owner of the Client Private Key.

10 Enter a **Connection Timeout** value in seconds.

11 Select the **Default** check box if you want this particular Connection to appear as the default connection in the appropriate Component wizards.

12 Click **Finish.** The connection is created.

NOTE: For more details on Connection Resources, see the “Getting Started” section in each Connect Guide.

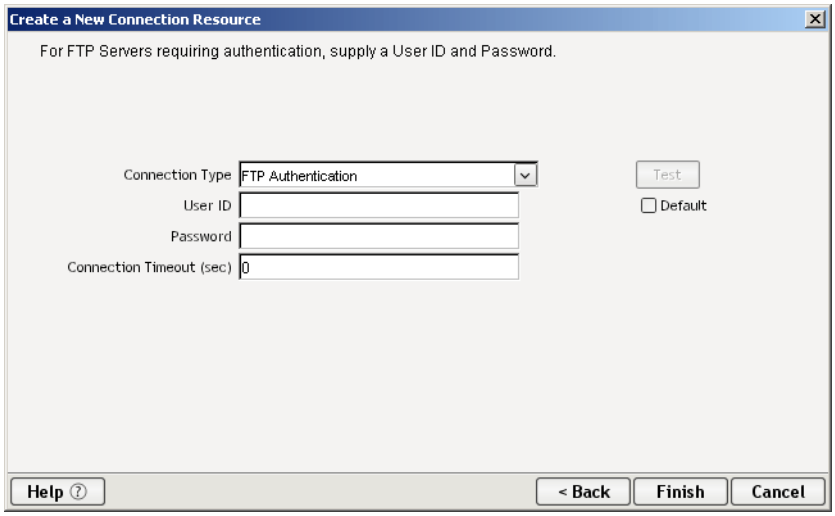


How to Create an FTP Authentication Resource

Most FTP connections require a user name and password. The FTP Authentication Resource wrappers basic credential information so that you can reuse the credentials as needed in various components that might use FTP to read or write remote documents. FTP access is supported, for example, in the XML Interchange Action. (See previous chapter.) In the setup dialog for that action, you can specify an FTP Authentication Resource to use when executing the action.

➢ **To create an FTP Authentication Connection Resource:**

1. Select **File>New>xObject**. From the **Resource** tab, select **Connection**. The Create a New Connection xObject wizard appears, as shown earlier (see Step 1 of procedure above).
2. Type in a **Name**.
3. Optionally, type **Description** text.
4. Click **Next**. The second panel of the Create a New Connection Resource wizard appears.
5. Select **FTP Authentication** from the drop down list. The panel changes appearance:



6. In the screen that appears, enter a **User ID**, **Password**, and **Connection Timeout** value (in seconds).

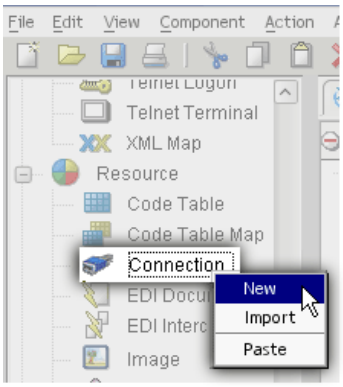
   NOTE: The timeout value represents the amount of time that will be spent trying to obtain a connection, not the amount of time devoted to keeping the connection open.

Mail Simple Authentication Connection Resource

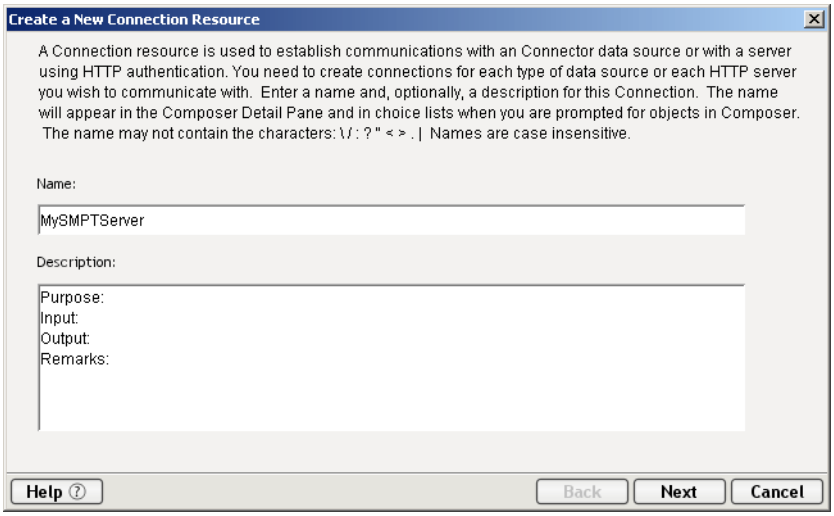
E-mail account information can be stored in a Mail Simple Authentication Connection Resource. Any of your services or components that make use of the Send Mail action (see “Mail via SMTP Simple Authentication” for details) can take advantage of a Mail Simple Authentication resource for obtaining account information.

➢ **To create a Mail Simple Authentication connection resource:**

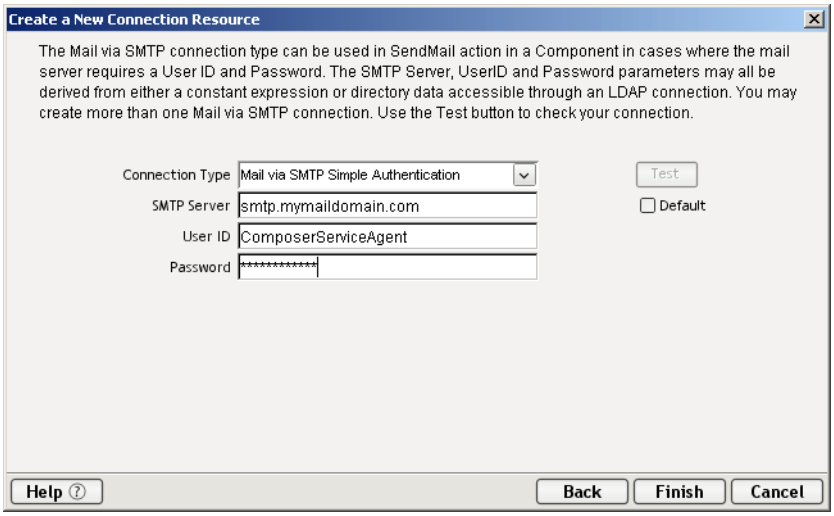
1 Under Resource in the navigation (explorer) frame, right-click on **Connection** and choose **New** from the context menu as shown below:



2 In the wizard pane that appears (see below), enter an arbitrary **Name** for this connection resource and (optionally) descriptive text.



3 Click **Next**. The second (and final) panel of the wizard appears:



4 Using the pulldown menu control, select Mail via SMTP Simple Authentication as the **Connection Type**.

5 Next to **SMTP Server**, enter the name or IP address of the mail server you intend to use.

6 Next to **User ID**, enter the user name associated with the mail account you wish to use.

7 Next to **Password**, enter the password associated with the user account in question.

8 Click **Finish**.

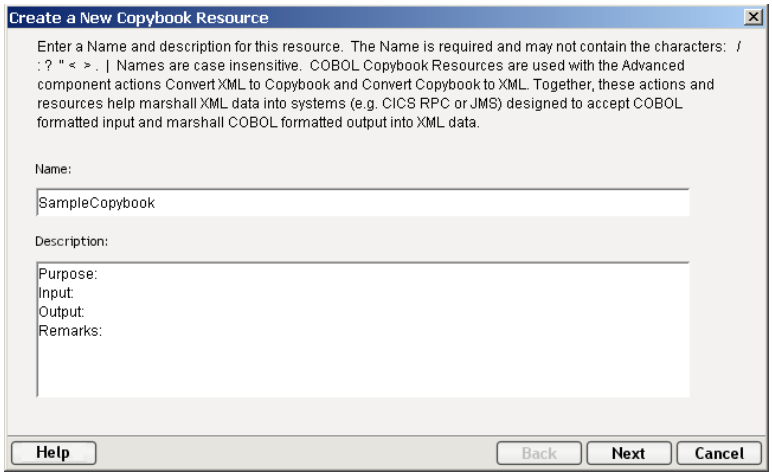
About Copybook Resources

If you are accessing a COBOL CICS mainframe or using a JMS system, your application may need to use a Copybook source file to define its data layout. Similarly, if your Composer project uses any kind of file manipulation (via FTP, EDI data exchange, etc.) you may have a need to work with COBOL Copybooks. exteNd Composer has a Resource called Copybook that allows you to convert XML data into a ByteArray that can be used as Input to CICS RPC or JMS components. It can also be used to convert the ByteArray Output from these components back into XML format using Convert actions, which are described in Chapter , “Advanced Actions” beginning on page 160.

➢ **To create a Copybook resource:**

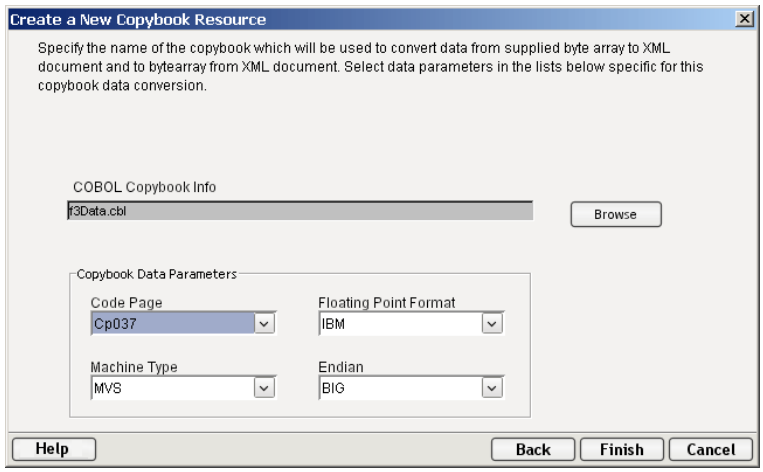
1 Select **File>New>xObject**. From the **Resource** tab, select **Copybook**. The Create a New Copybook wizard appears.

NOTE: Alternatively, you can select **Copybook** from beneath the **Resource** tree in the Navigator pane and click on **New**.



2 Type in a **Name**. Add Description information if desired.

3 Click **Next**. The Copybook parameters screen appears.



4 Use **Browse** to search your file system for a COBOL Copybook.

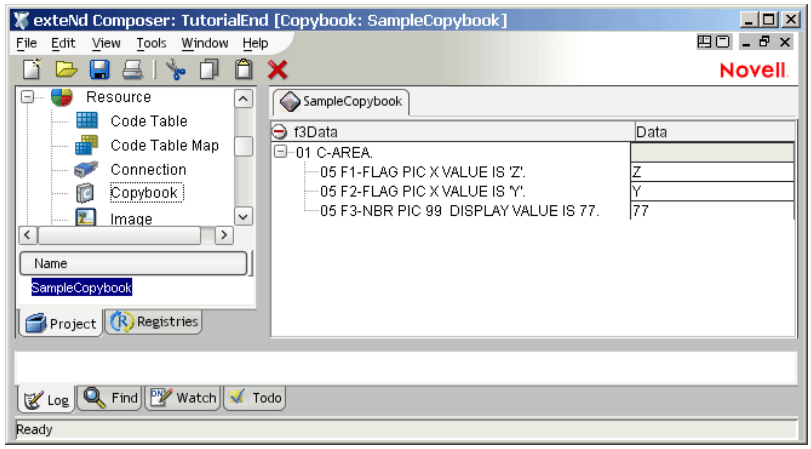
5 In the **Code Page** field, from the drop down menu, select the appropriate code page for the type of operating system character data standard your machine uses (i.e., CP037 for EBCDIC or 8859_1 for ASCII).

6 In the **Machine Type** field, from the drop down menu, select the platform of your CICS Region/Server (MVS, OS2, NT, AIX).

7 In the **Floating Point Format** field, from the drop down menu, select a name dependent upon the machine type selected: IBM or IEEE.

8 In the **Endian** field, from the drop down menu, select the order of the most/least significant bytes in integers (BIG if the most significant byte precedes the least significant byte in memory, otherwise select LITTLE).

9 Click **Finish** to add the Copybook Resource to your list of available Resources and open it in the Component Editor. An example of an open Copybook Resource is shown below:



NOTE: Copybook Resources must be created prior to the use of any Convert XML to Copybook or Convert Copybook to XML Actions.

About Custom Script Resources

A Custom Script Resource is a library of user-developed functions created in the ECMAScript programming language. You can make the functions available to be used throughout components and within other functions. Using custom scripts, you can develop functions that perform:

- Almost all the same functionality as the basic XML Map components, with your own customizations
- Data manipulations involving Strings, dates, numbers, regular expressions, etc.
- XML document manipulations using the W3C ECMAScript-to-DOM Binding methods
- Integration with standard or custom Java classes

NOTE: You must have a thorough understanding of the ECMAScript language in order to create custom functions. The following sections are intended only as general guidelines, not a tutorial in scripting. For more information on scripting in Composer, see the next chapter.

Organizing and Using Custom Functions

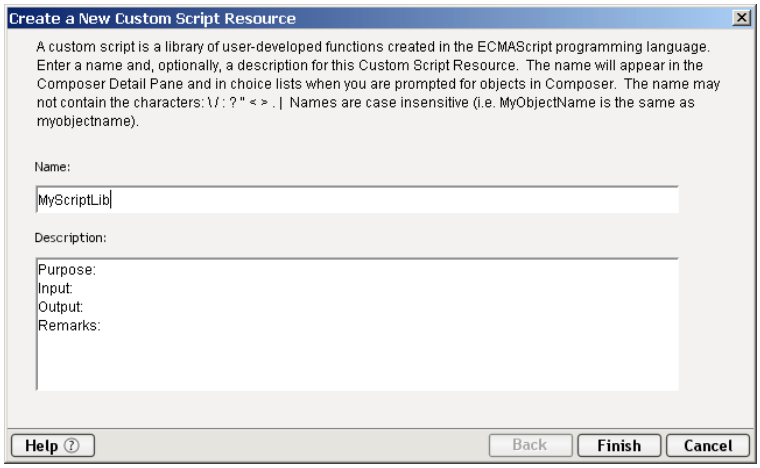
You may prefer to organize functions into different libraries. For example, you may have several math, string, or database functions that you'll need for your application. If you group similar functions (i.e., create all string functions in the same library), you can also use the Custom Script Editor to declare global variables that can be used by all functions within the same library.

As you create and validate functions, Composer makes them available in all expression editors within component actions.

For example, if you write a custom function library called "String" containing ten functions, they will appear in the Expression Editor under the Custom Scripts label with the other standard functions.

➢ **To create a custom script:**

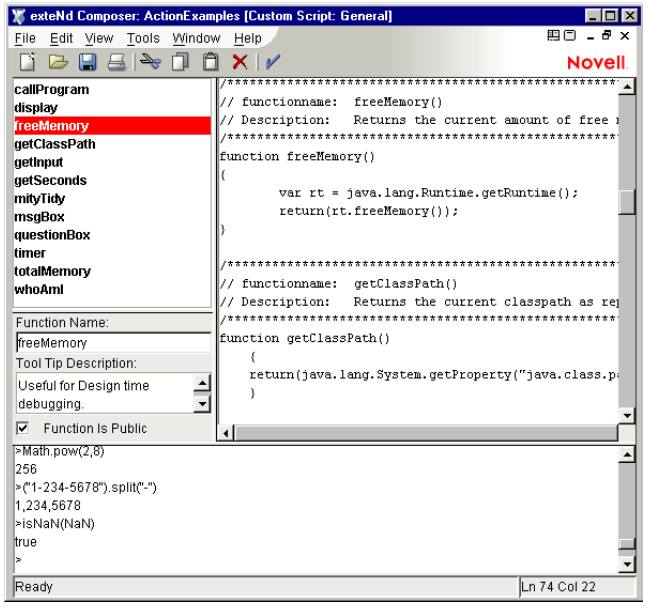
1 Select **File>New>xObject**. Then, from the **Resource** tab, select **Custom Script**. The "Create a new Custom Script xObject" dialog appears.



2 Type in a **Name**.

3 Optionally, you can type in **Description** information.

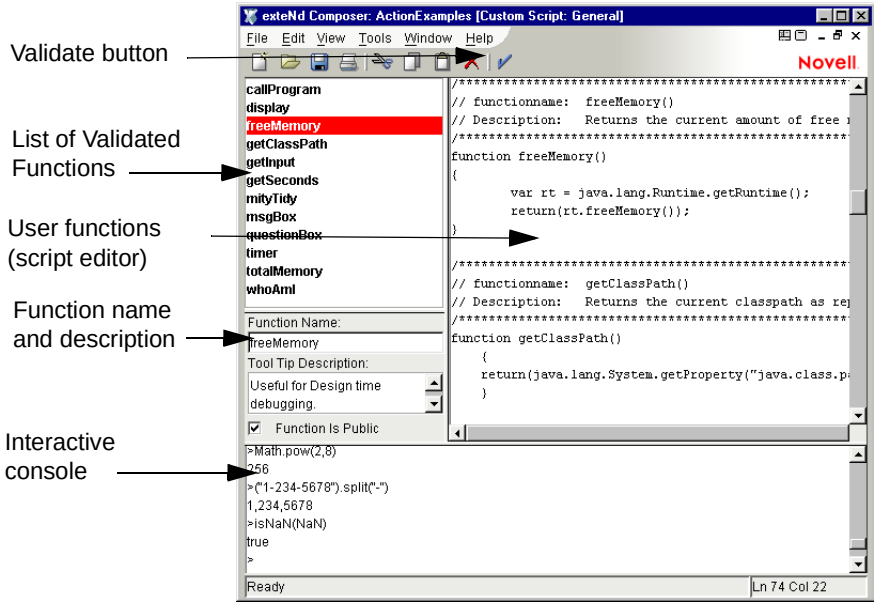
4 Click **Next**. The Custom Script editor appears with your newly-created Custom Script name in the title bar.



About the Custom Script Editor Window

The Custom Script Editor window is divided into several panes. You can change the view of the panes to include the content you need.

The illustration below shows the editor after several functions have been added.



Note: In the above view, the Output (Error Message) Pane has been hidden with Control-Shift-O and the Navigation Frame has been hidden with Control-Shift-N.

Creating and Validating a Function

You create a function by typing it in from scratch. You can also use the Expression Builder in creating your function. For more information, see “Using the Expression Editor to Build Functions” on page 225.

➢ **To create and validate a function:**

1. Type the word *function* in the function creation area.
2. On the same line, type the function name after the word *function*.
3. On the same line, type any function parameters, separated by commas, and enclosed with parentheses.
4. Type a left curly brace and press Enter.
5. Type in the function statement(s).
6. Type a right curly brace and press Enter.
7. Add comments to the function, if desired.

Your function should look similar to the following example:

```
/*****************************************************************************/
// functionname: chr(aiCharCode)
// Description:  this function will return the ansi character for an integer value
// aCharCode:    is required, any integer value
// Returns:      ansi character for the value of the integer
// Note:         Unicode Values 0x20 to 0x7E and 0xA0 to 0xFF correspond to ASCII
/*****************************************************************************/
function chr(aiCharCode)
{
    return String.fromCharCode(aiCharCode)
}
```

➢ **To validate the syntax of your function:**

- Click the **Validate** button.

If your function is valid, Composer adds the function name to the validated function list. If your function contains an error, Composer presents a detailed error message.

➢ **To test your function:**

1. Type the function name complete with valid parameters in the test area.
2. Press Enter.

Before you can test your function, it must pass the syntax validation described in the previous section.

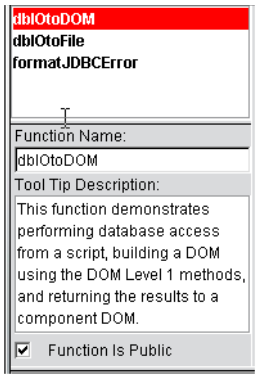
Adding a Function Tool Tip Description

Once your function has been validated, and is added to the validated function list, you can write a description for it. The description appears as a “tool tip” when your mouse rests on the function name, wherever the function appears in Expression Builders throughout Composer.

➢ **To add a description:**

1 Create and validate a function.

2 In the Description text box, overwrite the default description with the text for your function, as shown.



3 Select the **Function is Public** check box as desired. When you check this box, two things happen:

- The function can be used from any expression builder in any action.
- The function appears in the expression builder pick-lists under the "Custom Scripts" heading.

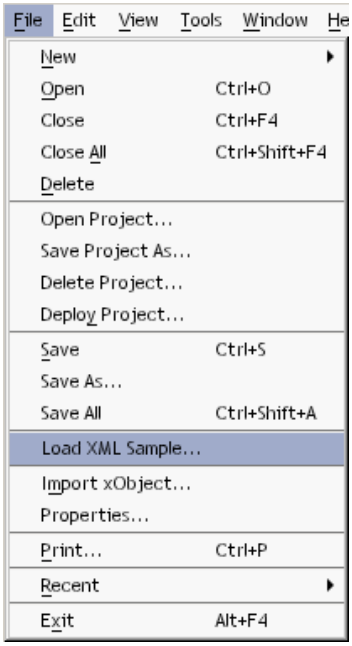
Viewing DOM Trees within the Script Editor

For many of the custom script functions that you create, you will want to reference or work directly with data in specific XPath location in your XML documents. To make this easier, the Custom Script Editor allows you to display XML documents (in any of the three available views: text, tree, or stylized) in the editor. This makes specifying XPath references easier by allowing you to *drag and drop XML elements into the body of your function definition*.

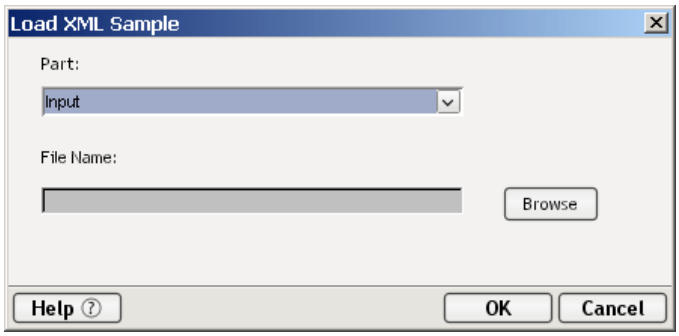
➢ **To show an XML document in its own pane:**

1 Be sure you have opened a Custom Script and are in the Script Editor environment.

2 In Composer's main menus, select **File** then **Load XML Samples**. (See illustration below.)



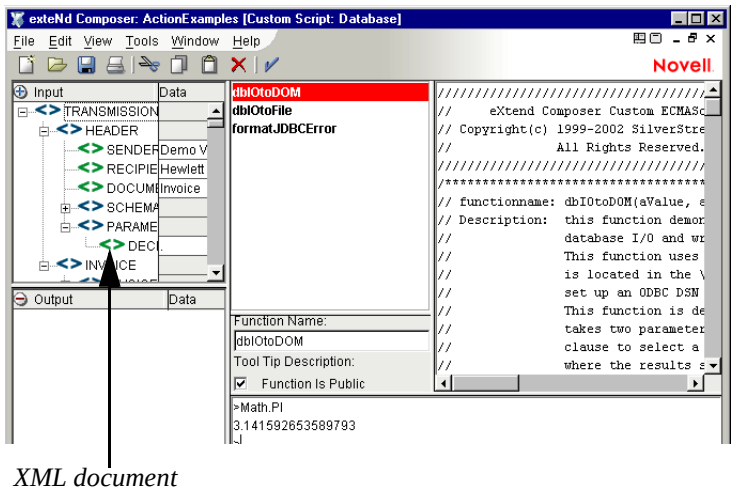
When you choose this command, the Load XML Sample dialog will appear:



3 In the pulldown menu control labeled **Part**, choose which DOM (Input or Output) to associate the file with.

4 Use the **Browse** button to bring up the file-navigation dialog. Navigate to the XML file you wish to load, and dismiss the navigator. If you wish to load in a file from a URL, you must explicitly type "http://," "https://," or "ftp://."

5 Click **OK** to dismiss the Load XML Sample dialog. The file you chose appears in its own pane.



XML document appears in its own pane

6 Navigate to the directory of the XML document you wish to use and select a file. An Input and Output Mapping Pane appear.

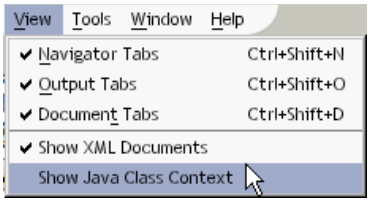
NOTE: If you want to use XML documents from your XML Templates, go to the appropriate "Imports" directory below the "XMLCategories" directory in your project (e.g.: **/Tutorial/XMLCategories/OfficeSupply/Imports**).

Integrating Java Classes with Custom Scripts

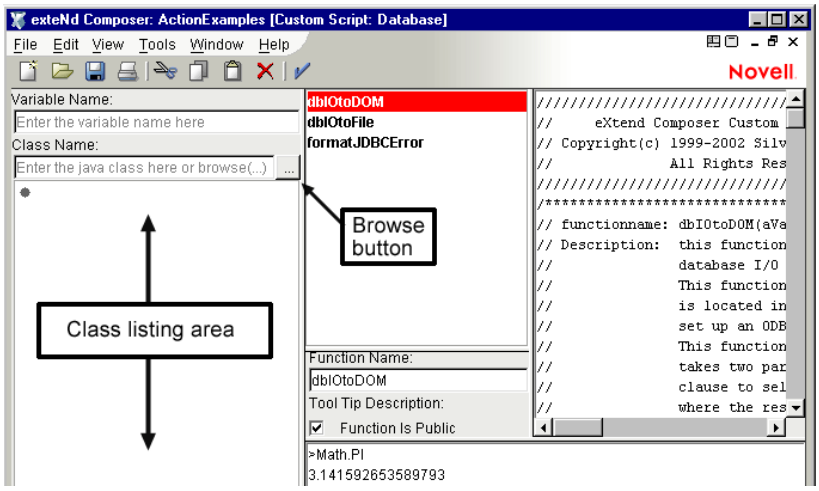
If you are building a custom script that needs to invoke Java methods or instantiate custom Java classes, you can expand the view for the Custom Script Editor window to show the information you need in a browsable class navigator. The Java class browser scans your current CLASSPATH (as well as any JAR Resources you have added to your project) and displays the classes, methods, and properties it finds. This makes specifying and using Java constructors, methods, and properties easier by allowing you to *drag and drop these items into the body of your function definition*, or into the test (console) area, to test your functions.

➢ **To use Java classes:**

1 Select **View** > **Show Java Class Context** from the Custom Script Editor menubar.

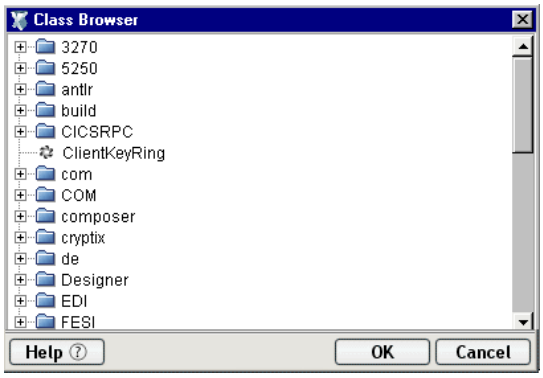


After choosing this command, the Java Class panel appears in the main editor.



2 In the Java Class panel:

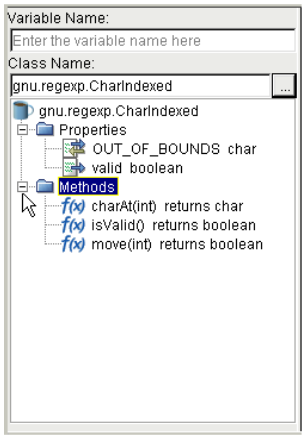
- Type a name in the **Class Name** field, *or*
- Click the **Browse** button. After a brief delay, the Class Browser dialog will appear, showing the Java packages available in the Composer CLASSPATH.



3 Navigate the context tree to get to the class you want to use. (Click the small plus signs to the left of the context nodes to expand them.)

4 Select the class you want to use, then click **OK** to dismiss the dialog. The class becomes visible in the class-list area of the editor.

5 Expand the class (by double clicking it or clicking the adjacent plus sign) to show its constructors, methods, and properties. See below.



6 (Optional) Drag and drop individual methods or properties into the editor pane or the console to use the properties in question.

If you want, you can add your own classes into the Class Browser by either putting them into the Composer CLASSPATH or extending Composer's CLASSPATH to include your classes.

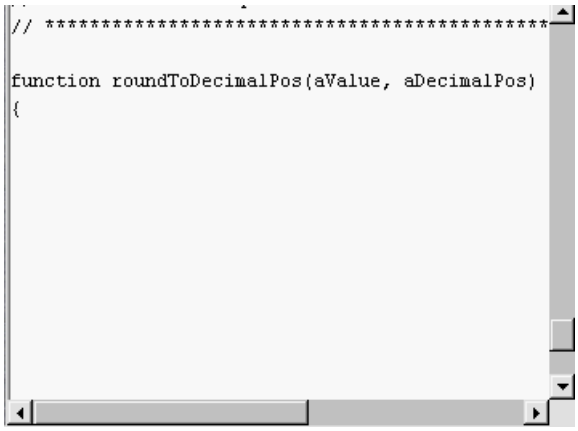
NOTE: If you are using custom Java classes, be sure to install those classes on the Application Server (or include them in your EAR/WAR files) when you deploy the application. You can use exteNd Director to do this. See the Director documentation for details.

Working with a Java Class in ECMAScript

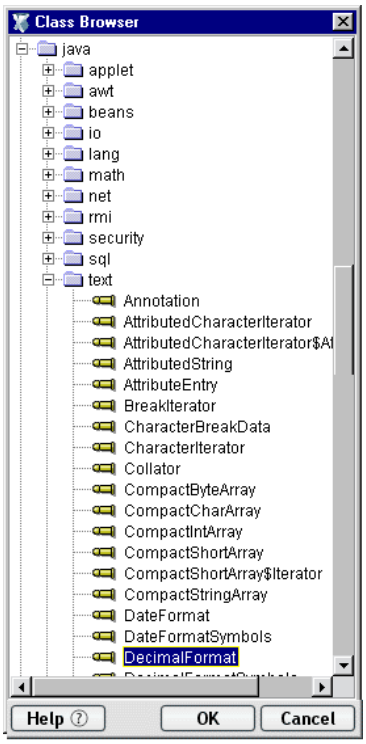
The following example shows you how to create a script function named `RoundToDecimalPos()` that uses the Java `DecimalFormat` class. In this example, your function accepts two parameters, a number to round, and the number of places to round.

➢ **To create a script function that uses Java:**

1 With the Java class panel displayed, enter a function signature in the function pane as shown.

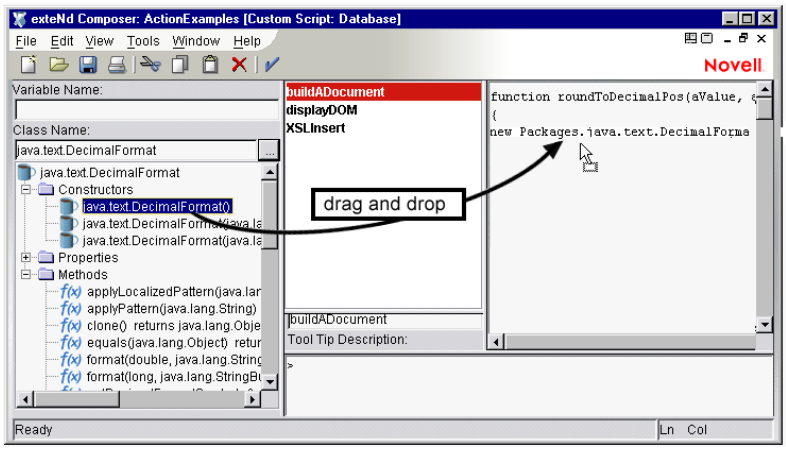


2 In the **Class Name** control, select the Browse button. The Class Browser dialog appears.



3 Navigate to **Java > Text > DecimalFormat** as shown above.

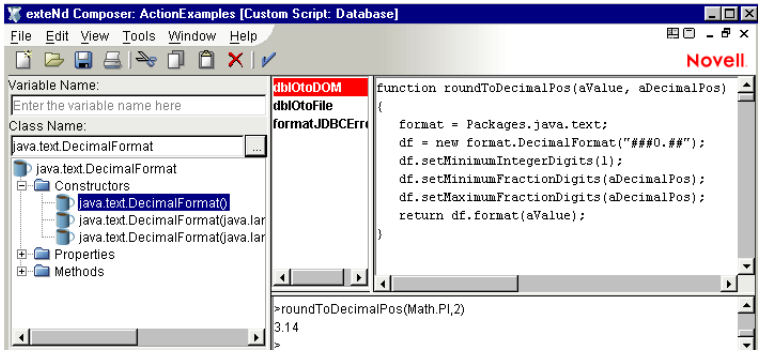
4 Click **OK**. The Custom Script window's **Class Name** field is now populated.



5 (Optional) Enter a name in the **Variable Name** field.

6 Drag and drop the desired **Constructor** to the function pane. Fill in the parameters for the constructor.

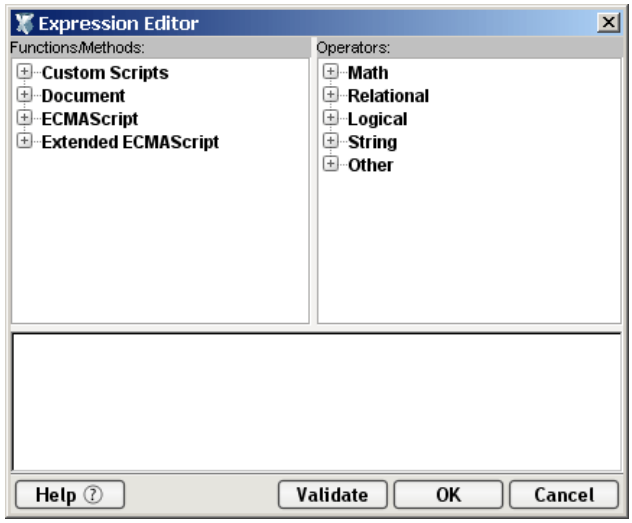
7 Edit your ECMAScript function as desired. One possible function is shown below.



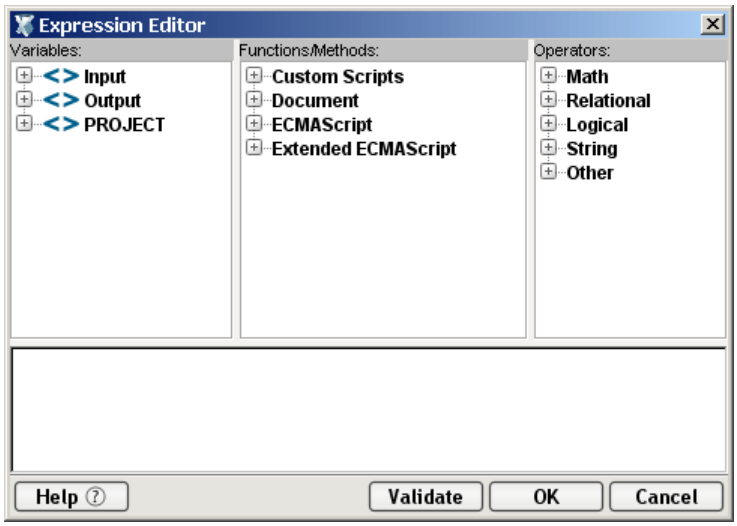
Using the Expression Editor to Build Functions

Rather than writing functions from scratch, you can use the Expression Editor to build them. The advantage of using this feature is that the Expression Editor exposes virtually all DOM methods, Composer extensions, built-in ECMAScript methods, and DOM node targets, via point-and-click pick-lists. Building an expression by the use of pick-lists is not only convenient and quick, but less prone to result in typos. It's also a useful reference, since the calling syntax for every available method is shown in rollover tooltips for each leaf node in the picktree(s).

The Custom Script Editor displays two different views in the Expression Editor, based on the view you select. The basic view lists the ECMAScript objects and operators available for building your own functions, as shown.

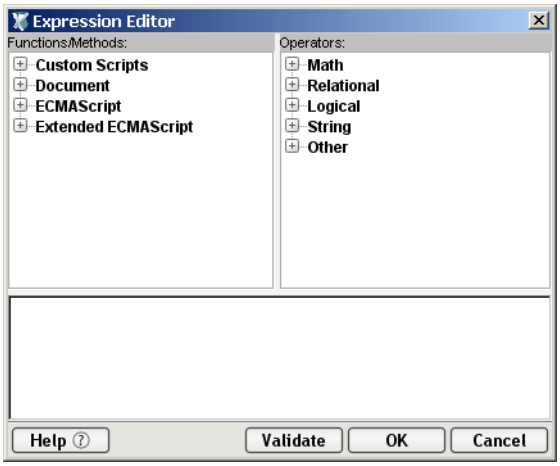


If you select, **View**, then **Show XML Documents** in the Custom Script Editor, the Expression Editor appears with an additional pick-list for selecting elements in the DOMs, as shown.



➢ **To use the Expression Editor:**

1 From the main menubar, select **Tools** then **Expression Editor**. The Expression Editor appears.



2 Expand the trees in the **Variables, Functions/Methods** or **Operators** panes and doubleclick on the elements to build your functions. Once you doubleclick an element, it appears in the bottom Expression pane.

Alternatively, you can use the right-mouse button to bring up the Expression builder in either the Function editing pane or the test area.

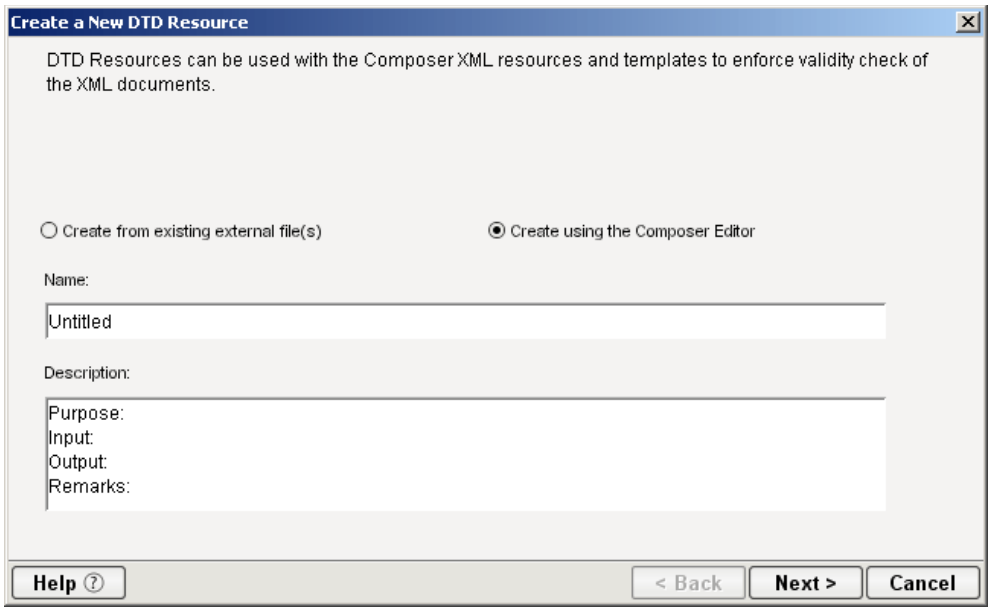
NOTE: For additional tips regarding ECMAScript, see the subsequent chapter called “Custom Scripting and XPath Logic in exteNd Composer” in this guide.

About DTD Resources

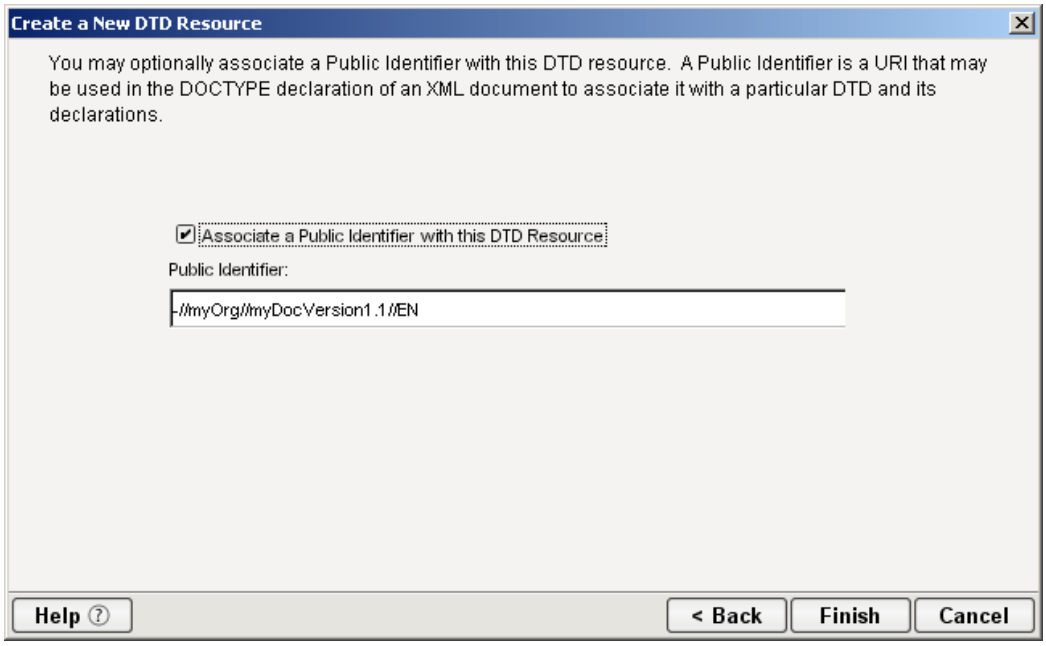
You can package DTDs into your project for deployment as part of your application. The procedure is as follows.

➢ **To create a DTD resource:**

1 From Composer's **File** menu, select **New**, then **xObject**, then from the **Resource** tab, select **DTD**. (Alternatively, right-click on the DTD Resource icon in the Category pane, and choose **New**.) The first pane of the DTD Resource wizard appears.



2 Provide a **Name** by which you will refer to the DTD, add descriptive information if desired, and (if you are creating this DTD using the Composer Editor) click on **Next** to proceed to the next screen.



3 Optionally click the "Associate Public Identifier" checkbox and enter the appropriate identifier string in the text field shown.

4 Click **Finish**. The Composer Editor opens with your DTD showing in text view.

About Form Resources

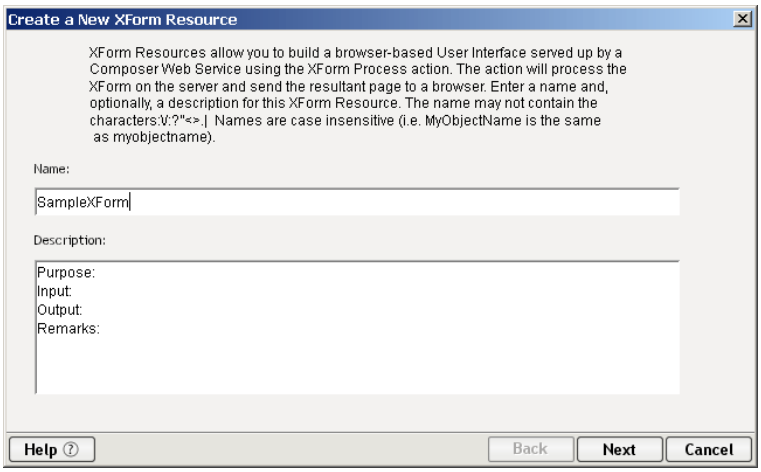
The Composer Form Resource gives you the ability to create XML-based forms (XForms) for use within your project.

NOTE: When installed as part of the Novell exteNd Professional Edition suite, Composer does not support this resource type. The following discussion applies only to users of the Enterprise Edition product.

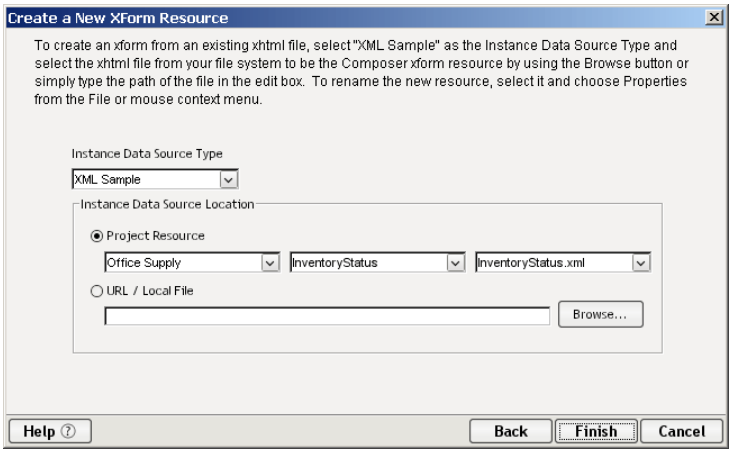
The Form wizard will guide you through the necessary steps to create the initial instance data.

➢ **To create an XForm:**

1 From Composer's **File** menu, select **New**, then **xObject**, then from the **Resource** tab, select **XForm**. (Alternatively, right-click on the XML Schema Resource icon in the Category pane, and choose **New**.) The first pane of the Form Resource wizard appears.



2 Provide a **Name** by which you will refer to the XForm, add descriptive information if desired and click on **Next** to proceed to the next screen.



3 Select an **Instance Data Source Type.** Choices include:

- Schema
- XML Sample
- WSDL

4 If you select a Data Source Type of **Schema**, you will need to indicate which **Schema Root** should be used for the instance data.

5 If you select a Data Source Type of **XML Sample**, select the Location of the XML Source using the dropdown lists under **Project Resources** for templates that already exist as part of the project. The first box contains Template Categories, the second contains Template Names, and the third box will be populated with Sample Names.

   Alternatively, you can **Browse** to select a local file or type in the fully qualified **URL** to locate the XML Source for the XForm.

6 If you select a Data Source Type of **WSDL**, you must select a **Service** and and an **Operation** that are associated with the instance data.

   Then, select a WSDL Resource using the dropdown below Project Resource. Alternatively, you may **Browse** to select a local file or type in a fully qualified **URL** to locate the WSDL Source for the XForm.

7 Click on **Finish** to create the Form Resource object and open it in the Forms editor.

NOTE: The forms editor interface is described in detail in the exteNd Director documentation and help.

About Image Resources

It is sometimes useful to package image files into a Composer project. For example, if your project contains Java Server Pages that reference GIF, JPEG, or PNG files, it's often convenient to package images within the same JAR or WAR file as the JSP that uses them. The JSP can then refer to the images via a relative URL (as described below).

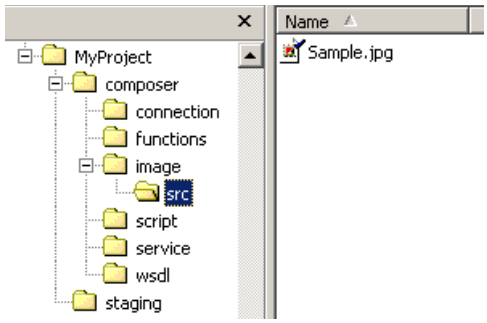
NOTE: When installed as part of the Novell exteNd Professional Edition suite, Composer does not support this resource type. This discussion therefore applies only to users of the Enterprise Edition product.

The first time you create an Image resource, Composer creates a subdirectory called **\image** in your project folder and puts a "src" folder under the **image** directory. Every image resource you create results in two files:

- An XML file describing the resource
- A copy of the original image

The former appears in the **\image** subdirectory. The latter appears in the **\src** subdirectory.

So for example, if your project is called **MyProject** and you created an Image resource based on an image called **Sample.jpg**, you would be able to find the following directory structure:



Inside the **\image** folder, you would also find a file called **Sample.jpg.xml**, representing the xObject wrapper (the metadata) associated with the **Sample.jpg** resource.

Image Resource Naming (and Renaming)

By default, when you create an Image resource, it acquires a name identical to that of the image you are assigning to the resource. Nevertheless, you can rename the Image resource after creating it (the same way you would rename any other resource): Just right-mouse-click on the resource instance in the object pane in Composer's main view, and choose **Rename** from the popup menu that appears. Then enter a new name for the resource. (Renaming the resource in this way changes the name of the actual image file as well as the xObject.)

NOTE: As with other resource renaming operations, you will need to *close* the resource in question (if it is open) before you can rename it.

Context in the JAR

At deployment time, your Image resource will be packaged inside a JAR file (along with all the other xObjects in your project) and placed inside the deployment EAR. Each physical image will have a default deployment context of

[Your project context]/image/src

Therefore, any Java Server Page that lives at root level in the JAR can refer to a given image using a relative URL of **/image/src/*imagename***. For example:

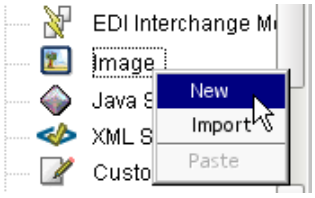
```
<img src="MyProject/image/src/Sample.jpg"/>
```

How to Create an Image Resource

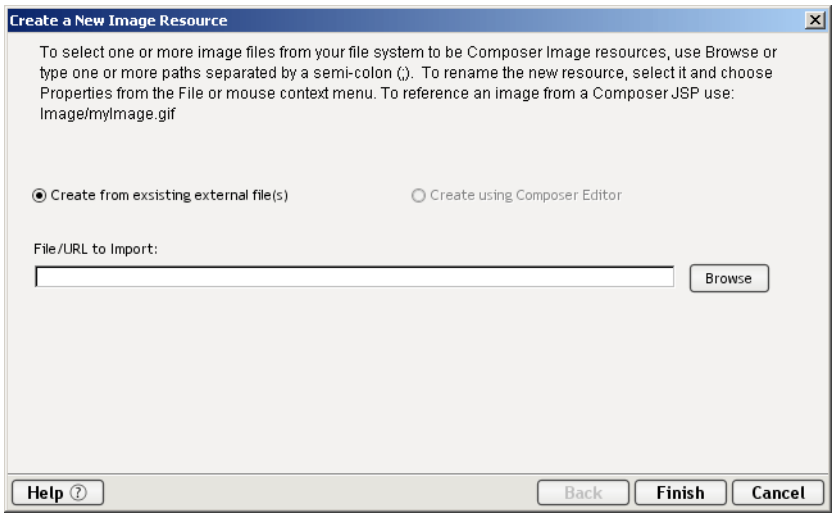
Creating an Image resource is much like creating any other resource xObject, except that in this case, the object's default name is chosen for you.

➢ **To create an Image resource:**

1 Either right-mouse-click on the word Image under Resources in Composer's navigation frame and choose the New command from the context menu (as shown below), or go to the **File** menu and select **File > New > xObject**. From the **Resources** tab, select **Image** and **OK**.



2 In the dialog that appears, choose the radio button labelled **Create from existing external file(s)**. See below.



3 If you know the location of the file, you can enter it directly in the text field (either as a file-system address or a fully qualified URI beginning with **http://** or **ftp://**). Otherwise, click the **Browse** button and navigate to an image file.

NOTE: Supported file types are GIF, JPEG, and PNG.

TIP: When using the file-chooser dialog (via Browse), you can Control-click or Shift-click to select multiple images; then all will be brought into Composer at once. Each image will retain its original name.

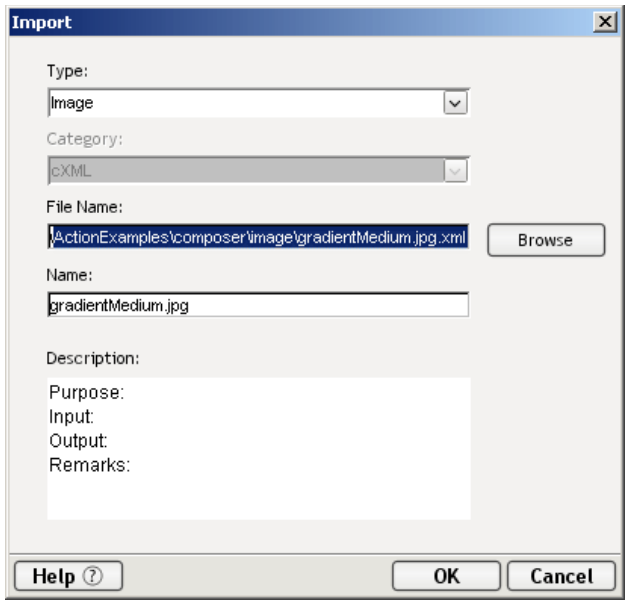
4 Click the **Finish** button. The dialog goes away and a new Image resource appears in the instance pane of Composer's navigator frame.

How to Import an Existing Image Resource

You may find that you want to import an existing Image resource from another project into your current project. You can do this as follows.

➢ **To Import an Image resource from another project:**

1 Right-mouse-click on the Image resource category (as shown in the last section) and choose **Import** from the context menu. The Import dialog appears:



2 Use the **Browse** button to go to a file-chooser dialog. Browse your network or file system as necessary, and when you have located the Image resource you wish to import, click Open to return to the above dialog. If you wish to load in a file from a URL, you must explicitly type "http://," "https://," or "ftp://."

NOTE: Image resources are XML files. They will always be found in the **\image** folder of a Composer project.

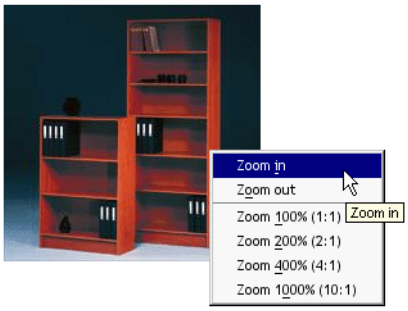
3 The name of the resource is shown in the **Name** field of the dialog. Use this text field to change the resource's name if you wish to do so at this time. (You can also rename it later.)

4 Click **OK**. The resource is added to the instance pane of Composer's navigation frame.

How to View an Image Resource

Once you have created or imported Image resources into your project, you can view an image by double clicking the resource in the instance pane (or by highlighting it and using RMB, then **Open**). The image will be rendered in its own tabbed window in the Composer desktop:

The size of the view (the magnification factor) can be controlled in various ways. If you right-click on the image itself, a context menu appears:



In addition to using the various Zoom commands on the context menu, you can control the magnification factor of the image view by means of the mouse and/or keyboard:

To zoom in: Use the plus (+) key on the numeric keypad, or left-click anywhere on the image.

To zoom out: Use the minus (-) key on the numeric keypad, or Control-left-click anywhere on the image.

To restore the original view: Use the equal sign (=) key or Shift-left-click anywhere on the image.

About JAR Resources

NOTE: When installed as part of the Novell exteNd Professional Edition suite, Composer does not support this resource type. The following discussion applies only to users of the Enterprise Edition product.

JAR resources provide you with the ability to add custom utility classes or business objects to a Composer project. Custom scripts and components you create within your project can then use these classes. The Custom Script editor's class browser provides access to the JAR resource, allowing you to drag and drop Java objects in to your custom scripts. You may also use function calls within your components to instanter objects of the imported class type, and invoke the class methods.

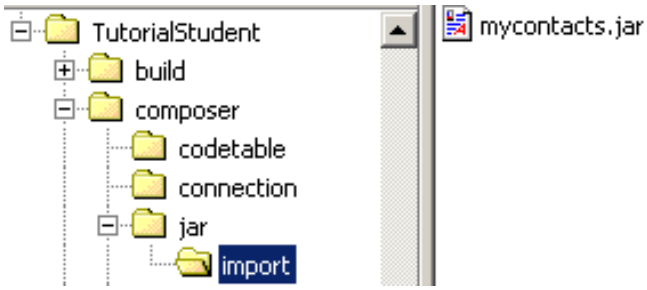
NOTE: If you wish to add another Composer project as a subproject, do not use the JAR Resource mechanism. Instead, use **Tools > Project Serttings > Subprojects** to import another Composer project into your current project.

The first time you create a JAR resource, Composer creates a subdirectory called **\JAR** in your project folder and puts an import folder under the **JAR** directory. Every JAR resource you create results in two files:

- An XML file describing the resource
- A copy of the original JAR

The former appears in the **\JAR** subdirectory. The latter appears in the **\import** subdirectory.

For example, if your project is called **TutorialStudent** and you created a JAR resource based on a JAR called **mycontacts.jar** you would be able to find the following directory structure:



Inside the **\jar** folder, you would also find a file called **mycontacts.xml**, representing the xObject wrapper (the metadata) associated with the **mycontacts.jar** resource.

JAR Resource Naming (and Renaming)

By default, when you create an JAR resource, it acquires a name identical to that of the JAR file you are assigning to the resource. You can rename the JAR resource after creating it, if you wish. From Composer's main view click the right mouse button on the resource instance in the object pane. Choose **Rename** from the popup menu that appears. Then enter a new name for the resource. Renaming the resource in this way changes the name of the actual JAR file as well as the xObject.

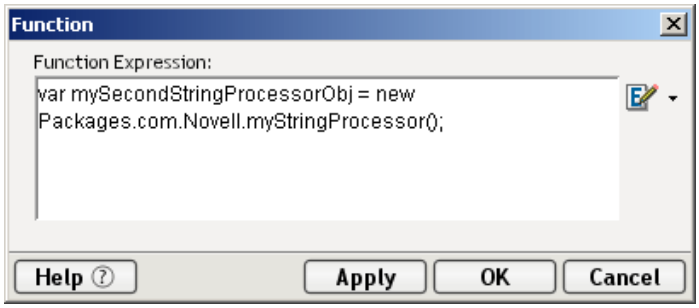
NOTE: The JAR file resource must be closed in order to rename the resource.

Context in the Composer Project

To reference classes in the JAR resource you reference the context of the class package. If the class is in the root of the jar no context is needed. In the example below two jar object classes are referenced, the first myFirstStringProcessorObj references a class at the root of the JAR. The second, mySecondStringProcessorObj is referenced by its context, com.Novell. Notice that in both cases the Packages keyword precedes the context. To avoid name collisions between classes with identical method names, you should package your classes within a context.

```
// To reference functions from JAR resources, you don't refer to the JAR name, only the context of the class.
// If the class is in the root of the JAR, no context is needed.
f(x) CALL var myFirstStringProcessorObj = new Packages.myStringProcessor();
// To avoid name collisions of identical function names, you must use a different context for the functions in each JAR
f(x) CALL var mySecondStringProcessorObj = new Packages.com.Novell.myStringProcessor();
```

The following example depicts the function expression used to instantiate an object of the myStringProcessor class residing in the com.Novell context.



Context in the Composer Project JAR

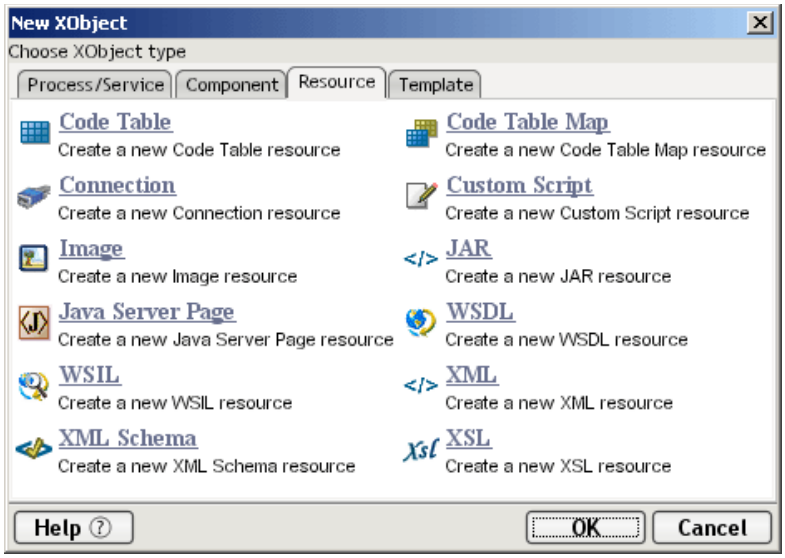
At deployment time, your JAR resources will be packaged inside a WAR file (along with your project JAR) and placed inside the deployment EAR. Each physical JAR will have a default deployment context of

/JAR/import

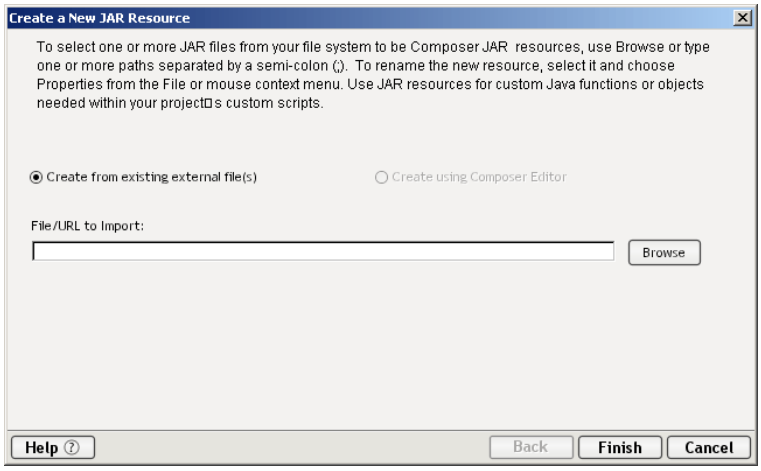
How to Create a JAR Resource

➢ **To create a JAR resource:**

1 Either right-mouse-click on the word JAR under Resources in Composer's navigation frame and choose the New command from the context menu, or from the **File** menu select **File > New > xObject**. Select the **Resources** tab from the New xObject dialog (shown below), select **JAR.**



2 In the dialog that appears, the radio button labelled **Create from existing external file(s)** is selected. See below.



3 If you know the location of the file, you can enter it directly in the text field (either as a file-system address or a fully qualified URI beginning with **http://** or **ftp://**). Otherwise, click the **Browse** button and navigate to the JAR file.

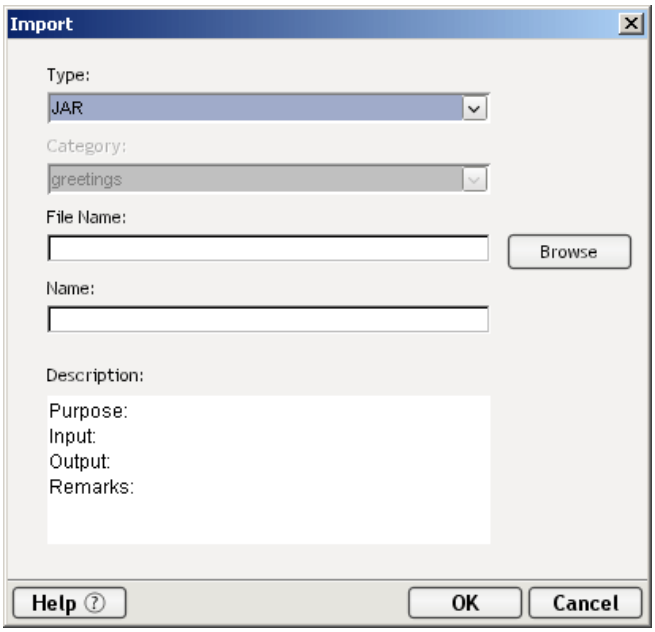
TIP: When using the file-chooser dialog (via Browse), you can Control-click or Shift-click to select multiple jars; then all will be brought into Composer at once. Each jar will retain its original name.

4 Click the **Finish** button. The dialog goes away and a new jar resource appears in the instance pane of Composer's navigator frame.

How to Import a JAR Resource

➢ **To Import a JAR resource:**

1 Right-mouse-click on the JAR resource category and choose **Import** from the context menu, alternatively you may select **File > Import xObject** from the menu. The Import dialog appears:



2 Select JAR form the Type dropdown list.

3 Use the **Browse** button to go to a file-chooser dialog. Browse your network or file system as necessary, and when you have located the JAR resource you wish to import, click Open to return to the above dialog. If you wish to load in a file from a URL, you must explicitly type "http://," "https://," or "ftp://."

4 The name of the resource is shown in the **Name** field of the dialog. Use this text field to change the resource's name if you wish to do so at this time. (You can also rename it later.)

Click **OK**. The resource is added to the instance pane of Composer's navigation frame.]

About JSP Resources

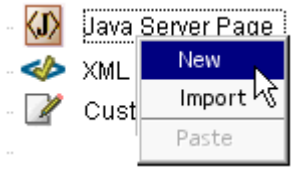
NOTE: When installed as part of the Novell exteNd Professional Edition suite, Composer does not support the JSP Resource xObject. (You can, however, create Composer-aware JSPs in Director. See the "Director JSP Wizard" discussion in the chapter on Deployment.) The following discussion applies only to users of the Enterprise Edition product.

You can create Java Server Pages directly in Composer (and then store them in your project as JSP Resources), import JSPs from a local drive or URI, or import existing JSP Resources from another Composer project. Once you create a JSP Resource, it is deployed as part of your project's deployment JAR.

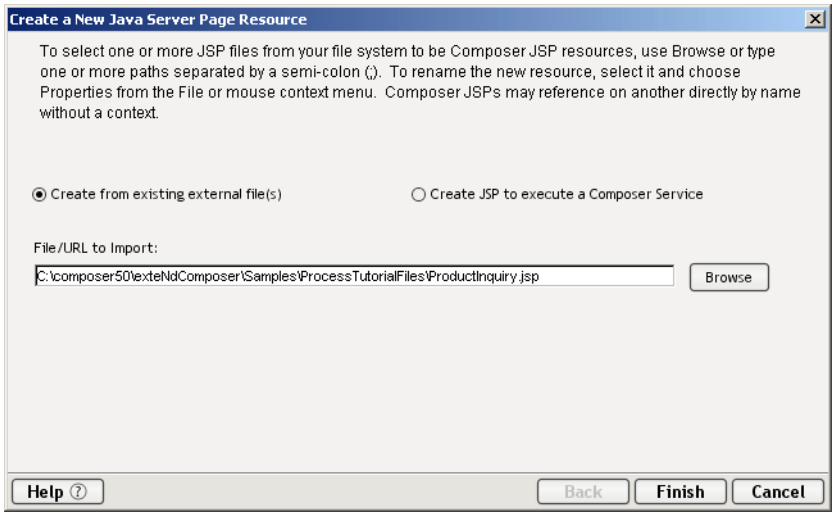
Composer's native JSP editor offers a convenient way not only to edit and create JSPs but to generate JSP-based triggers for Composer services (using Composer's custom tag libraries). This is described further below.

➢ **To create a new JSP Resource from an existing file:**

1 Either right-mouse-click on JSP under Resources in Composer's navigation frame and choose the New command from the context menu (as shown below); or go to the **File** menu and select **File > New > xObject**. From the the **Resources** tab, select **Image** and **OK**.

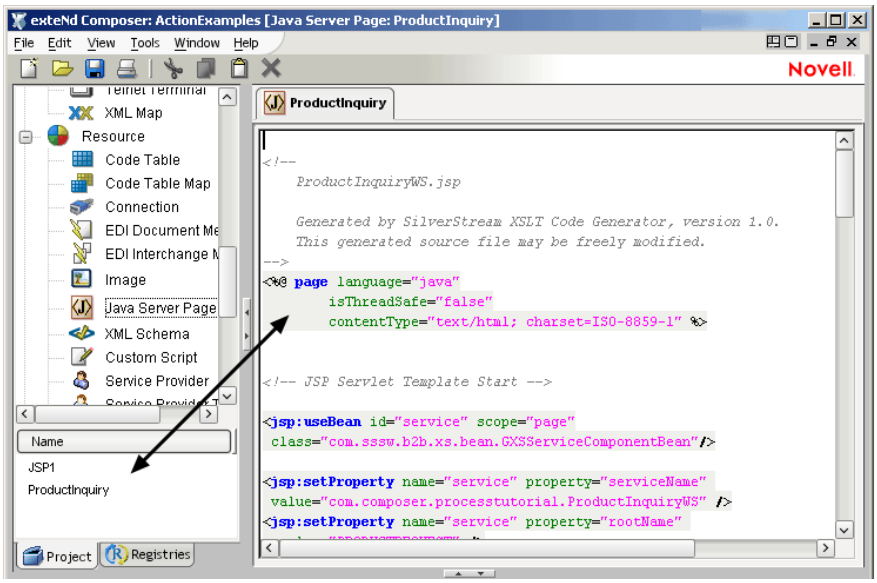


2 In the dialog that appears, choose the radio button labelled **Create from existing external file(s)**. See below.



3 If you know the location of the JSP file that you want to use in this resource, manually enter it in the text field under **File/URL to Import**. Otherwise, use the **Browse** button (and the file chooser) to navigate to the file and select it.

4 Click **Finish**. A new resource is added to the instance pane and the JSP in question opens in Composer's JSP editor as shown below.



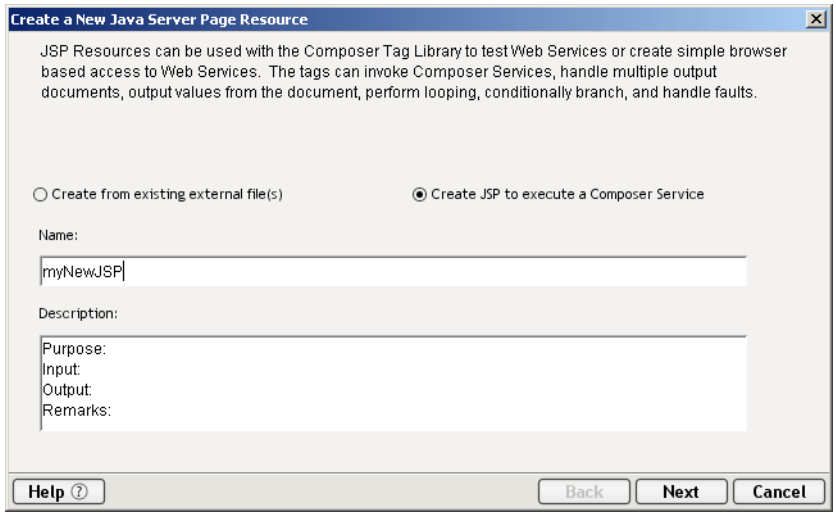
Creating a JSP-Based Service Trigger

Composer can, if you wish, automatically generate a JSP that contains code for triggering an existing service in your project. The following steps tell how.

➢ **To create a new JSP Resource containing service-trigger code:**

1 Either right-mouse-click on JSP under Resources in Composer's navigation frame and choose the New command from the context menu (as shown in the previous section); or go to the **File** menu and select **File > New > xObject**. from the **Resources** tab, select **JSP** and **OK**.

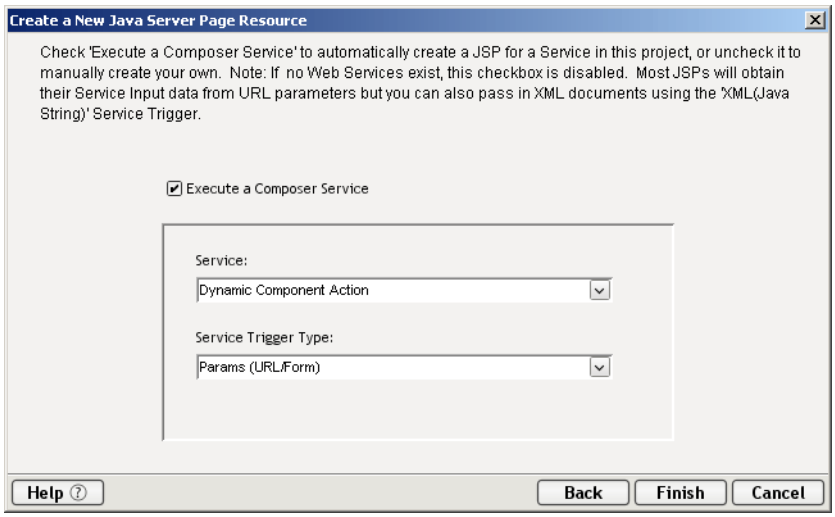
2 In the dialog that appears, choose the radio button labelled **Create JSP to execute a Composer service**. When you click this radio button, the dialog will change to have the appearance shown below.



3 Under **Name**, enter a name for this JSP.

4 (Optional) Under **Description**, enter any descriptive text that might apply to this resource.

5 Click the **Next** button. A new wizard panel appears:

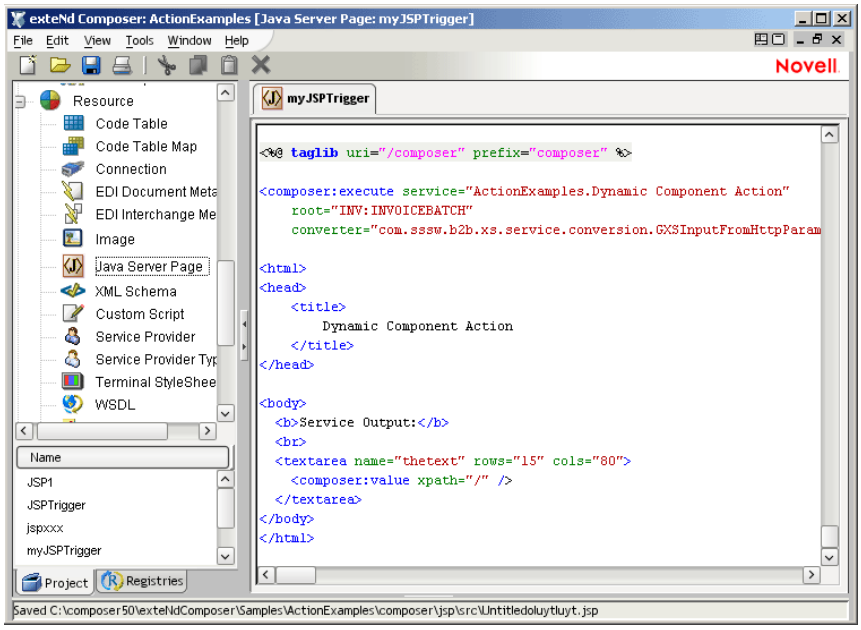


6 Check the **Execute a Composer Service** checkbox if you want Composer to generate the custom-tag code to trigger a particular service. (If you do not check this box, you will simply be creating an empty JSP. Click **Finish** now if your intent is to hand-write a new JSP.) When you check the checkbox, the controls below it become enabled.

7 Under **Service**, select a service from the dropdown menu. The menu will be pre populated with the names of services in your project.

8 Under **Service Trigger Type**, select one of the available values.

9 Click **Finish**. A new JSP containing code that executes your service will appear in the editor pane as shown below.



About WSDL Resources

WSDL (Web Services Description Language) is an XML vocabulary for describing web services. Using WSDL, it is possible to describe (in a standardized manner) the interface, protocol bindings, and various other types of information about web-based services, at a level of detail sufficient for businesses to begin to interact online. The complete standard can be seen at **http://www.w3.org/TR/wsdl**

There are three ways to create or acquire WSDL Resources.

- Use Composer's XML editor to create your WSDL by hand.
- Let Composer generate WSDL for you. (Composer can generate a WSDL file automatically for any Web Service that you have added to your project.) The procedure for this is described below.
- Acquire WSDL from a registry (such as a UDDI public registry) by downloading it directly into your project. This method will be discussed further below.

➢ **To generate a WSDL Resource from an existing service or create one in the XML editor:**

1 From Composer's **File** menu, select **New,** then **xObject**. From the **Resource** tab, select **WSDL**.

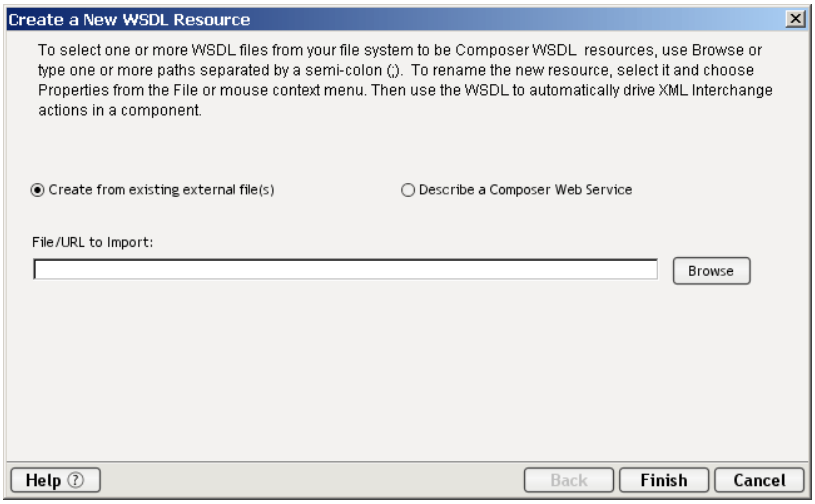
- or

Right-click on the WSDL Resource icon in the Category pane, and choose **New**. (This will associate the WSDL resource with an existing service.)

- or

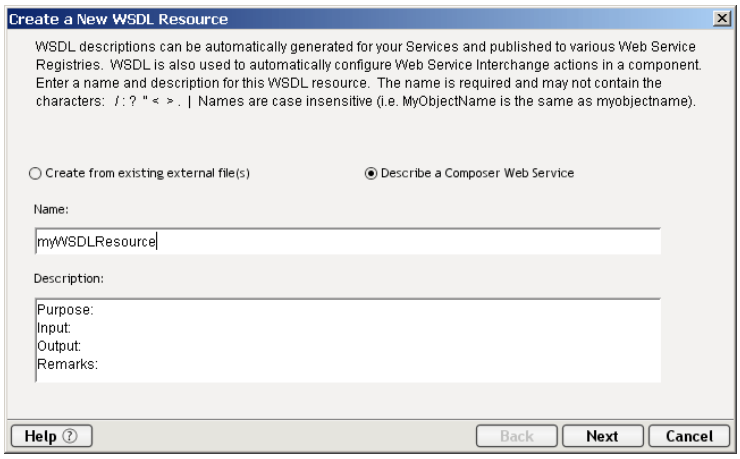
Right-click on the WSDL Resource listed in the Instance pane of the Navigator and choose **Create WSDL**. (This method will also associate the WSDL resource with an existing service.)

Any of these methods will cause the first pane of the WSDL Resource wizard to appear.



2 As indicated by the radio buttons, you have the choice to Create WSDL from an existing file, or describe a new Composer Service.

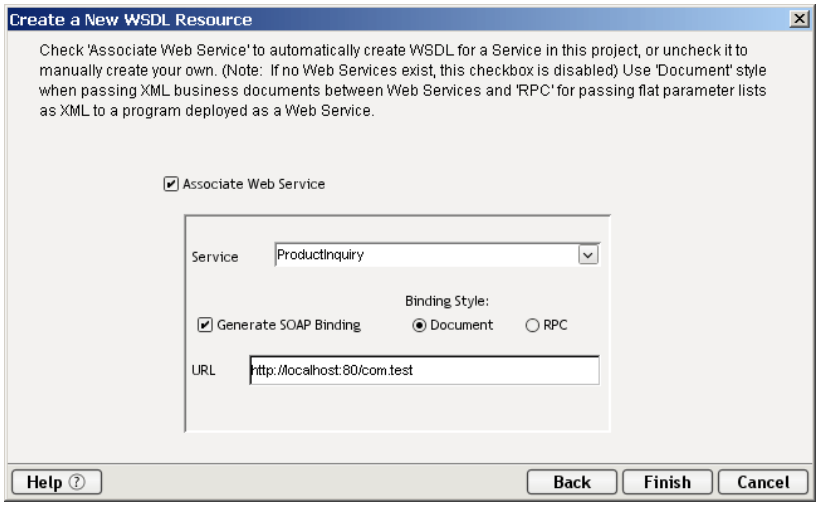
- If you choose to create the WSDL from an existing file, simply browse through your file system to locate the WSDL, and click **Finish** once you have located it.
- If you choose to describe a new Composer Service, select that radio button and follow the steps below:



3 Enter a **Name** for the resource. (This will also show up in the name attribute of the **/service** element in your WSDL.)

4 Optionally enter descriptive information.

5 Click **Next**. A new pane appears.



6 Check the **Associate Web Service** checkbox if you intend to create WSDL based on an existing Web Service in your project.

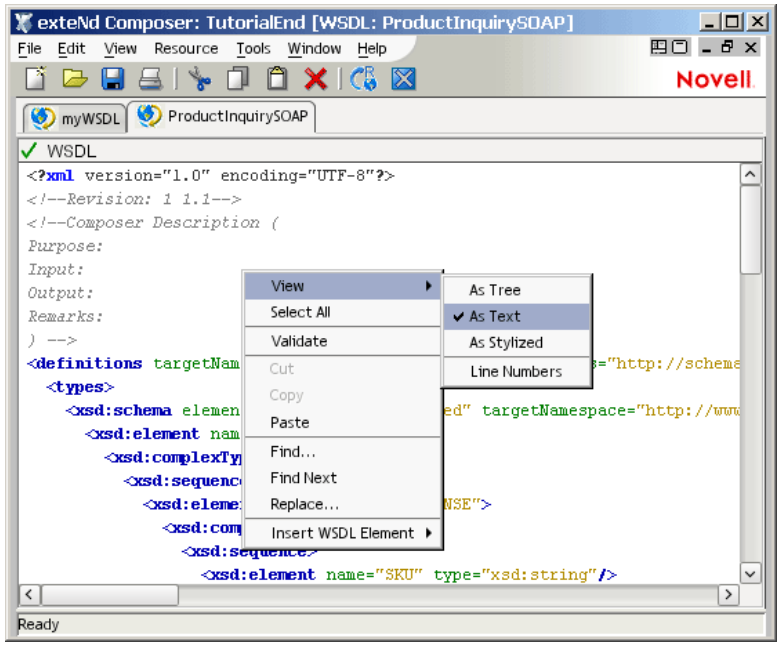
NOTE: If you wish, instead, to hand-create your own WSDL in the XML editor environment, leave the Associate Web Service checkbox unchecked and click **Finish.** After the dialog goes away, right-click in Composer's content pane and select View As Text from the context menu, then begin typing.

7 Select a **Service** from the pulldown menu.

8 Check the **Generate SOAP Binding** checkbox if you wish to have Composer automatically create SOAP Binding information in your WSDL. Choose the binding style that you want from the radio buttons labelled **Document** and **RPC**.

9 Enter the URI that you want to appear in the location attribute of the WSDL's **/service/port/address** element.

10 Click **Finish**. The newly generated WSDL appears as a DOM tree in a content window in Composer. Right-click on the DOM and choose **View > As Text** to see a text view of the WSDL document, which you can then edit manually if need be. See below.



➢ **To acquire WSDL from an external service via the Registry browser:**

1 Click the **Registries** tab in Composer's nav frame.

2 Begin a search (either of Organizations or Services) as described in "Registry Browsing" on page 327.

3 Choose a service for which detail information is available in the Service Pane.

4 Acquire the WSDL for that service as described in "Retrieving WSDL from the Registry" on page 334. The tree view for the acquired WSDL will appear automatically in the component editor's content pane. (To choose other views, right-click inside the content pane and select **View As** from the context menu.)

5 Choose **Save As** from Composer's File menu. Enter a name for the resource and click **OK**.

6 The new WSDL Resource, based on the retrieved WSDL, appears in the Instance pane of Composer's nav frame. (The WSDL is also persisted to disk at this point.)

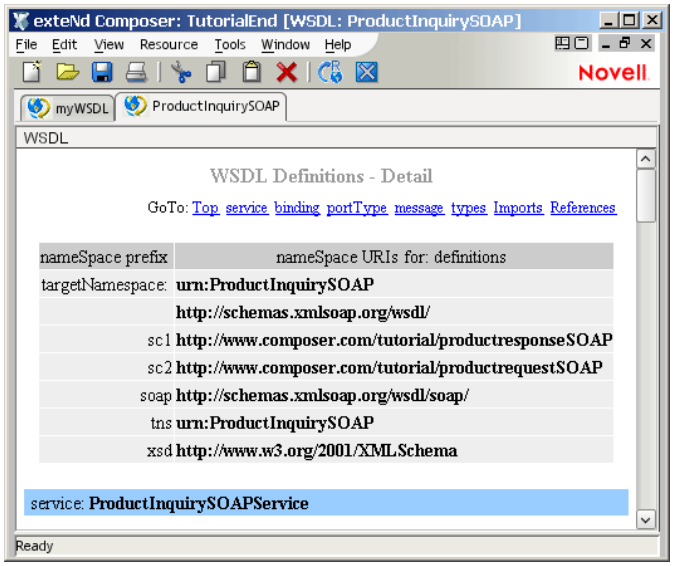
Obtaining a Stylized View of WSDL

By default, when you first open a WSDL Resource, the document is displayed in a syntax-colored text-edit view. But you can also see a stylized view of WSDL documents, created by applying an XSL stylesheet to your document.

➢ **To see a stylized view of a WSDL document:**

1 Open a WSDL Resource.

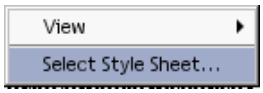
2 Right-click in the WSDL editor pane and choose **View** > **As Stylized** from the context menu. After a short delay, tview changes to a stylized view.



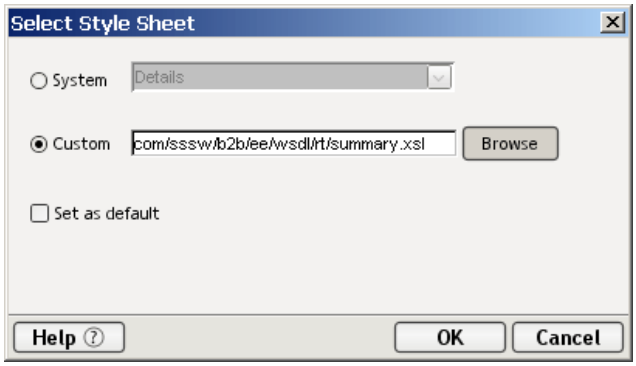
In this case, the Summary stylesheet has been applied to the document. You can apply a custom stylesheet instead, if you prefer; see procedure below.

➢ **To choose a custom stylesheet for the stylized view:**

1 *With the WSDL document already visible in Stylized form*, right-mouse-click inside the pane. A contextual menu appears. Click **Select Stylesheet.**



The following dialog will appear:



2 Choose the **System** radio button if you wish to select one of the existing standard stylesheets (Details or Summary) as the basis for the stylized view.

- **Details** provides a detail-oriented plain-text view of the WSDL document (with no XML tags).
- **Summary** provides a more concise view of WSDL contents.

3 Otherwise, choose the **Custom** radio button and enter the path to the stylesheet of your choice (or use the Browse button to bring up a standard file navigation dialog). If your path is in the form of a URL, you must explicitly type “http://,” “https://,” or “ftp://.”

4 Check the **Set as default** checkbox if you want to apply the stylesheet you’ve chosen as the default in Stylized views. Your preference is now set across Composer sessions.

Adding Elements to a WSDL Document

Although Composer can automatically generate WSDL for you, there are times when you may want or need to edit (or create) WSDL elements by hand. Composer’s WSDL Editor (a WSDL-aware version of the Composer XML editor) allows normal text insertion and cut-and-paste editing, the same as any text editor. But you can also make use of special context features that are designed to let you create standard WSDL document elements quickly and easily.

WSDL documents contain (either directly, or by importation) a minimum of four standard element types: *message*, *portType*, *binding*, and *service*. These elements build upon one another with cascading cross-references, so it is advisable that when you create a WSDL file without the use of the dialogs, you create the *message* section first, followed by the *portType* section, then the *binding* section and finally the *service* section. The WSDL Editor offers dialog-based assistance in creating each of these four types.

Adding a Message Element

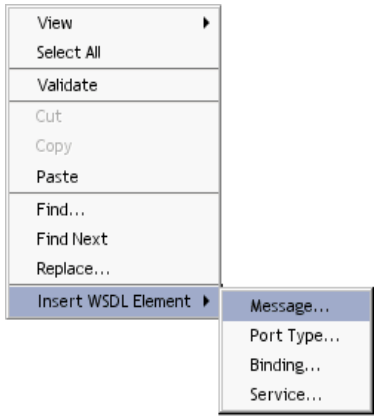
In WSDL, the Message is an abstract, typed definition of the data being exchanged. At runtime, the actual message is represented as a DOM.

➢ **To create a new Message element:**

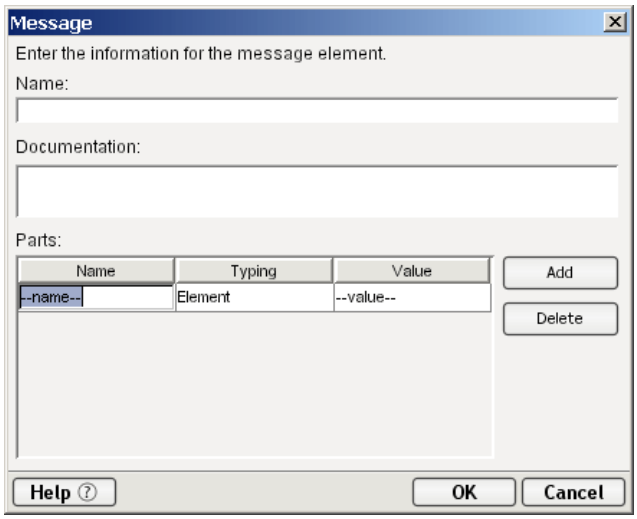
1 Open a WSDL Resource if one is not already open.

2 Be sure the WSDL document is in Text View mode. (Right-click anywhere in the document and choose **View > As Text**.)

3 Click the right mouse button inside the Text View pane. A context menu appears.



4 Select **Insert WSDL Element > Message . . .** to bring up the Insert New Message dialog.



5 Click on the **Add button** to add a blank row in the parts table.

6 In the **Name** text field, enter the value of the `name` attribute for the main `<message>` element.

7 In the **Documentation** field, enter any human-readable comment or descriptive language you would like to associate with the definition element.(Optional.)

8 Under Parts, in the **Name** column, enter the name attribute for the first `<part>` element of your message section.

9 Select a **Typing** value from the pulldown menu (Element or Type).

10 Under **Value**, enter the `element` value for this part.

11 Click the Add button to add another part entry to this message.

12 To remove an entry, first click into the entry to highlight the row in question, then click the **Remove** button to remove that entry.

13 Click **OK**. The dialog box closes and a new section is added to your document:

```
<message name="GetLastTradePriceOutput">
         <part name="body" element="xsd1:TradePriceResult"/>
</message>
```

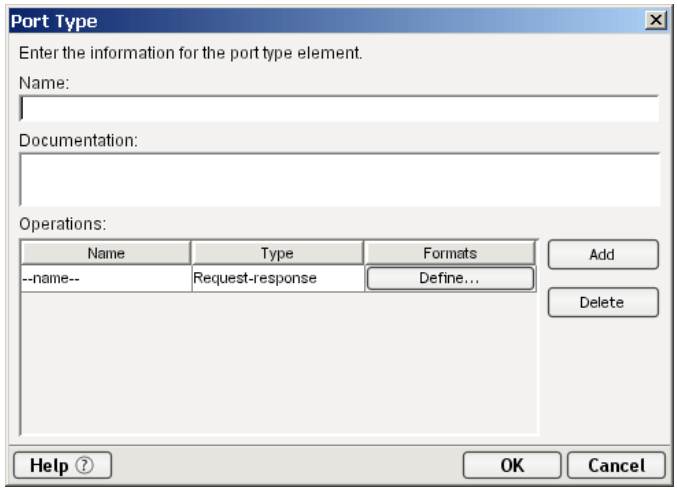
Adding a Port Type Element

The WSDL Port Type is an abstract definition of the operations supported by a service and the communications mode (one-way, request-response, etc.) that will be used in the service.

➢ **To add a new Port Type to a WSDL document:**

1 Place the mouse inside the Text View pane of the editor and click the right mouse button. A contextual menu appears.

2 Select **Insert > Port Type . . .** to bring up the Insert New Port Type dialog.



3 Click on the **Add button** to add a blank row in the parts table.

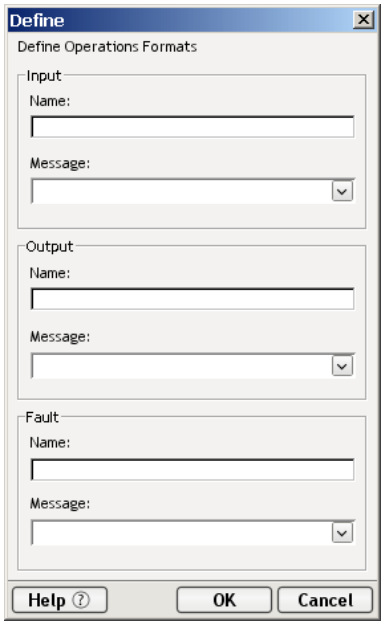
4 In the **Name** field, enter the value of the `name` attribute for the `<portType>` element you are creating.

5 In the **Documentation** field, enter any human-readable comment or descriptive language you would like to associate with the definition element.(Optional.)

6 Under Operations, enter a **Name** for this operation.

7 In the **Type** column, select One-Way, Request-Response, Solicit-Response, or Notification, as appropriate, from the pulldown menu.

8 Under **Formats**, enter an input and output message or build the appropriate messages using the Edit Operation dialog. To open the Edit Operation dialog, click the **Set. . .** button at the end of the row. A new dialog appears.



9 The Edit Operation dialog has several control groupings. Only those that are appropriate to the Operation in question (Request-Response, Solicit-Response, etc.) are enabled. For example, if you chose Notification in the Type column in Step 6 above, only the Output control group is enabled. For each enabled group, a **Name** and **Message** appropriate to the operation is required for **Input** and **Output**. However, **Fault** group is not required but optional.

10 Click **OK** to close the Edit Operation dialog.

11 Click **Add** to add more operations to the current Port Type section.

12 To remove operations, select the operation you want to remove, then click the **Delete** button.

13 Click **OK** to close the Insert New Port Type dialog. A new section is added to your WSDL document:

```
<portType name="StockQuotePortType">
    <operation name="GetTradePrice">
        <input name="input" message="tns:GetLastTradePriceInput"/>
        <output name="output" message="tns:GetLastTradePriceOutput"/>
    </operation>
</portType>
```

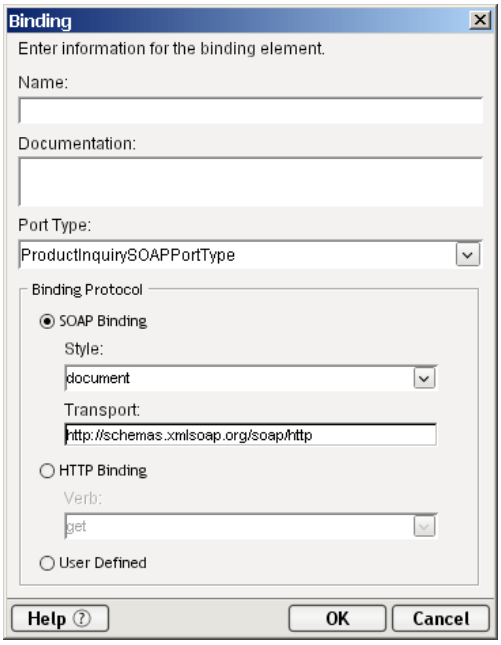
Adding a Binding Element

The Binding specifies concrete protocol and data format specifications for the operations and messages defined by a particular Port Type.

➢ **To add a new Binding to a WSDL document:**

1 Place the mouse inside the Text View pane of the editor and click the right mouse button. A contextual menu appears.

2 Select **Insert > Binding . . .** to bring up the Insert New Binding dialog.



3 In the **Name** field, enter the value of the `name` attribute for the `<binding>` element you are creating.

4 In the **Documentation** field, enter any human-readable comment or descriptive language you would like to associate with the definition element.(Optional.)

5 Select the proper Port Type for this binding, using the pulldown menu next to **Port Type**. The pulldown menu contains the names of Port Types that you have previously created (if any) for this document; see “Adding a Port Type Element” above.

6 If your WSDL document will specify a SOAP binding, check the **SOAP Binding** checkbox, then select a **Style** (RPC or Document) from the pulldown menu and enter a **Transport** value (or accept the default).

7 If an HTTP Binding will be used, check the **HTTP Binding** checkbox and enter the appropriate **Verb** (GET or POST).

8 Click **OK** to dismiss the dialog. A new Binding section is added to your WSDL document:

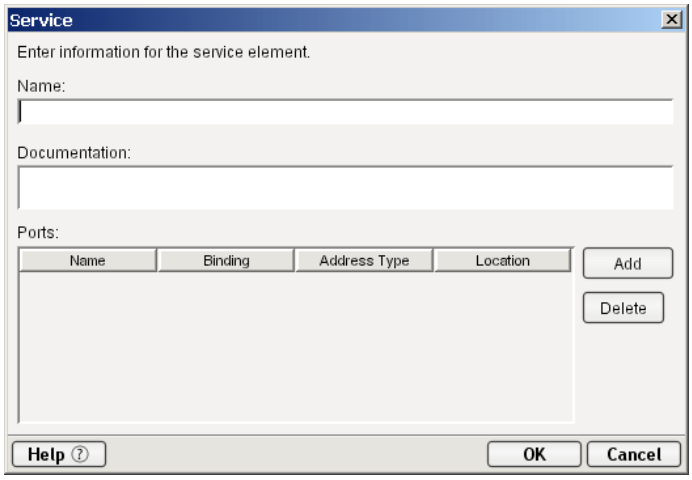
```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
        <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
        <input>
            <soap:body use="literal" namespace="http://example.com/stockquote.xsd
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </input>
        <output>
            <soap:body use="literal" namespace="http://example.com/stockquote.xsd"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </output>
    </operation>
</binding>
```

Adding a Service Element

The Service element names the entry-point address (or addresses) for the service in question. These addresses are in the form of URIs and constitute *ports*.

➢ **To add a new Service to a WSDL document:**

1. Place the mouse inside the Text View pane of the editor and click the right mouse button. A contextual menu appears.
2. Select **Insert > Service . . .** to bring up the Insert New Service dialog.
3. Click on the **Add button** to add a blank row in the service table.



4. In the **Name** field, enter the value of the `name` attribute for the `<service>` element you are creating.
5. In the **Documentation** field, enter any human-readable comment or descriptive language you would like to associate with this service element. (Optional.)
6. In the Ports section, under **Name**, enter the name of this `<port>` element.
7. In the **Binding** column, select an existing binding from the pulldown menu. The available bindings will reflect Binding sections that have already been created for this document (if any).
8. In the **Address Type** column, specify None, SOAP, or HTTP, as appropriate, using the pulldown menu.
9. Under **Location**, enter the URI via which your service will be available.
10. Click Add to add more rows (more port entries) to the Service.
11. To remove an entry, select the entry, then click the **Delete** button.
12. Click **OK** to close the dialog. A new Service entry is added to your WSDL document:

```
<service name="StockQuoteService">
    <port name="StockQuotePort" binding="tns:StockQuoteBinding">
        <soap:address location="http://example.com/stockquote"/>
    </port>
</service>
```

Type-Ahead (Code Completion) in the WSDL Editor

The WSDL editor incorporates a “smart type-ahead” feature that comes into play whenever you type a less-than sign (i.e., the start of an element tag). A contextual menu will pop up automatically, displaying available tag-name choices based on the schema specified for the file and where you are in the document.

For example, if you are creating a WSDL document manually and you are near the top of the document, typing ‘<‘ will cause a menu to appear near the cursor location, with the following choices:

Notice that the element names in this menu correspond to legal tag names in WSDL. To choose a menu item, just doubleclick it.

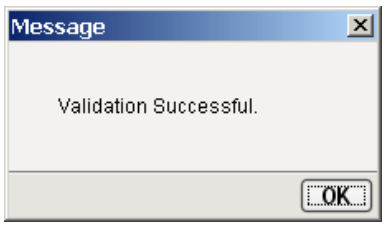
The menu choices are highly context-sensitive in that if you are deep in some portion of an element tree and you type a less-than symbol, the choices that appear in the type-ahead menu are constrained to just the values that would be legal in the XPath context in which you are typing. For example, if you are inside a `<documentation>` node anywhere in a WSDL file and you type '<', the type-ahead menu will appear with only one choice, namely `</documentation>`, because the only legal tag you could create at this point would be a closing tag. (The WSDL schema does not permit child elements inside documentation elements.)

You can, of course, always ignore the type-ahead menu altogether and enter whatever you want, as the occasion requires. For example, you might want to enter a comment.

NOTE: Type-ahead hints are based on the schema that applies to the document. Obviously, if the document does not specify namespaces or schemas, there is no way for the editor to "know" what the valid tag choices are, and you'll get no type-ahead menu.

Validating a WSDL document

When a WSDL document is open and its contents are visible in the editor, you can validate it by changing the View to "As Text" and clicking the small green check-mark icon in the top left corner of the WSDL document window. If the document validates successfully, you will see a dialog:



Otherwise, you will see an error alert giving information identifying the malformed statement(s) in the document.

NOTE: You should carefully review your WSDL even if the document validation is successful. The W3C WSDL specification does allow for extensibility elements throughout all levels of a WSDL document. So if you built the document without using the dialogs, or did an extensive amount of cut and paste from other sources, it's possible the document will test as valid, but not necessarily be what you want.

About WSIL Resources

WSIL (Web Services Inspection Language) is a specification for the discovery and publishing of Web services. It was designed to be more lightweight and portable than the previous standard, UDDI (Universal Description, Discovery and Integration), and in a sense, to pick up where UDDI leaves off. Although WSIL has yet to be submitted to one of the standards bodies (W3C and OASIS) it is gaining in popularity. To read the WSIL specification, see **http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html**.

Like WSDL, WSIL is an XML vocabulary. Its focus, however, is on exposing services rather than describing them. It is meant to facilitate *discoverability* of Web Services.

There are two ways to generate WSIL Resources. One way is to acquire WSIL from an external, existing file. A second way is to create a WSIL document using Composer's WSIL wizard and XML editor. The wizard will generate a stub file containing empty `<service>` and `<description>` elements which you can then fill in.

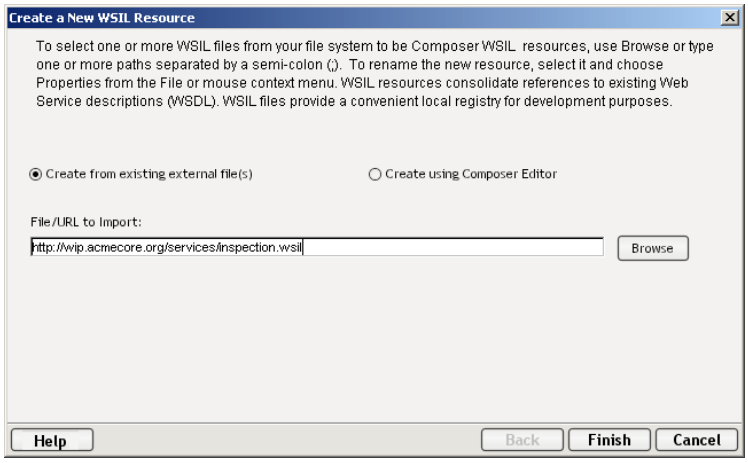
➢ **To generate a WSIL Resource from an existing file:**

1 From Composer's **File** menu, select **New,** then **xObject**. From the **Resource** tab, select **WSIL**.

- or

Right-click on the WSIL Resource icon in the Category pane, and choose **New**.

Either of these methods will cause the first pane of the WSIL Resource wizard to appear.



2 If you choose to create your WSIL from existing external files, type in the fully qualified URL or click on Browse to locate a file on your local hard drive or network.

3 Click **Finish** to open the WSIL file in the XML Content editor (see below).

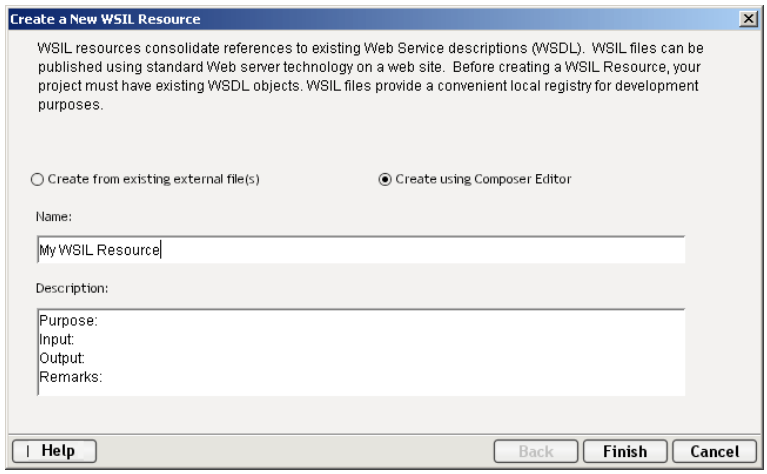
➢ **To generate a WSIL Resource manually:**

1 From Composer's **File** menu, select **New,** then **xObject**. From the **Resource** tab, select **WSIL**.

- or

Right-click on the WSIL Resource icon in the Category pane, and choose **New**.

2 Select **Create Using Composer Editor.**

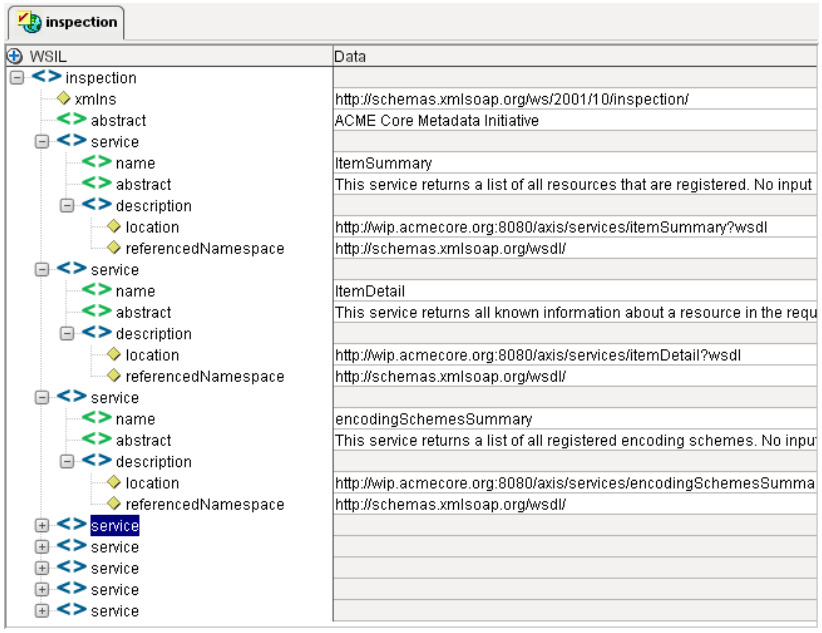


3 Enter a **Name** for the resource.

4 Optionally enter descriptive information.

5 Click **Finish**. Begin entering your WSIL in the Content Editor Screen.

The XML Content Editor Pane, with an open WSIL resource, is depicted below:



As with a WSDL document, if you right-click on the content editor and choose **View > As Text**, you will see a text view of the WSIL document, which you can then edit manually, including the node names. Similarly, you can select **View>As Stylized** for a Stylized view of the WSIL. Type-ahead code-completion and text validation (described above) also apply to editing WSIL documents.

About XML Resources

Composer allows you to specify individual XML files as first-class *resources* (xObjects). When you specify a file as an XML resource, a copy of the file is made in a folder called **\xml_resource** under your project hierarchy. This file then gets included in the deployment JAR (in that context) so that, for example, your Java Server Pages can refer to the file with a relative URL in an `href` attribute. More commonly, you'll access XML resource documents in a component's action model by using the Composer Resource action (see previous chapter, and further discussion below) to bring the document into an `Input` or `Temp` message part (DOM).

How Do XML Templates and XML Resources Differ?

XML Resources are different from XML Template documents. An XML Template is merely a design-time aid (a hint, if you will) that allows you to work with a "scratch copy" of a particular type of document (which, in turn, may or may not be based on a schema) at design time. *Instance data in the template doc is visible at design time but not at runtime.* At runtime, the template doc is never used as a data source.

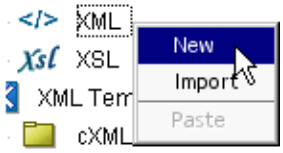
An XML Resource, by contrast, is a physical document that can be used as a static data store for "pre-canned" instance data of various kinds. Such data might consist of legal notices (copyirghts, disclaimers, warranties, etc.); names and addresses of key people who may need to be notified during the execution of a component; lookup-table data with hierarchical structure (or data that's too complex to use in a Code Table resource, but too straightforward to warrant the connectivity overhead of RDBMS storage); or common data needed by more than one service in the project.

If the dataset in question is of reasonable size, you may be able to realize significant performance benefits by using an XML Resource instead of a relational database for data lookups.

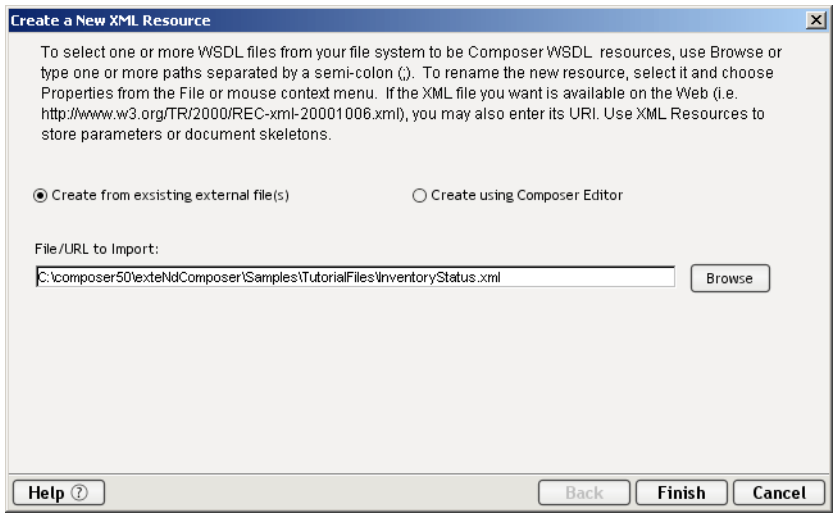
Think of an XML Resource is as a lightweight structured data store—a low-overhead container for hierarchically organized static (read-only) data.

➢ **To create an XMLResource:**

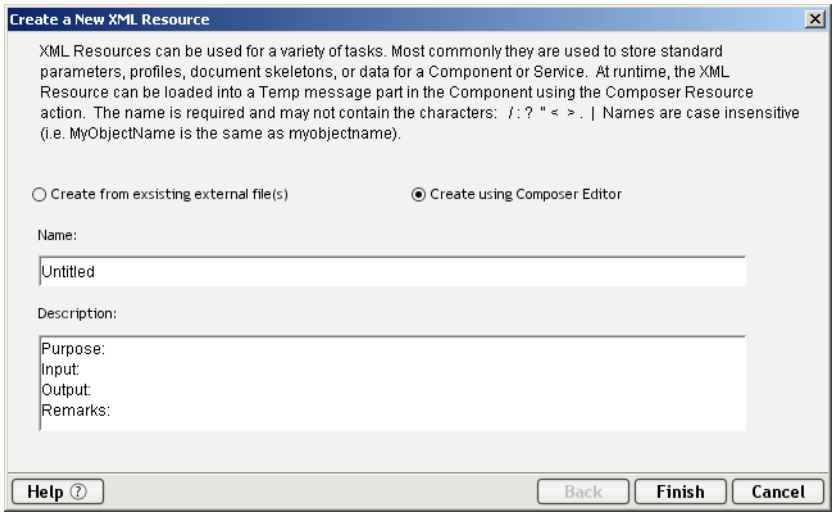
1 Either right-mouse-click on the XML category under Resources in Composer's navigation frame, then choose the **New** command from the context menu (as shown below); or go to the **File** menu and select **File > New > xObject,** then the **Resources** tab, then **XML** and **OK**.



2 In the dialog that appears (see below), choose one of the two available radio buttons as described below.



- Choose the **Create from existing external file(s)** radio button if you wish to use a preexisting XML file. (A copy of the file will be brought into your project.) Then specify the file's URI in the text field provided, explicitly typing "http://," "https://," or "ftp://," *or* use the **Browse** button to navigate to the file of interest.
- Alternatively, choose the **Create using Composer Editor** radio button if you want to create the XML file yourself, by hand. If you select this radio button, the dialog changes appearance:



At this point, you can enter a **Name** (and optionally, descriptive information) for the resource.

3 Click **Finish** to exit the dialog. A new XML Resource appears in the instance pane of Composer's nav frame, and the file itself opens in tree view in the editor pane.

How to Import an XML Resource

Unlike non-XML resource types (such as Image, JSP, and JAR), XML Resources are *not* indirected through xObject metadata stored in a separate file. Therefore, when you *import* an XML Resource, you are not restricted to importing xObjects from other Composer projects. Instead, you are actually importing an XML file directly (into a folder called **\xml_resource** under your project-folder hierarchy). That is to say, the resource and the underlying data file are one and the same. So although the steps below are slightly different from those in the previous section, they essentially give the same result.

➢ **To Import an XML Resource:**

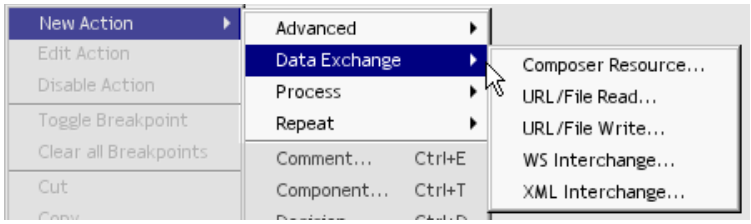
1. Right-click on XML under Resources (as described above) and choose the **Import** menu command.
2. The Import dialog appears. (See the discussion, and screen shot, at "To Import an Image resource from another project:" earlier in this chapter.) Enter the file name or URI of the XML file you want to import, *or* use the **Browse** button to navigate to an XML file.
3. Click **Finish**. The newly imported XML file will be added to the instance pane of the navigation frame, but the file itself will not automatically open in the editor pane. (If you wish to open it for editing, you can either doubleclick the file name in the XML resource instance pane, or right-mouse-click on it and choose Open from the context menu.)

How to Access an XML Resource in a Component

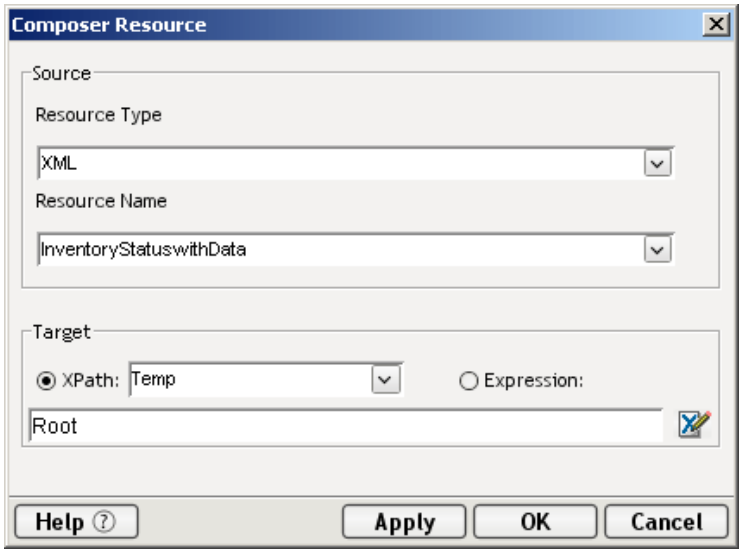
To load an XML Resource into a Part (DOM) at runtime, use the Composer Resource action type. The document and its data will be accessible via XPath or ECMAScript like any other document. You can map its nodes to other Parts, selectively pull data from certain elements, or even map the entire document to Output. Of course, you should bear in mind that an XML Resource document is a static resource (i.e., read-only). If you try to modify it or write to it using XPath or ECMAScript, it may appear as though you are changing the document—and you are—but the changes will last only for the life of the component instance in which changes are made. In other words, changes are volatile and do not get saved or carried over to future invocations of the component/service.

➢ **To load an XML Resource document into a Part:**

1. In the action model, right-mouse-click at the point where you want to load the XML resource. In the context menu that appears, select **New Action > Data Exchange > Composer Resource**. See below.



2. In the dialog that appears, under **Resource Type**, use the pulldown menu to select XML. (See below.)



3 Under **Resource Name**, select the (preexisting) XML Resource that you wish to bring into your component. (The pulldown menu will be pre populated with the names of all XML Resources that exist in the current project.)

4 Under Target, select either the **XPath** or the **Expression** radio button. Assign a target location for the XML Resource DOM.

NOTE: You can assign the contents of the XML resource doc to any node of any existing DOM. If you want to assign it to a Temp Part, you will need to create the Temp Part in advance, or else go to **File > Properties > Messages** to add a Temp Part to the currently open component.

5 (Optional) Click **Apply** if you want to test the action now. You should see the XML resource appear in the expected location, in the specified target DOM.

6 Click OK to dismiss the dialog. A new Composer Resource action is added to your action model, and from this point on in that model, you can map to or (more likely) from the nodes of the XML resource doc.

About XSD Resources

XML Schema Definition (XSD) files are specified in their own resource type so that they can be reused by various components, services, and Composer projects, and also so they can be edited or modified over time without having to be re-imported one at a time into every project or component that uses them.

There are two ways to create an XSD Resource for use in your project.

- Generate XSD directly from a sample document using Composer's schema generator, *or*
- Designate a preexisting XSD document as an XSD Resource using the Create XSD Resource wizard

We will discuss each option in turn.

Using Composer's Schema Generator

You can tell Composer to generate a schema (XSD Resource and corresponding .xsd file) from any existing XML sample document. The procedure is as follows.

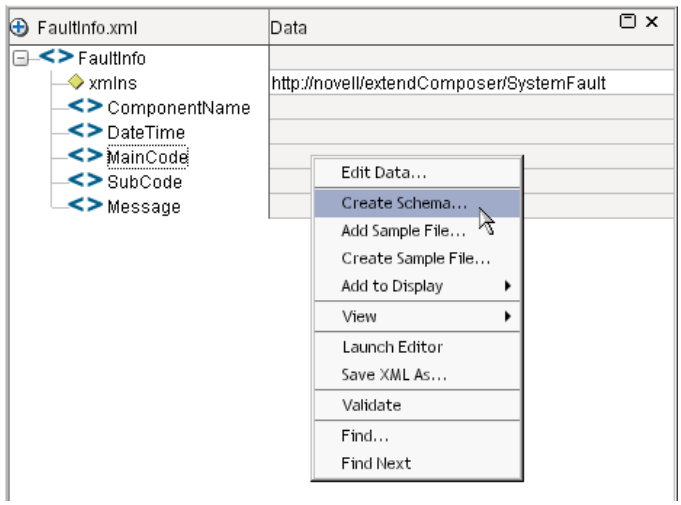
➢ **To generate a Schema (XSD) Resource from an existing XML document:**

1 Add the XML document to an existing XML Template, or create a new XML Template based on the XML sample document.

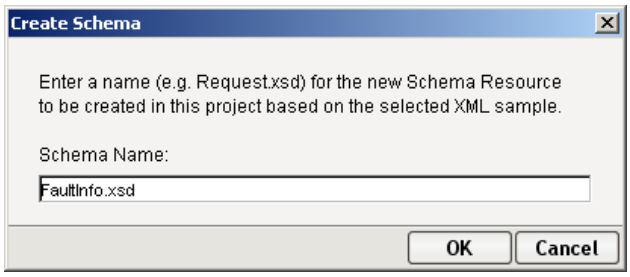
2 Open the XML Template containing your sample document. (Right-click on the template instance's name in Composer's explorer frame, and choose **Open...** from the context menu.)

3 Be sure the sample document is showing in **Tree View** in the document window. (If you were looking at it in Text View, right-click on the editor pane and choose **View > As Tree** from the context menu.)

4 **Right-click** inside the document, in Tree View, to bring up a context menu.



5 Select **Create Schema...** from the menu. A dialog appears:



6 Enter a **Name** for the new Schema Resource.

7 Click **OK**. Note that a new resource appears in the instance pane under the XSD Resource category.

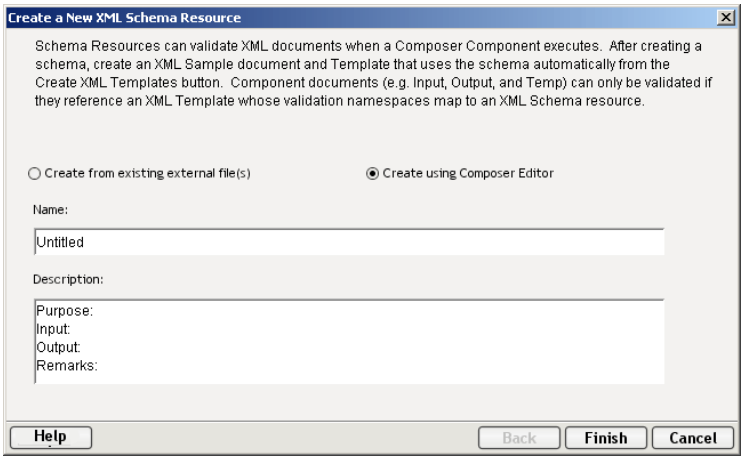
NOTE: You may need to edit your original sample document to use the namespace prefixes shown in the generated schema before the sample will validate against the schema.

Using the XSD Resource Wizard

If you wish to use an existing .xsd file as the basis of an XSD Resource, you can do so by following this procedure.

➢ **To add a Schema (XSD) Resource based on an existing .xsd file, using the resource wizard:**

1 From Composer's **File** menu, select **New**, then **xObject**, then from the **Resource** tab, select **XML Schema**. (Alternatively, right-click on the XML Schema Resource icon in the Category pane, and choose **New**.) The first pane of the XML Schema Resource wizard appears.



2 If you wish to create the Schema using an external file, check the **Create from existing external file(s)** button and type in a file or URL to import. You can also **Browse** to navigate your file system to select a file on your disk or network.

3 If you wish to create the Schema using the Composer Editor, check the **Create Using Composer Editor** button.

- Type a **Name** for the resource.
- Optionally enter descriptive information about the resource.

4 Click **Finish**. If you chose to import an existing file, the file will be opened in the Composer Component Editor. If you chose to create a schema definition file manually, you will be able to create your schema in the content window of Composer.

5 In either case, an XML Schema Resource is added to the Instance Pane.

6 Optionally right-click in the content pane and choose **View As > Text** to go to the XML editor.

About XSL Resources

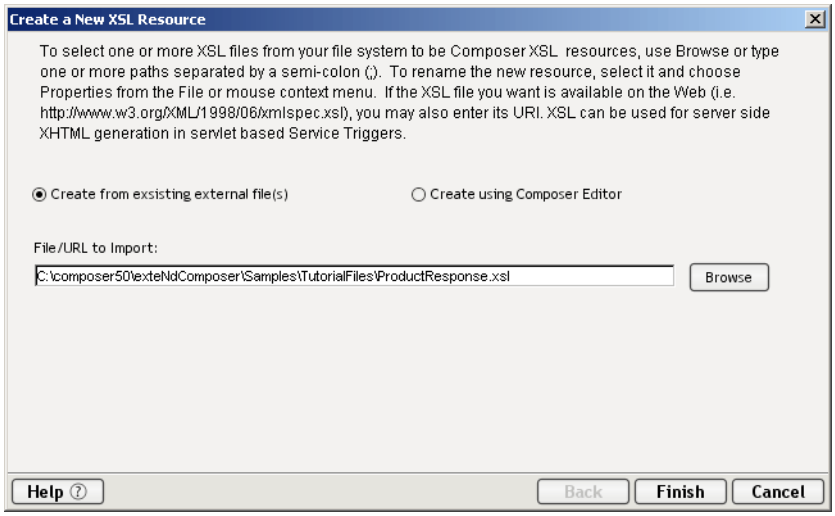
The XSL Resource offers a convenient way to package XSL stylesheets into your project's deployment JAR. You can refer to them via relative URLs from other documents, or you can load an XSL Resource into a DOM, dynamically, using the technique described at "How to Access an XML Resource in a Component" further above.

How to Create an XSL Resource

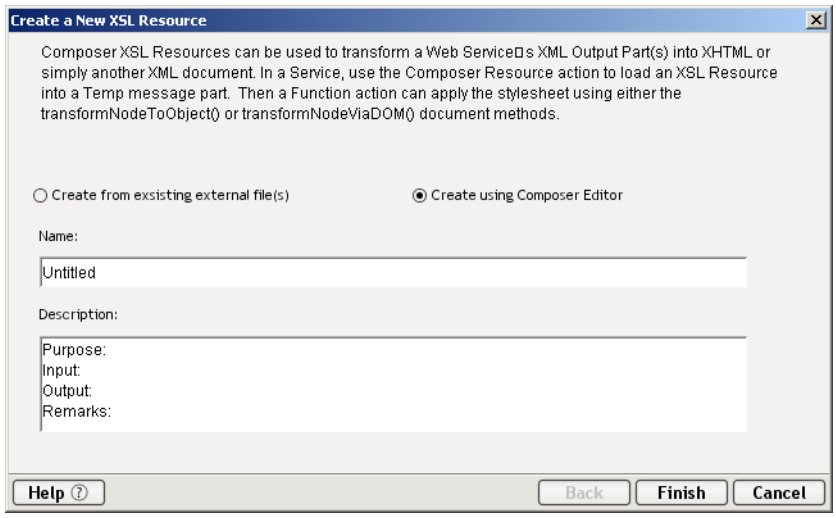
➢ **To create an XSL Resource:**

1 Either right-mouse-click on the XSL category under Resources in Composer's navigation frame, then choose the **New** command from the context menu (as shown below); or go to the **File** menu and select **File > New > xObject,** then the **Resources** tab, then **XSL** and **OK**.

2 In the dialog that appears (see below), choose one of the two radio buttons as described below.



- Choose the **Create from existing external file(s)** radio button if you wish to use a preexisting XSL file. (A copy of the file will be brought into your project.) Then specify the file's URI in the text field provided, *or* use the **Browse** button to navigate to the file of interest.
- Alternatively, choose the **Create using Composer Editor** radio button if you want to create the XSL file yourself, by hand. If you select this radio button, the dialog changes appearance:



At this point, you can enter a **Name** (and optionally, descriptive information) for the resource.

3 Click **Finish** to exit the dialog. A new XSL Resource appears in the instance pane of Composer's nav frame, and the file itself opens in text view in the editor pane.

How to Import an XSL Resource

Unlike non-XML resource types (such as Image, JSP, and JAR), XSL Resources are *not* indirected through xObject metadata stored in a separate file. Therefore, when you *import* an XSL Resource, you are not restricted to importing xObjects from other Composer projects. Instead, you are actually importing an XSL file directly (into a folder called **\xsl** under your project-folder hierarchy). That is to say, the resource and the underlying data file are one and the same. So although the steps below are slightly different from those in the previous section, they essentially give the same result.

➢ **To Import an XSL Resource:**

1 Right-click on XSL under Resources (as described above) and choose the **Import** menu command.

2 The Import dialog appears. (See the discussion, and screen shot, at "To Import an Image resource from another project:" earlier in this chapter.) Enter the file name or URI of the XSL file you want to import, or use the **Browse** button to navigate to an XSL file.

3 Click **Finish**. The newly imported XSL file will be added to the instance pane of the navigation frame, but the file itself will not automatically open in the editor pane. (If you wish to open it for editing, you can either doubleclick the file name in the XSL resource instance pane, or right-mouse-click on it and choose Open from the context menu.)

10 Custom Scripting and XPath Logic in exteNd Composer

Novell exteNd Composer incorporates an onboard EMCAScript interpreter, which allows you to extend the functionality of Composer applications in various ways. For example, you can use scripting to:

- Manipulate XML data directly, via DOM Level 2 methods
- Execute Composer components programmatically
- Call Java directly
- Perform file I/O operations
- Augment your Action Model with custom processing logic
- Develop your own utility libraries for performing common data-manipulation tasks
- Bind data connections dynamically at runtime
- Use `alert( )` functions in debugging
- Quickly prototype and test concepts that might ultimately be implemented in Java

The XPath language also offers opportunities to exploit custom logic in your Composer components. The XPath specification includes over two dozen predefined functions that can be used to filter, qualify, aggregate, and/or locate XML data.

This chapter discusses some of the techniques and capabilities applicable to the use of custom ECMAScript and/or XPath logic in Composer and describes the relationship of various W3C standards to XPath and ECMAScript.

What is ECMAScript?

ECMAScript is a lightweight, object-oriented scripting language for extending the functionality of diverse host environments (such as web browsers, editors, and IDEs) by enabling the use of custom logic. It is designed to complement or extend existing functionality in a host program such as exteNd Composer. In the web-browser world, ECMAScript is often called JavaScript or JScript.

ECMAScript is especially appropriate for a Java host environment, since:

- It is an object-oriented language with a distinctly Java-like syntax.
- Scripts written in ECMAScript can call Java constructors and methods directly.

The extensibility of ECMAScript, its powerful string-handling tools (including *regular expressions*), its DOM binding, and its ability to provide a bridge to Java, make it an ideal language to augment the programming constructs and standards used by exteNd Composer.

NOTE: You can find detailed information regarding ECMAScript at the European Computer Manufacturers Association (ECMA) web site: **http://www.ecma.ch/**

What Capabilities Does ECMAScript Offer?

In addition to letting you incorporate finely tuned custom logic into your Action Model, scripting gives you a great deal of flexibility in manipulating data, because of the various DOM- and XPath-related objects and methods available in Composer's ECMAScript extensions. Also, as an *extensible* language, custom user-defined objects can be created on-the-fly in ECMAScript and used in your Composer components and services.

The usefulness of ECMAScript is especially apparent when dealing with in-memory DOMs. Composer constructs XML documents as in-memory objects according to the W3C DOM Level 2 specification. The DOM-2 specification, in turn, defines an ECMAScript binding (see **http://www.w3.org/TR/DOM-Level-2-Core/ecma-script-binding.html**), with numerous methods and properties that provide ready access to DOM-tree contents. The standard Composer DOMs—Input, Input1, Input(n), Temp, and Output—are objects recognized by ECMAScript in Composer, and any of the W3C-defined ECMAScript extensions that apply to DOMs can be accessed from Composer.

ECMAScript also provides bridges to other expression languages such as XPath. In Composer's case, this allows you to use the Novell-supplied method `XPath()` on a DOM to address various elements within its document structure.

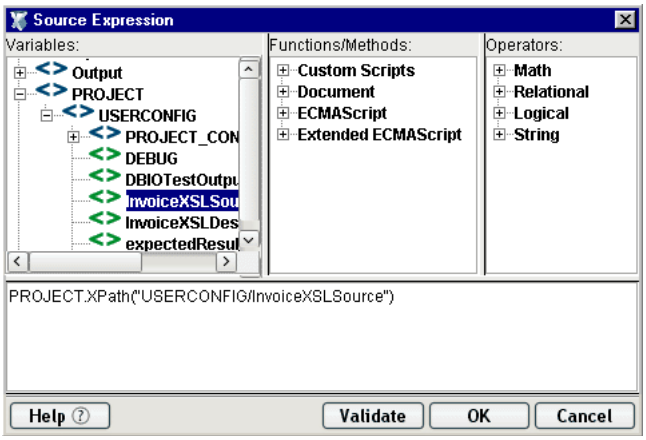
Another useful aspect of Composer's ECMAScript binding is its inclusion of *file-I/O extensions* (which are not a part of the core language). Using custom scripts, you can easily read or write scratch files, persist information to disk, or perform other common file-access tasks.

Composer's ECMAScript binding also includes database extensions that permit programmatic access to databases via JDBC. SQL statements can be passed as strings and executed against any database to which a connection can be defined.

How Scripting Is Exposed in Composer's User Interface

Composer offers access to ECMAScript in many parts of the component editor user interface, as described throughout this Guide. The most common form of access is through the Expression Builder, which can be entered whenever you see this icon:

This icon can be found in many Composer dialogs, such as the Map Action dialog, Connection Resource dialogs, etc. If you click this icon, you bring up a dialog similar to the following.



The Expression Builder dialog provides pick-lists of available objects, methods, and properties in the top panes (all of which are resizable), with rollover tool tips to help you build ECMAScript statements. Doubleclicking any item in any picktree will cause a corresponding ECMAScript statement to appear in the small edit pane in the lower portion of the window. In the example shown above, the DOM picktree corresponding to PROJECT has been opened in the Variables pane, and the node at

```
USERCONFIG/PROJECT_CONFIG/DESIGNER_EMULATION_MODE
```

has been doubleclicked. The ECMAScript expression that can access the contents of this node in the PROJECT DOM appears automatically in the edit pane.

In the window's button bar, there is a Validate button. Clicking this button will result in the ECMAScript interpreter syntax-checking your expression(s) in real time. If there are problems involving ECMAScript syntax, you will see an error dialog immediately. You can then edit the expression(s) and revalidate as needed. (Validation is, however, optional.)

NOTE: The Validate process does not *execute* your expression(s). It merely checks syntax.

Expressions for Dynamic Parameter Values

Each Composer action typically requires one or more parameters needed to perform the action. Wherever possible, Composer allows you to substitute ECMAScript expressions for these parameters. You can enter a static string, or an expression, or a series of expressions separated by semicolons. Since expressions are evaluated at runtime, this enables you to defer the choice of a parameter value until execution. This kind of late binding of parameter values is essential in cases where input values simply aren't known in advance.

Example: You might choose to hard -code a static string value for a Send Mail action's Recipient parameter. But you could also use ECMAScript to construct an e-mail address from data inside an incoming XML document, creating a flexible data-driven action with the ability to provide customization based on runtime knowledge.

Most of Composer's actions accept ECMAScript expressions for parameter values. In most cases, an XPath expression is also accepted. You will usually be able to choose from two radio buttons, labelled "XPath" and "Expression." To access the ECMAScript Expression Builder, choose the Expression radio button and click the small Expression Builder icon next to the text field where the parameter value should appear.

Custom Script Libraries

ECMAScript is also integrated into Composer as a general resource called Custom Scripts. The Custom Script resource provides an editing environment for creating custom ECMAScript functions, which you can run and debug with a command-line evaluator right inside the editor. The script editor also provides access to sample XML documents (DOM trees) and has a Java class browser so that you can easily write scripts that bridge to custom Java code. You can save libraries of custom scripts as Custom Script resources and see them listed in the instance pane of Composer's navigation frame. Also, when you've assembled custom functions into a Custom Script Resource, they automatically appear in all Expression Builder dialog pick-lists.

See "About XSD Resources" on page 256 for more detailed information about Custom Script resources and the script editor.

Function Actions

Another way in which ECMAScript functionality is exposed in Composer is through the Function Action, which is one of the core actions available in all component editors. You can insert a Function Action anywhere in your action model, to initialize variables, call custom functions, etc. One of the handiest uses of the Function Action is as a debugging aid. You can call the built-in `alert()` function with any string argument (the content of a DOM node, for example) in order to inspect the contents of a parameter value before and after an action or block of actions. The `alert()` function will bring up a dialog showing the string.

NOTE: For obvious reasons, you should disable `alert()` calls prior to deployment. This is strictly a design-time method with no applicability to the server environment.

See "The Function Action" on page 135 for information on how to create and use Function actions.

ECMAScript Access from XPath

Some dialog fields require an XPath expression. But in some instances, you may find that you prefer the greater expressivity of ECMAScript over XPath, or your logic requirements may not be accommodated by XPath's relatively limited set of built-in functions. In cases of this sort, you can still use ECMAScript: Access to ECMAScript is available, in any field requiring XPath, via the `userfunc` namespace.

For example, let's say you've defined your own custom ECMAScript function called `getTotal()`:

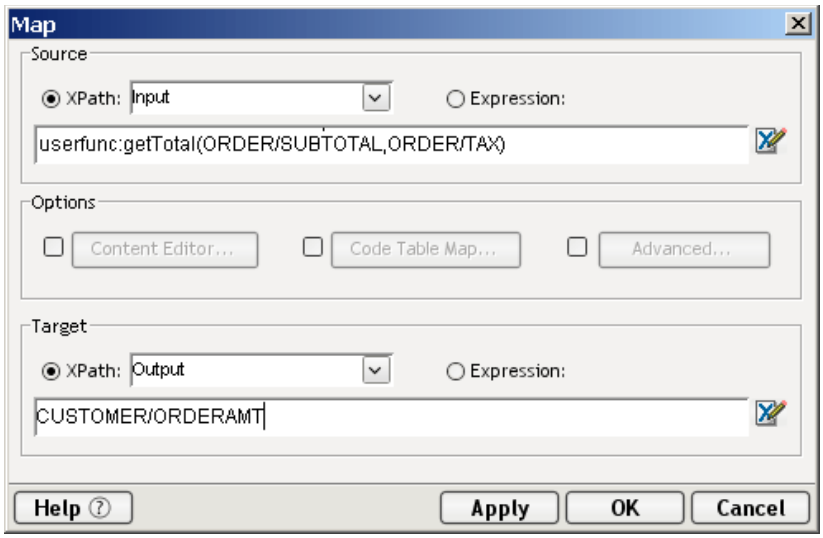
```
function getTotal(a,b) {
    return Number(a) + Number(b);
}
```

You could define this function either in a Custom Script resource in your project, or inside a Function Action.

Suppose you want to call this function from an XPath statement, passing (as arguments) values stored in two DOM nodes given by ORDER/SUBTOTAL and ORDER/TAX. Here is how you would write the XPath:

```
userfunc:getTotal(ORDER/SUBTOTAL,ORDER/TAX)
```

Here is how this call might look in an XML Map Action dialog:



XPath Access from ECMAScript

Just as you can reach ECMAScript functions from XPath, you can also obtain node objects, node data values, etc.via ECMAScript. Composer offers a variety of ECMAScript extensions for manipulating DOM elements (discussed further below). Probably the most often-used of these extensions is the `XPath()` method, which takes an XPath-style path string as the sole argument:

```
var taxNode = Input.XPath("ORDER/TAX");
var taxAmt = taxNode.toString() * 1;
```

Notice that the `XPath()` method, which is parented off a DOM root (in this case, Input), always returns a *node object*, not the node's value. To obtain the node's data value, apply the core-language ECMAScript method `toString()` to it. If the resulting string value will be used as a number, cast it to a number either by wrapping it in ECMAScript's `Number()` constructor or by multiplying by one (as shown).

NOTE: The most common error when using the `XPath()` method is to assume that it returns a data value (string, number, etc.), when in fact it returns a or *node list*. Use `item(0).toString()` to obtain the data value from the first node object in the returned node list.

Scope of Custom Script Functions and Variables

Functions stored in Custom Script resources are available to any component or service in your project, at any point in any action model. (Note, however, that after you've written a custom function, the associated Custom Script resource must be Saved before the function is available to a component.)

Global variables within Custom Script resources (that is, variables declared *outside* of custom functions) are visible only to Custom Script resource functions that use the variable(s). In other words, if you declare a variable, `myVariable`, inside a Custom Resource called **myFunctions**, only the functions within **myFunctions** will be able to see and use `myVariable`.

Variables declared within a component's action model are scoped to the component. That is, a variable declared at the top of an action model (in a Function Action) is visible to any action downstream of the declaration, and lives for the lifetime of the component, but that variable is not available to external components.

To achieve *inter-service* scope of variables, use the `putSessionValue()` and `getSessionValue()` methods described further below, in the section titled "Component (xObject)" on page 275.

Looking at an ECMAScript Example

Inside the body of any custom function, you can treat a DOM as an ECMAScript object and call valid methods on the object—such as `toString()`, which writes the DOM out to a string as text.

NOTE: In addition to custom functions, all of the standard built-in ECMAScript objects (Array, Boolean, Date, Function, Math, Object, Number, RegExp, String, and the top-level Global object), and their associated methods and properties, can be accessed from your expressions.

An example of a custom ECMAScript expression that you might use in a Function action is:

```
var onHand = Input.XPath("INVENTORYSTATUS/ONHAND");
if (Number(onHand) < 10)
   Output.XPath("PRODUCTRESPONSE/INVENTORYSTATUS") =
            "Time to reorder";
```

This script says to check the value in the Input DOM at the INVENTORYSTATUS/ONHAND element node, and if it is less than 10, map the string "Time to reorder" to the Output DOM at element PRODUCTRESPONSE/INVENTORYSTATUS.

Note that in accordance with ECMAScript syntax rules, no data-type label need be included in the declaration of the local variable `onHand`. The value retrieved in `onHand` is likely to be a *string*, however. To cast it to a number, we apply the core ECMAScript `Number()` function to it. This permits us to use the less-than operator inside the conditional.

It's entirely possible, of course, that onHand might end up being assigned a value (such as an empty string) that cannot be cast to a number, in which case `Number()` will return the problematic value NaN, which will then cause our conditional to generate an exception. In order to handle this possibility without generating the exception, one could do:

```
if ( !isNaN(Number(onHand) ) ) ?
    if (Number(onHand) < 10)
        [ code here ]
```

The `isNaN()` method is a core ECMAScript-language method which checks for "numberness."

As an alternative to the `isNaN()` tactic, one could wrap the example code in a *try/catch* statement and handle any exception in the *catch* block. (The *try/catch* construct is supported by ECMAScript.)

NOTE: For more ECMAScript examples, open (or import into your project) any of the Custom Script resources included in the sample Composer project called "Expressions."

Performance Considerations

ECMAScript is an interpreted language, which means that every line of script in an expression must be parsed and translated to the Java equivalent before it can be executed. This adds considerable overhead to the code and results in overall slower execution of scripts than pure Java. Before using ECMAScript extensively in your components and services, you should think about the possible performance ramifications.

The following guidelines will help you achieve optimal performance in your components and services:

- Whenever a logical task can be accomplished using one of Composer's built-in Action types, you should implement the task in terms of ordinary actions so that the majority of your logic runs in Java.
- When a task can't be accomplished using actions, consider whether it can be accomplished via the use of a custom Java class (which you can call from ECMAScript).
- In cases where you either can't perform a task in terms of actions or you need the fine control offered by scripting, use ECMAScript.

Bear in mind that the key to good performance is always a good implementation: choosing the correct algorithm, attention to reuse of variables, etc. Good code written in a slow language will often outperform bad code written in a fast language. Writing something in Java *does not guarantee* that it will be faster than the equivalent logic written in ECMAScript, because Java has its own overhead constraints involving, for example, constructor call-chains. (When you call a constructor for a Java object that inherits from other objects, the constructors for *all* ancestral objects are also called.)

ECMAScript's core objects (String, Array, Date, etc.) have many built-in convenience methods for data manipulation, formatting, parsing, sorting, interconversion of strings and arrays, etc. These methods are implemented in highly optimized Java code inside the interpreter. It is to your advantage to use these methods whenever possible, rather than "roll your own" data-parsing or formatting functions. For example, suppose you want to break a long string into substrings, based on the occurrence of a delimiter. You could create a loop that uses the String methods `indexOf()` and `substring()` to parse out the substrings and assign them to slots in an array. But this would be a very inefficient technique when you could simply do:

```
var myArrayOfSubstrings = bigString.split( delimiter );
```

The ECMAScript String method `split()` breaks a string into an array of substrings based on whatever delimiter value you supply. It executes in native Java and requires the interpreter to interpret only one line of script. Trying to do the same thing with a loop that iteratively calls `indexOf()` and `substring()` would involve a great deal of needless interpreter and function-call overhead, with the attendant performance hit.

Skillful use of built-in ECMAScript methods will pay worthwhile performance dividends. If you will be using scripts extensively, take time to learn about the fine points of the ECMAScript language, because this can help you eliminate performance bottlenecks.

What Is XPath?

XPath is the W3C standard that describes a syntax for addressing or locating content within an XML document. XPath also provides a lightweight *expression language* for manipulation of strings, numbers and booleans, so that users can exercise fine control over the harvesting and aggregation of XML data.

XPath models an XML document as a tree of *nodes* with parents and children. The nodes include element nodes, attribute nodes and text nodes. XPath uses an addressing scheme that resembles the directory/file path-specification conventions of some file systems, in that a slash separates parents from children. The following familiar constructs apply:

- / (forward slash) – a separator between a parent and child element in the tree
- **.** (dot) – the current location in the tree
- **..** (dot dot) – the parent location in the tree

An XPath address is often called an expression and is evaluated in reference to a *context*. A context in Composer is usually a DOM such as Input, Input1, Input(n), Temp or Output. A context in Composer can also be a Group name which itself is simply an alias or shorthand for an XPath expression.

Who Is the Target Audience for XPath?

XPath is intended to be used by all users of Composer for almost all tasks needed in processing XML documents. In some cases, as a programmer, you may find XPath insufficient in its addressing capabilities. In these cases, you may choose instead to use the more granular DOM methods, (described in “About DOMs” on page 278) for addressing an XML document. If XPath and DOM both prove inadequate then you can always choose to process an XML document directly with a Java program.

When Would I Want to Use XPath?

You can use XPath expressions whenever you want to reference an element (or attribute) or group of elements (attributes) in an XML document. In particular, you will use XPath expressions frequently in Map actions in order to specify inputs and outputs for data transfer between XML documents. You will also use XPath in Group declarations (which create a list of tree nodes matching an XPath expression) and Repeat for Element actions, which create an alias name for a repeating pattern of elements in a document.

You can also use XPath expressions in the custom ECMAScript expressions you create. Composer provides a special bridge method called `XPath()` that allows you to use XPath expressions *within* ECMAScript functions. A typical syntax is:

```
Input.XPath("ROOT/PARENT/CHILD")
```

Notice that the `XPath()` method is parented off the DOM object, which in this example is named Input. Also notice that the argument to `XPath()` is a string. (It can be either a literal, static string, *or* a string variable.)

How Is XPath Integrated into Composer?

XPath is the fundamental addressing mechanism in Composer. It is integrated directly into Composer via the dialogs for such actions as Map, Repeat for Element, and Group (plus many others). In these actions, an XPath is specified as two parts: a *context* and an *expression*. The XPath context represents the "base address," relative to which evaluation of the rest of the expression should occur. In most cases this is simply the name of a DOM (Input, Input1, Temp, Output, etc.), which represents the root in an XML document (i.e., the Document object).

The *expression* part of an XPath specifies, in top-down order, the chain of elements that leads to the node (or list of nodes) to be processed.

An XPath is created in Composer automatically by Map actions created via drag and drop. You can specify XPath expressions yourself in Map Action dialogs using the XPath Expression Builder, which provides pick-lists of valid XPath statements. You can access the XPath expression builder by pressing the Expression Builder button (shown below) whenever the XPath radio button is selected in a dialog.

 Expression Builder icon

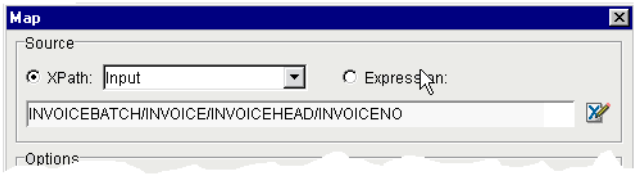
Composer integrates XPath with ECMAScript by the special method **.XPath()**. This allows you to address parts of an XML document using XPath syntax within the ECMAScript language.

Composer also provides the concept of *groups* in conjunction with XPath. When you declare a group name, it is associated with an XPath pattern that occurs multiple times in a document. This results in two special lists of nodes in the tree. The first list is a Group containing one entry for each unique node value found in the XML document based on the pattern. You can then set up a Repeat for Group loop that processes actions once for each group.

The second list is a Group(Detail) containing one entry for each member of each group (unique or not). You can then set up a Repeat for Group loop that processes actions once for each group member

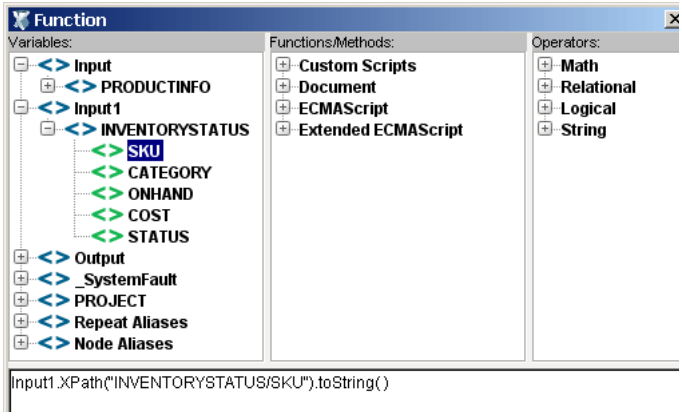
Looking at an XPath Example

XPath in the Map Action



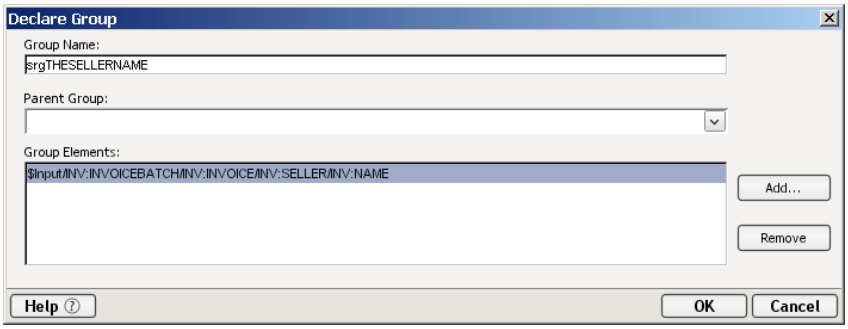
In the above example, the context is the "Input" DOM. The XPath expression is INVOICEBATCH/INVOICE/INVOICEHEAD/INVOICENO, specifying the element location of INVOICE NO as a child of INVOICEHEAD, which is a child of INVOICE, which is a child of INVOICEBATCH.

XPath in ECMAScript



In the above example, the context is the XML document object “Input1” which uses the method “.XPath( )” to specify a location of INVENTORY_STATUS/SKU and convert it to a text string (source XML). This text string object can then be manipulated using ECMAScript methods.

XPath in Groups



In the above example, the group name “srgSELLERNAME,” creates a list of nodes based on the unique data values in the XPath “$Input/INVOICEBATCH/INVOICE/SELLERNAME.” This list of unique nodes can then be processed by a Repeat for Group loop action to map data based on the unique group values instead of the individual values of each member of each group.

XPath Functions

By way of augmenting XPath’s literal-addressing capabilities, XPath’s designers built an *expression language* into the specification, to allow sophisticated filtering, introspection, and aggregation of node sets. XPath, in fact, predefines more than two dozen convenience functions (see Table) that natively recognize four data types: string, number, boolean, and node-set. The use of these functions in conjunction with ordinary XPath addressing gives the XML developer a powerful tool for manipulating XML data.

Note that all of these functions are exposed in Composer's Expression Builders, complete with rollover (tooltip) help.

XPath Functions
Node-Set Functions
number last()
number position()
number count(node-set)
node-set id(object)
string local-name(node-set)
string namespace-uri(node-set)
String Functions
string name(node-set)
string string(object)
string concat(string, string, string*)
boolean starts-with(string, string)
boolean contains(string, string)
string substring-before(string, string)
string substring-after(string, string)
string substring(string, number, number)
number string-length(string)
string translate(string, string, string)
Boolean Functions
boolean boolean(object)
boolean not(boolean)
boolean true()
boolean false()
boolean lang(string
Number Functions
number number(object)
number sum(node-set) .
number floor(number)
number ceiling(number)
number round(number)

While a detailed discussion of the use of XPath functions is beyond the scope of this Guide (see instead the complete XPath specification at **http://www.w3.org/TR/xpath**), a few quick examples will illustrate the power and elegance of the XPath expression language:

XPath Expression	Meaning
//*	The node set consisting of all nodes in the document
count(//*)	The number of nodes in the document
count(//*[contains(name(),'myNode')])	The number of nodes in the document whose name contains the (sub)string “myNode”
name(//*)	From the set of all nodes, find the name of the first node of the document, in document order. (That is, find the root node’s name.)
//*[name()='myNode']/@*	Starting with the set of all nodes, find a node whose name is “myNode” and obtain the value of the first attribute under that node, in node order.
name(//*[name()='myNode']/@*)	Obtain the *name* of the first attribute node found in the node ‘myNode’
concat(//*[name()='myNode4']/@*,' is what was found')	Combine the *value* stored in the first attribute that occurs under the element “myNode4” with the string “ is what was found”.

For more XPath examples, see the “Action Examples” project that ships with Composer.

Documentation Resources for XPath

- You can find detailed information regarding XPath at the following Web site: **http://www.w3.org/TR/xpath.**
- The W3C XML Path Language (XPath) documentation is also provided in the \Docs directory of your exteNd Composer installation.

About XSL

The following section describes writing custom scripts that use XSL .

What is XSL?

Extensible Stylesheet Language is a language for transforming XML documents into other kinds of documents.As a stylesheet language, XSL includes an XML vocabulary for specifying formatting.

Unlike HTML, element names in XML have no intrinsic presentation semantics. Without a stylesheet, an XML delivery process has no way of knowing how to render the content of an XML document other than as an undifferentiated string of characters. XSL provides a comprehensive model and a vocabulary for writing understandable stylesheets using an XML syntax.

The functionality of XSL is augmented by XSLT (XSL Transformations), which is a non-presentation-oriented transformation language for manipulating XML structure. XSLT makes use of the expression language defined by XPath for selecting elements for filtering, conditional processing, and generating string values either supplied from a source XML document or by the stylesheet author.

Who is the Target Audience for XSL?

Users who are interested in XSL are webmasters, eCommerce site builders, portal builders, and anyone else in need of a graphical representation of XML documents as part of business-to-business transactions.

Given an XML document, designers can use an XSL stylesheet to express how that structured content should be presented; in other words, how the source content should be styled, laid out, and/or paginated onto some presentation medium, such as a window in a Web browser or a hand-held device, or a set of physical pages in a catalog, report, pamphlet, or book.

When Would I want to Use XSL?

XSL is designed to permit XML delivery devices to display XML in a way that is meaningful to humans. XML data exchanges often involve user interactions—Web shopping experiences, data auditing, notifications, and other XML uses requiring a graphical display of data. In short, you would use XSL whenever you need to make XML presentation-enabled.

How is XSL Integrated into Composer?

XSL is integrated into Composer by means of the XSL Transform Action, which is available in all components. To use the action, you need to specify parameters for a source DOM, an XSL Stylesheet, and a destination DOM (e.g., Temp or Output). See the next section for an illustration.

Composer also provides special XSL methods for use in Custom Scripts or Function Actions:

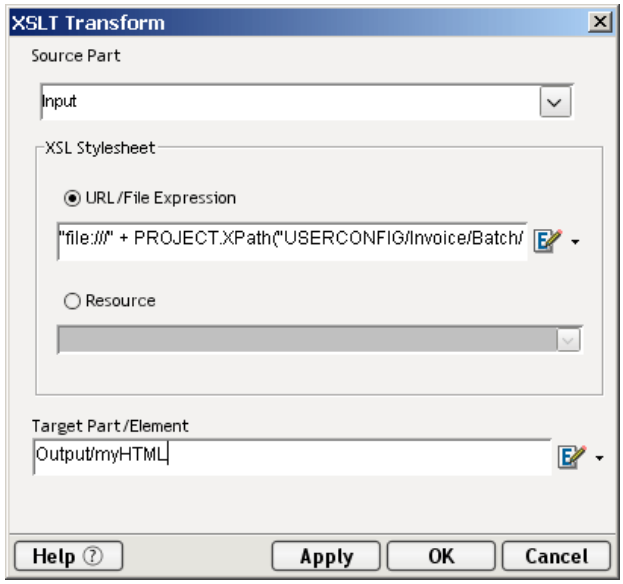
```
transformNodeViaDOM()
transformNodeToObject(,)
transformNodeViaXSLURL()
```

See the API descriptions further below for details on these methods.

Web services that you create using exteNd can also be set up to output XSL-transformed XML directly as HTML. See the Deployment chapter in the *exteNd Composer Enterprise Server Guide* for your particular app server platform for more information on deployment to HTML using Processing Instructions.

Looking at an XSL Example

The Process XSL action shown below uses the XSL stylesheet specified in the XSL URL field to transform the input Part, placing the result into an XML element called “MyHTML” in the output doc.



For additional examples of how to use XSL, be sure to see the “Action Examples” project in your Composer installation.

Resources for XSL

- You can find detailed information regarding XSL at the following WEB site: **http://www.w3.org/TR/xsl**
- XSL documentation (from W3C) is also provided in the **\Docs** directory in your Composer installation hierarchy.
- For working examples in Composer, see the “Action Examples” project in your Composer installation.

About Novell Scripting Extensions

The Novell extensions to ECMAScript consist of a set of convenience methods for general purpose scripting involving xObjects, DOMs, and other Composer objects. All of the methods are exposed in the Expression Builder pick-lists. An introduction to the API is given below.

General Purpose Extensions

The general purpose extensions are categorized by the type of objects they operate on and consist of the following:

Node

`XML`—This property returns a string representing the DOM.

`createXPath(XPathType asPattern)`—Creates the XPath pattern.

`getXML ()`—This property returns a string representing the DOM.

Document

`text`—This property returns a concatenated string of all the text nodes (content) under it.

`setDTD(node RootElementName, object PublicName, object URL)`—Sets DTD file for the document.

`setValue(Object aValue)`—Sets the Value of a Document from the passed object, if it is in another document, then this method copies child nodes (elements and attributes). If passed object is text, it is parsed to create a DOM.

`toString()`—Converts a DOM document to an XML formatted string.

`transformNodeViaDOM(XSLDOM)`—Transforms the document according to the XSLDOM and returns a string. The parameter XSLDOM is an XSL stylesheet, that may have been read into the component by an XML Interchange Action. This method could be used in the source of a Map Action, or call it in a Servlet using the Server Framework class IGXSXSLProcessor.

`transformNodeToObject(XSLDOM, OutputDOM)`—Transforms the document according to the XSLDOM and returns results to Output DOM. The parameter XSLDOM is an XSL stylesheet that may have been read into the component by an XML Interchange Action. The parameter Output DOM is the target DOM for the results. From a component, this method could be in a Function Action, or from a Custom Script, you can use it once you have all three DOMs, or call it in a Servlet, using the Server Framework class IGXSXSLProcessor.

`transformNodeViaXSLURL(XSLURLLocation)`—Transforms the document according to the XSLURLLocation and returns a string. The parameter XSLURLLocation is an XSL stylesheet. This method could be used in the Source of a Map Action, or from a Custom Script you can use it once you have a DOM, or call it in a Servlet using the Server Framework class IGXSXSLProcessor.

`validate()`—XPathTypes can be of type NodeList, String, Number, or Boolean. Usually used to return a Nodelist matching the XPath pattern. Use brackets to select a particular node from the list [e.g.,.Input.XPath("INVOICE/LINEITEM[1]") or Input.XPath("INVOICE/LINEITEM[last()]")]. Use the @ to select a node by attribute (e.g. Input.XPath("INVOICE/LINEITEM[@myattr]"). To select by attribute value...Input.XPath("INVOICE/LINEITEM[@myattr='abc']").

`Nodelist XPath(XPathType asPattern)`—XPathTypes can be of type NodeList, String, Number, or Boolean. Usually used to return a Nodelist matching the XPath pattern. Use brackets to select a particular node from the list [e.g.,Input.XPath("INVOICE/LINEITEM[1]") or Input.XPath("INVOICE/LINEITEM[last()]")]. Use the @ to select a node by attribute [e.g. Input.XPath("INVOICE/LINEITEM[@myattr]")] To select by attribute value...Input.XPath("INVOICE/LINEITEM[@myattr='abc']").

Element

`text`—This property returns the concatenated text of all the text nodes under it.

`booleanValue()`—Returns the boolean value (true | false) of this object if possible.

`countOfElement(String propertyName)`—Returns a count of the named child.

`doubleValue()`—Returns a double value for this object if possible.

`exists(String propertyName)`—Check for the existence of the named child.

`getIndex()`—Returns back the current index.

`getParent()`—Returns the parent element.

`setIndex(int aiIndex)`—Sets the iterator index value for this element.

`setText(String asText)`—Sets the text node associated with this element.

`setValue(Object aValue)`—Sets the Value of an Element from the passed object, if it is another element then this method copies child nodes also (elements and attributes).

`toNumber()`—Gets the text node and converts it to a number.

`toString()`—Gets the text node associated with this element.

`Nodelist XPath(XPathType asPattern)`—XPathTypes can be of type NodeList, String, Number, or Boolean. Usually used to return a Nodelist matching the XPath pattern. Use brackets to select a particular node from the list, e.g., Input.XPath("INVOICE/LINEITEM[1]") or\nInput.XPath("INVOICE/LINEITEM[last()]"). Use the @ to select a node by attribute, e.g., Input.XPath("INVOICE/LINEITEM[@myattr]") To select by attribute value...\nInput.XPath("INVOICE/LINEITEM[@myattr='abc']").

Attribute

`text`—This property returns the text value of the attribute.

`setValue(Object aValue)`—Sets the Value of an Attribute from the passed object.

`toString()`—Gets the text node associated with this attribute.

Nodelist

`avg(NodeList)`—Returns a number equal to the average value in the NodeList. The NodeList parameter of type XPath. If no parameter is supplied, then the current NodeList/GroupName is used.

`count([NodeList])`—Returns a number equal to a count of the nodes in the NodeList. The optional NodeList parameter is of type XPath. If no parameter is supplied (the usual case), then the current NodeList/GroupName is used.

`min([NodeList])`—Returns a number equal to the lowest value in the NodeList. The NodeList parameter of type XPath. If no parameter is supplied, then the current NodeList/GroupName is used.

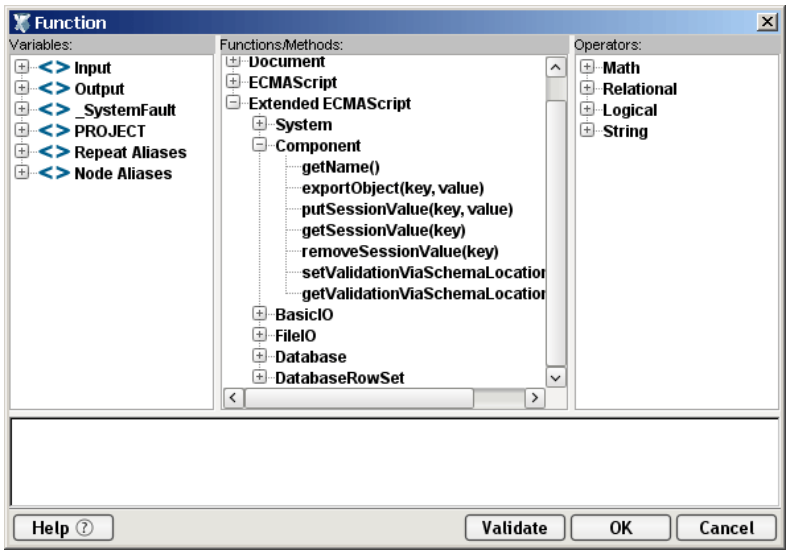
`max([NodeList])`—Returns a number equal to the highest value in the NodeList. The NodeList parameter of type XPath. If no parameter is supplied, then the current NodeList/GroupName is used.

`sum([NodeList])`—Returns a number equal to the sum of the values in NodeList. The NodeList parameter of type XPath. If no parameter is supplied, then the current NodeList/GroupName is used.

`where(XPathType asPattern)`–Gets a NodeList of nodes matching the XPath pattern.

Component (xObject)

An object called `theComponent` is exposed within each Composer component via the Expression Builder. (You can open the Expression Builder window by clicking the Expression icon in any Function action, Map action, or other dialog in which the icon appears.) The component-based methods are exposed in the pick-list under Extended ECMAScript/Component, as shown below.



The object called `Component` has the following methods:

`getName()`—Returns the name of the currently executing component. To obtain the name of the currently executing component, you would call:

```
Component.getName()
```

`exportObject(key,value)`—Allows you to store a reference to any ECMAScript variable or Java object in a hash table so that other components within a given service can look up the object and use it. (Otherwise, user variables are scoped to the component in which they are declared and cannot be seen by other components.) For example, suppose you have created a variable, `testString`, and you wish to make it available to other components in the same service:

```
// create an instance of the string:
testString = 'hello';

// now export it:
theComponent.exportObject("myExport", testString)
```

Note that the hash key “myExport” is simply any arbitrary name. Other components will need to use this name to look up the exported object (`testString`). Within another component, you can do:

var copyOfString = theComponent.getExportValue("myExport");

The component that executes this code will then have access to the string `'hello'` that was in the variable `testString` within the other component.

It is important to understand that variables or objects exported in this fashion are scoped to the *service* instance in which they are created. This means:

- When the service ends, the exported objects go out of scope.
- Only components that run within the given service can “see” exported variables.

NOTE: To achieve *inter-service* scope of session variables, use the `putSessionValue()` and `getSessionValue()` methods described further below.

It is also important to understand that at design time, exported variables will not be in scope (will not be usable) unless the service or component that created them is itself running. Suppose Service A creates a variable `myVar` and exports it as `'myExportedVariable'`. Service A calls (executes) Component B. Inside Component B is a Function Action that looks up the exported variable:

```
theVar = theComponent.getExportValue('myExportedVariable’)
```

In order for `theVar` to contain the correct value, Service A must already be running and it must already have exported `myVar`. In other words, if you merely animate Component B without running Service A, you'll encounter a problem since `myVar` is not in scope. The correct thing to do is to begin your animation from Service A. Step through Service A until you reach the Component Action that executes Component B. Use the Step Into button to step into the action model for Component B. Then step through Component B. This way, both A and B are in scope at the same time and any exported variables will be usable.

`getExportValue(key)`—Allows you to access a reference to any ECMAScript variable or Java object that was previously exported by another component. (See discussion above.)

`putSessionValue(key,value)`—Allows you to store a reference to a Java object in a global variable so that it can be referenced from any other service or component running in the same servlet session (which may span many HTTP hits). Objects published in this fashion have servlet-level scope. (Session life is dependent on the HTTP Server Session timeouts.) The first argument is a String representing the name for the published object. The first argument is a String representing the name you wish to associated with the published object. The second argument is the object. (The syntax follows the convention for `exportObject`, shown above.)

NOTE: This method will generate an exception if it is used in a Web Service that was deployed using an EJB. Also, this method cannot be used in a JMS Service.

`getSessionValue(key)`—Allows you to obtain a reference to a Java object that was previously published via the `putSessionValue()` method (above). This method will return null if no object matching the key is found; otherwise, it returns an Object.

NOTE: This method will generate an exception if it is used in a Web Service that was deployed using an EJB. Also, this method cannot be used in a JMS Service.

`removeSessionValue(key)`—Allows you to destroy a reference to a Java object that was previously published via the `putSessionValue()` method (above).

NOTE: This method will generate an exception if it is used in a Web Service that was deployed using an EJB. Also, this method cannot be used in a JMS Service.

LDAP Methods

`getLDAPAttr(String connResource, String dn, String attr)`—Looks up a value stored in a particular attribute of a named object in an LDAP directory, using the connection resource whose name is supplied in the first argument. The second argument is the object's LDAP distinguished name. The third arg is the attribute of interest. The value returned may be numeric or String data. Use ECMAScript's `typeof` operator to determine if the value is of type "number" versus type "string."

The `getLDAPAttr()` method is available for use in any component, service, or connection resource, whenever ECMAScript can be used. (In other words, its use is not limited to LDAP Components.) Other ECMAScript extension methods involving LDAP are available only within the LDAP Component editor. See the separate *LDAP Connect User's Guide* for more information on those methods.

Connector-Specific Extensions

Additional custom ECMAScript objects and methods beyond those described here are available in conjunction with most Connect products for exteNd. (For example, JMS-specific methods are available for use in components and services created using the JMS Connector.)

Consult the appropriate Connect documentation for details about connector-specific ECMAScript objects and methods.

When Would I Want to Use Novell Scripting Extensions?

Use Composer's general purpose extensions wherever you find them helpful and /or where they are more robust than similar methods in XPath, DOM or XSL.

You might want to use some of Composer's grouping or aggregation-related extensions when you want to summarize common data that repeats but is scattered about an XML file. For instance, an XML file may arrive with 50 randomly organized invoices, generated by only seven departments in your organization. Using Composer's grouping capability and group-oriented methods, you can easily organize the 50 invoices by the departments and summarize "invoice totals" across each group.

How Are Novell Scripting Extensions Integrated into Composer?

The general purpose extensions are built right into ECMAScript and appear on the Expression builder pick-lists alongside other objects, properties and methods.

Composer's Group action is used to specify Groups by simply clicking a pick-list to generate an XPath pattern that forms the basis for the group. There is a Repeat for Group Action that allows you to process a set of actions for each member of the Group or Group(Detail). The aggregate calculation methods are available in the ECMAScript Expression Builder in the Map dialog.

The types of actions described here are available in all Connect component types.

Extension Code Examples

See the "Action Examples" project in the **\Samples** directory of your exteNd Composer installation for examples of how to use ECMAScript in Composer to accomplish a wide variety of tasks.

About DOMs

What is DOM?

The Document Object Model is an interface that allows programs and scripts to dynamically access and update the content, structure and style of XML documents. W3C's Document Object Model (DOM) is a standard internal representation of an XML document structure inside a software program and aims to make it easy for programmers to access elements, attributes and data and delete, add or edit their content and style.

What Does a DOM Do? What are the Key Features?

The DOM defines a set of standard methods and properties for creating and operating on XML documents programmatically as objects. It provides methods for manipulating all parts of an XML document including Elements, Attributes, Text, Processing Instructions, etc.

The DOM also provides a set of methods for addressing or locating nodes in an XML document.

Who is the Target Audience for DOM Methods?

The DOM methods and programming model is targeted at professional developers who need absolute control over DOM manipulation. Working with DOM methods gives developers control over primitive operations involved in constructing and manipulating DOMs. A simple Map action in Composer might translate to tens of lines of ECMAScript/DOM instructions.

When Would I Want to Use DOM Methods?

You may wish to use DOM methods when the basic Composer actions combined with ECMAScript functionality cannot meet your XML document processing needs.

How Are DOM Methods Integrated into Composer?

The methods and properties of the DOM for manipulating Composer DOMs are available only in the Custom Script editor or ECMAScript expression builder in Actions. You can access the ECMAScript Expression builder by pressing the expression builder button (shown below) whenever the Expression radio button is selected in a dialog.

Looking at a DOM Methods Example

- See the Custom Script function titled dbIOtoDOM() in the Database.es file in the directory..\exteNd\Samples\CustomScripts.
- See the sample project “Expressions” for a full treatment of all the DOM methods/objects.

Documentation Resources for DOMS

- You can find detailed information regarding DOM at the following WEB site: **http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/**
- The complete Document Object Model (DOM) Level 2 Specification documentation is provided in the exteNd\Docs directory

About Java Integration

Java is more than merely a programming language. It’s a computing platform designed to allow the same software to run on different kinds of devices: PCs, Unix workstations, wireless devices, handheld computers, consumer electronics, and embedded systems of various kinds. Using networks, it is possible to create distributed applications using Java that tie together diverse devices into a single working application.

In addition to being a computing platform in its own right, Java is a robust, object-oriented computer language that forms the basis of the Java 2 Enterprise Edition (J2EE) computing architecture, which is increasingly preferred by IT organizations because of its adaptability, robustness, platform neutrality, and track record of successful adoption by large companies. J2EE is also firmly connected to emerging standards in the areas of XML and Web Services, which makes Java an ideal enterprise-programming language.

How Is Java Accessible in exteNd Composer?

Java is integrated into Composer services through the ECMAScript scripting environment, which provides a direct bridge to external Java objects. Composer provides a Java class browser in the Custom Script editor with drag-and-drop functionality, enabling you to quickly integrate Java objects and use their constructors, properties, and methods within your scripts.

When Should You Use Java?

Most Composer users will be able to achieve their Web Services and XML integration objectives without needing to augment Composer's native functionality through the use of custom Java classes. Nevertheless, there are situations where it may be desirable to integrate Java objects into exteNd. For example:

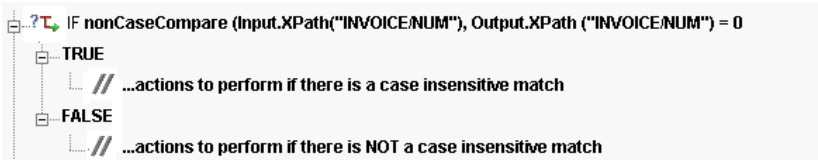
- You want to access (reuse) existing Java business objects, some of which may access data in other computing environments or programs
- You want to provide an XML interface to your own Java business objects
- You need to perform complex manipulations of XML documents that are more effectively handled by Java than with Composer actions or ECMAScript
- You've prototyped a concept using ECMAScript and now want (for reusability and/or performance reasons) to implement the same concept in Java

Looking at a Java Integration Example

A simple use of Java in an XML application may be to perform a case-insensitive comparison of data in two XML elements. In this case, you could create a Custom Script function using the Java string object as follows:

```
// Case Insensitive Compare, returns 0 if strings are equal, non-zero if not…
function nonCaseCompare(string1,string2)
{
var s1 = new Packages.java.lang.String(string1);
var s2 = new Packages.java.lang.String(string2);
return s1.compareToIgnoreCase(s2);
}
```

Then you might use the function in a Decision action to conditionally execute different actions, as the illustration shows.



Documentation Resources for Java

You can find detailed and definitive information regarding the Java platform and the Java programming language at the following Web site: **http://java.sun.com**.

11 Applying Actions to Common Tasks

Actions are the atomic units of work in all Composer components. They are responsible for the control flow and logical constructs that make custom applications possible. As you might expect, some actions are used more than others, and certain design patterns reoccur frequently in Web Service applications created with Composer. This chapter discusses some of the actions and design patterns you're most likely to encounter when using Composer.

About the Examples in this Chapter

The exteNd Composer design-side installation includes a few sample projects, one of which is called ActionExamples.spf. The project contains sample documents called InvoiceBatch*.xml. This chapter's examples are based on using the InvoiceBatch template as Input to XML Map components.

If you'd like to follow along as you read the examples, you can open ActionExamples.spf from the Composer File menu. The file is located in:

..\exteNdComposer\Samples\ActionExamples\ActionExamples.spf

You can find other Action Model examples like the ones in this chapter in the Tutorial project.

About Element and Data Mapping

One of the powerful tools in exteNd Composer is element mapping. You can map elements between DOM trees with different structures, allowing you to pass data between XML documents.

NOTE: For a summary of the basic mapping behavior in Composer, see the table "Map Type" on page 141.

Mapping Leaf Elements

Many of the element mappings you create with Composer will be between leaf elements (terminal nodes) of two DOMs. For instance, you might map a product SKU in the Input Part to a product part number in the Output Part. When the service executes, the transfer between the two Parts takes place and the SKUs from the input XML document are written to the part numbers in the output XML document.

One method of mapping two leaf elements is to select them in the Input and Output panes of the XML Map Component Editor and add a Map action.

NOTE: By default, Composer's Map actions transfer *element* data, but not attribute data.

➢ **To map leaf elements using the Action menu:**

1. Open a component.
2. Select a line in the Action Model pane where you want to place the Map action. The new Map action will be inserted below the line you selected.
3. In the Input pane, expand the Input Part until you see the leaf element you want to map.
4. Select the leaf element.
5. Repeat steps 3 and 4 in the Output pane.
6. From the **Action** menu, select **New Action**, then **Map**.
7. When the Map dialog box appears, click **OK**. The mapping from input element to output element is created automatically.

You can also use Composer's *drag and drop* feature to map an input leaf element to an output leaf element, as explained next.

➢ **To map leaf elements using drag and drop:**

1. Open a component.
2. Select a line in the Action Model pane where you want to place the Map action. The new map action will be inserted below the line you selected.
3. In the Input pane, expand the Input Part until you see the leaf you want to map.
4. In the Output pane, expand the Output Part until you see the leaf you want to map.
5. Select the Input leaf element.
6. While holding the left mouse button down, drag the Input leaf element on top of the Output leaf element.
7. Release the mouse button. The Map action appears in the Action Model pane.

Mapping a Parent and its Children (Deep Copy Mapping)

The second way you can map elements is to map a *parent element* to the target Part. When a parent element is mapped, all of its child elements and their attributes are included in the mapping. For instance, suppose you select a parent element named `Line_Item`, and it has child elements that include `Item_SKU, Item_Description, Item_Quanity,` and `Item_Cost`. Suppose you map the `Line_Item` element to an element in the Output Part named `PO_Line`. The resulting map action transfers the `Line_Item` element *and all its child nodes* to the `PO_Line` element in Output, retaining the original branch's structure.

NOTE: The default mapping behavior of Composer's Map action can be overridden. See

➢ **To map a parent element and all its children:**

1. Open a component.
2. Select a line in the Action Model pane where you want to place the Map action. The new Map action will be inserted below the line you selected.
3. In the Input pane, expand the Input Part until you see the parent element you want to map.
4. Select the parent element.
5. Repeat steps 3 and 4 in the Output pane.
6. From the **Action** menu, select **New Action** then **Map**.
7. When the Map dialog box appears, click **OK**.

You can also use Composer's drag and drop feature to map an input parent element to an output parent element, as explained in "To map leaf elements using drag and drop:" on page 282.

The third way to map elements is to map a parent element without its descendant elements. In essence, you are mapping the high-level data and ignoring the data and its descendants. For instance, if you map an element named Invoice and it contains descendant elements, the Output Part will only receive data pertaining to the invoice element.

➢ **To map a parent element without its child elements:**

NOTE: Since the default Map action behavior is to transfer descendant elements, you need to create and apply an ECMAScript method to the element name to map only the element.

1 Open a component.

2 Select a line in the Action Model pane where you want to place the Map action. The new Map action is inserted below the line you selected.

3 In the Input pane, expand the Input Part until you see the parent element you want to map. Select it.

4 Do the same in the Output pane.

5 From the **Action** menu, select **New Action** then **Map**.

6 When the Map dialog box appears, select Expression and click the Expression builder button in the Source.

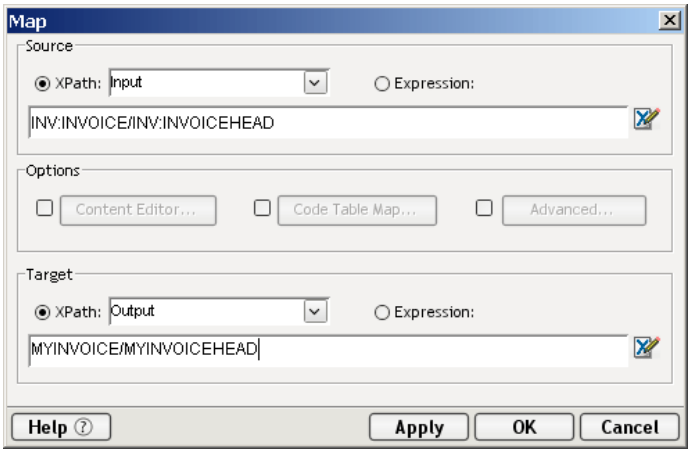
7 Type the following in front of the XPath fragment: `Input.XPath("`

8 Enter the XPath fragment.

9 Type the following at the end of the XPath fragment: `").toString()`

10 Click **OK** twice.

The following illustration shows a Map action for a parent element without its child elements.



Transforming Elements

There will be times when you want to map two elements that have different formatting. For instance, the element leaf in the Input DOM might be formatted with four numbers and uppercase characters (1234CAT) while the output element leaf might be formatted with lowercase characters and six numbers (cat001234).

Composer provides three methods for transforming element formatting so data can be mapped appropriately between DOMs. The three methods are all available from the Map Action:

- The Content Editor
- Code Table Maps
- Functions

Transforming Elements With the Content Editor

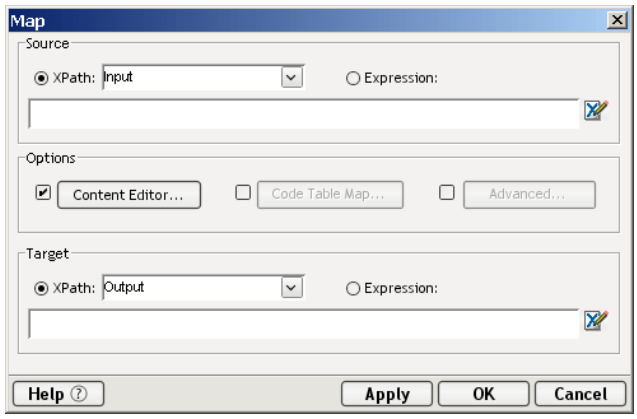
The Content Editor allows you to change the format and content of the input element to match that required by the output element. Using the Content Editor, you can slice the input data into small parts, move the parts to different locations relative to one another, add new parts, omit some parts, and apply functions to individual parts.

➢ **To access the Content Editor:**

1 Open a component.

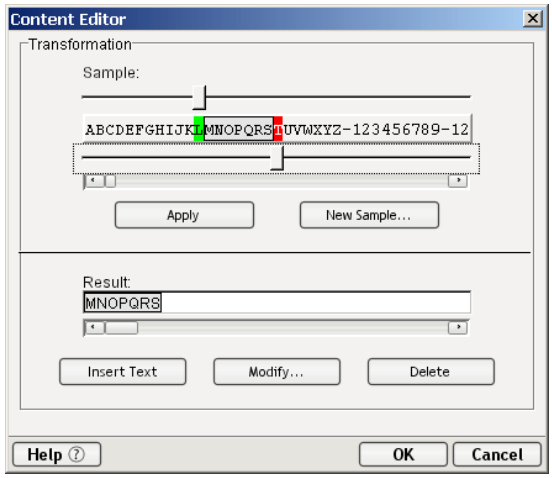
2 Select the two elements to map from different Parts.

3 From the **Action** menu, select **New Action** then **Map**. The Map dialog box appears.



4 Click the checkbox beside the **Content Editor** button. This enables the Content Editor button.

5 Click the **Content Editor** button. The Content Editor appears.



6 Optionally click the **New Sample** button and enter a sample string.

Dismiss the dialog.

7 In the **Sample** field, move the top slider to the position where you want the first cut to take place and the bottom slider to the position where you want the end cut to take place. The sliders determine how to take a substring from the input data.

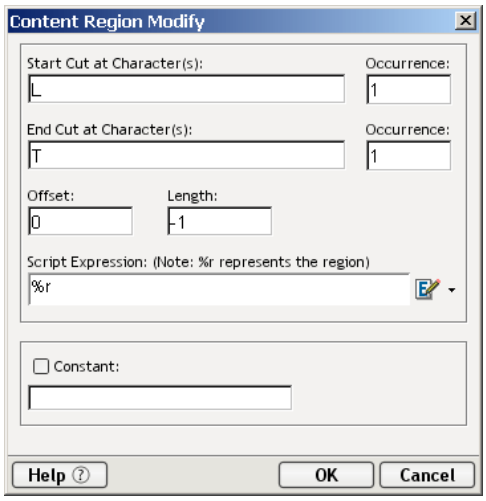
8 Click **Apply**. The substring is copied to the **Result** field as a separate object.

9 Repeat steps 6 through 8 for each part of the sample you want, in the result in the order you want. In this way, you can build a new string out of portions (substrings) of the original input.

➢ **To change the format of an object in the Result field:**

1 Select an object.

2 Click **Modify**. The Content Region Modify dialog box appears.



NOTE: The **Start Cut at Character(s)** field displays that character in the string where the first cut will take place. The first **Occurrence** field displays when the cut will take place. In the previous illustration, the first cut will take place at the first occurrence of the letter **T**. The **End Cut at Character(s)** field displays that character in the string where the last cut will take place. The second **Occurrence** field displays when the cut will take place. The Offset field displays the number of characters from the beginning of the original string where the object will start. The Length fields displays the length of the object.

3 The Script Expression field supports the ECMAScript expression builder. Any content region created by the Content Editor can have the full functionality of the expression builder applied to it.

NOTE: The %r is a local variable representing the content region to which you would like to apply a function. For example, if you want to apply the uCase() function to the content region, you would write the Script Expression as: uCase(%r).

4 You can assign a constant to an object by highlighting it, checking the **Constant** box, and typing a constant string.

5 Click **OK** to apply any format changes.

When you are finished mapping string formats with the Content Editor, click **OK** to save the changes and close the Content Editor.

Transforming Elements With Code Tables

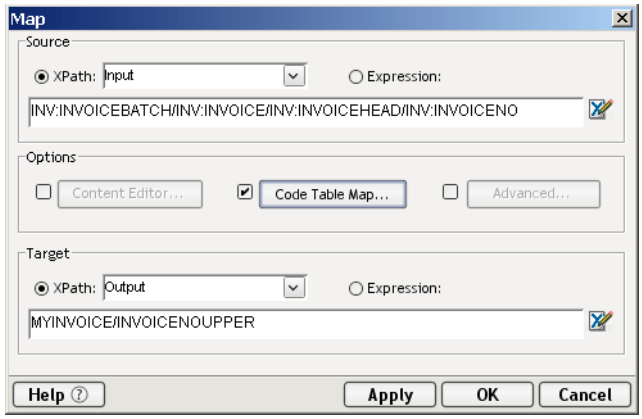
Mapping with Code Tables allows you to automatically transform one set of codes used in the Input Part into another set of codes used on the Output Part. In order for you to transform elements with Code Tables, you must have already created Code Tables and Code Table Maps.

➢ **To transform elements with Code Tables:**

1 Open a component.

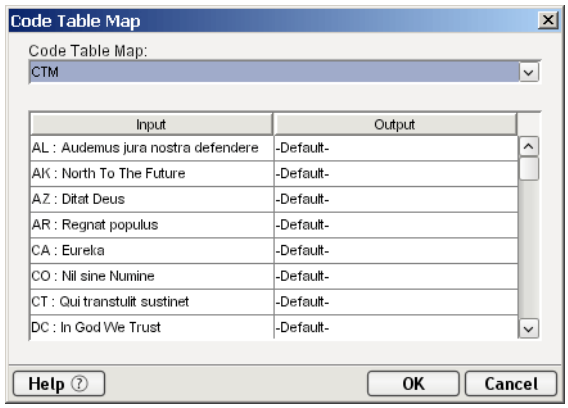
2 Select two elements to map.

3 From the **Action** menu, select **New Action** then **Map**. The Map dialog box appears.



4 Click the checkbox beside the **Code Table Map** button.

5 Click **Code Table**. The Code Table Map dialog appears.



6 Select a Code Table Map.

7 Click **OK** to assign the Code Table Map.

8 Click **OK** again to save the Map action.

Transforming Elements With Functions

You might come across situations where the Content Editor is not sufficient to transform element format structures. For instance, you might want to extract the month number from a date format (i.e. **5**/23) and convert it to the month name (**May** 23). You can perform custom transformations by creating ECMAScript and XPath custom functions and applying them to element expressions.

Composer comes with a library of sample custom script functions organized in the following categories:

- String
- Math
- File
- General
- Date
- Database

You can import a category of functions from the **\exteNd\Samples\CustomScripts** subdirectory.

➢ **To apply an ECMAScript custom function to an XPath expression:**

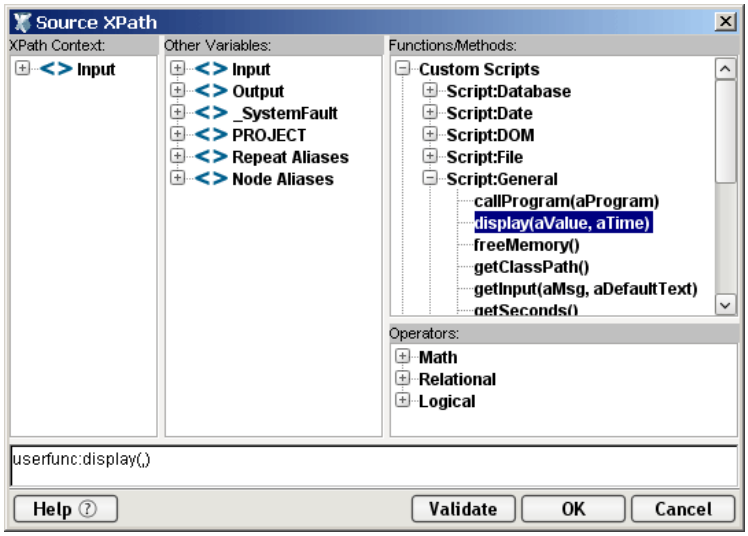
1 Open a component.

2 Select an input and output element to map.

3 From the **Action** menu, select **New Action** then **Map**. The Map dialog box appears.

4 Open the XPath expression builder.

5 Use the pick-lists to navigate to the custom script function you want and double click.



6 Edit the expression as necessary to make it syntactically correct.

7 Click **OK** to add the Map action.

NOTE: When transforming element data within a Map action using a function, make sure that the result of the function returns a fully qualified DOM element name.

If you want to transform an element's data outside of a Map action, use the Function action. See "The Function Action" on page 135.

"Userfunc:" is a bridge Novell extension method that allows you to use ECMAScript function on XPath expressions. XPath also has limited set of native functions categorized as Node-set, String, Boolean, and Number. These functions do not require the use of the userfunc: keyword. For more information, refer to the "XML Path Language(XPath)" doc provided in the exteNd/Docs/XPath directory.

Using Loops in Action Models

In the chapter on Advanced Actions, you read about the three Repeat actions and how they are used to perform iterative processing within an Action Model. This section further explains the Repeat actions and shows how they are used to read, map, and write data Input and Output Parts.

The Repeat action has three types of loops. They are:

- Repeat for Element
- Repeat for Group
- Repeat While

The Repeat for Element Action

XML allows multiple instances of an element in a document. The number of instances can vary from document to document. For instance, you might receive an XML document containing invoices on a daily basis. Each day the XML document has a different number of invoices. Not knowing how many instances of the invoice are in the XML document poses a problem if you want to transfer the invoice number from each invoice in the input XML document to an output XML document. The Repeat for Element action solves this problem.

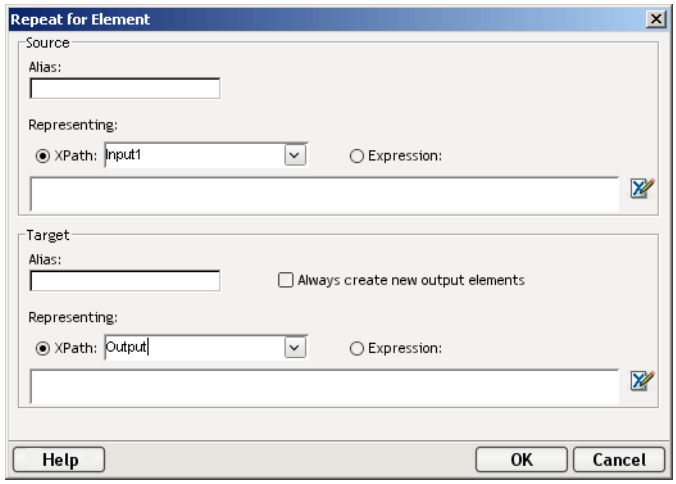
The Repeat for Element action allows you to mark an element that occurs multiple times. The action then sets up a processing loop that executes one or more actions for each instance of the marked element until no more exist. In the example above, the processing loop would contain a single Map action to transfer the invoice number.

The Repeat for Element processing loop allows you to process more than one action. In the simplest case, the repeat loop might only contain one Map action that transfers the value of the current instance from the Input DOM to the Output DOM. You can also set multiple actions in the processing loop: a Map action to transfer the current value and a Log action that writes to a file, creating an audit of each transfer.

The first step in adding a Repeat for Element action is to position the cursor in the Action Model pane where you want the repeat processing to take place.

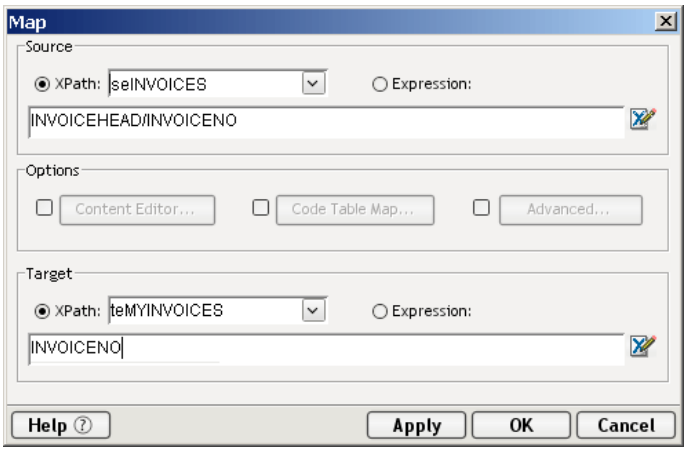
➢ **To add a Repeat for Element action:**

1. Open a component.
2. Select a line in the Action Model pane where you want to place the For Element Repeat action. The new action is inserted below the line you selected.
3. In the Input DOM, select the first instance of the element that repeats.
4. Using the context menu, select **Repeat for Element**. The Repeat for Element dialog box appears.



5. Type an alias name for the **Source** element.
6. Accept the default XPath, or select **Expression**, and type in a valid expression.
7. Repeat steps 4 through 6 for the **Target**.
8. Check the **Always create new output elements box** if you have repeating actions which should add new elements rather than updating existing ones.
9. Click **OK**.

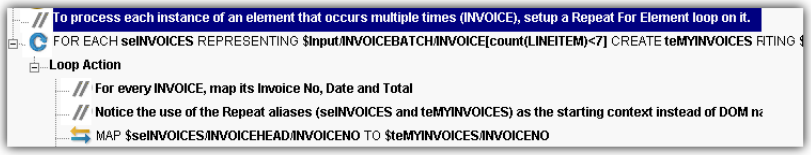
Once the Repeat For Element action is created, you can add a Map (or any other) action within the loop. For instance, to simply transfer the invoice number element from an input XML document to an output XML document, define a Map action as shown in the following illustration:



Notice the use of the repeat alias as the XPath context. The alias is defined in the Repeat action and resolves to an actual DOM name and path.

The **Source** field specifies that data from the location in the Input Part (seINVOICES/INVOICEHEAD/INVOICENO) will be transferred to a location in the Output Part (teMYINVOICES/INVOICENO).

The Repeat for Element action and the Map action should appear in the Action Model pane as shown in the next illustration.



The Repeat for Group Action

The format of an XML document that you receive is not always the format that will meet the requirements of your business process. For instance, an XML document might contain invoices from different sellers. The data is received as individual invoices, but in the context of a business-to-business transaction, you might need to summarize the data and send the summary data to a manager, and at the same time, send the invoice data to the Accounts Payable department.

A Repeat For Group action allows you to re-structure your data and establish a framework to calculate aggregates on your data. Grouping allows you to select a repeating element in your Input Part and create fewer elements based on the unique values across all instances (siblings) of that repeating element. Instead of multiple seller elements across the invoices (some with the same seller value), you end up with one element for each unique seller value in our Output Part.

The Repeat For Group action sets up a processing loop that executes for each unique value in the group. Once you have one element per seller, you can add Map actions to the processing loop to calculate how many invoices each seller had. You can also list the individual invoice numbers beneath each seller. By combining a Repeat For Group processing loop with Map commands, you can create a new XML document whose structure and data are different from the original.

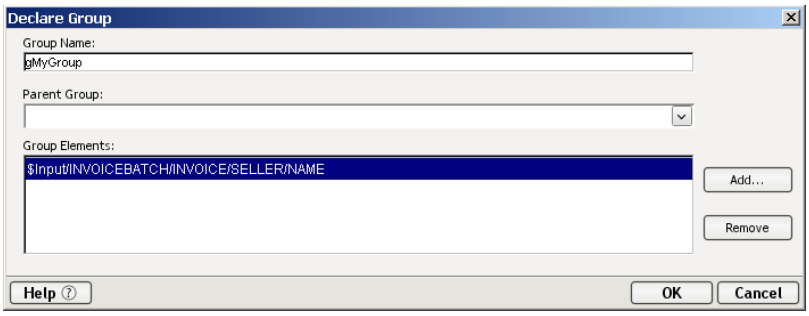
To create an action Repeat For Group, you need to complete these three tasks:

- Create a group to identify the repeating element.
- Create an action Repeat For Group.
- Create Map actions inside the loop.

➢ **To create a group:**

1 Select the element in the Input Part on which you want to repeat.

2 Click the right mouse button and select **Declare Group**. The Declare Group Info dialog box appears.



3 In the **Group Name** field, type in the alias name you wish to use to reference the group by in your Map actions.

4 If you want to create multiple group levels, select a group in the **Parent Group** field.

5 The **Group Elements/Attributes** field specifies the full name of the element you selected. If you wish, you can add other elements to this list, thus creating groups based on the concatenation of two or more values in different elements.

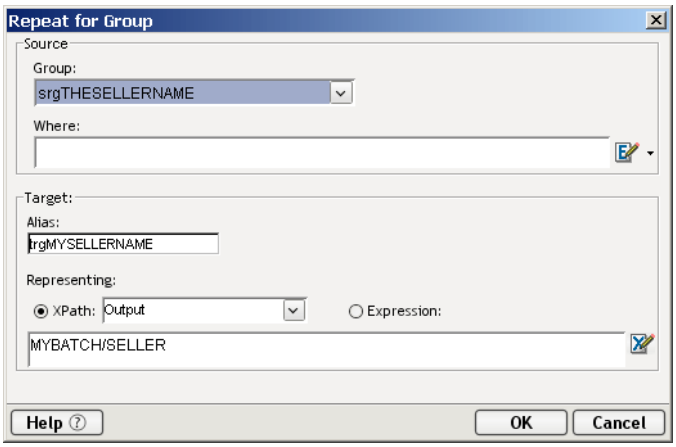
6 Click **OK** to save the group. A **Declare Group** line appears in the Action Model.

Once you have created a group based on an Input Part element, you can create a Repeat For Group action.

➢ **To create the Repeat for Group action:**

1 Select a line in the Action Model where you want to place the Repeat for Group action. The new action is inserted below the line you selected.

2 From the **Action** menu, select **New Action>Repeat** then **Repeat for Group**. The Repeat for Group dialog box appears.



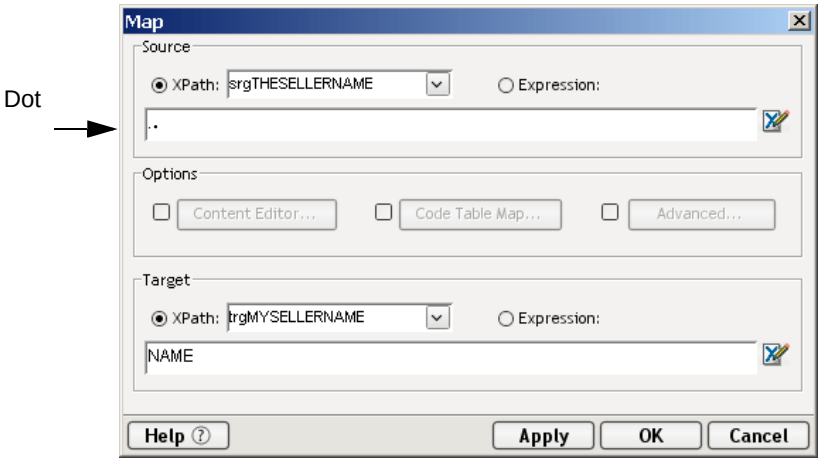
3 The Source fields specify the basis for the processing loop. Select the group you wish to use as the basis for the loop.

4 The optional **Where Script Expression** field allows you to selectively omit some repeating elements from the group processing. Type an expression, or click the **Expression Builder** button, and write an ECMAScript expression that determines which elements participate in the group.

5 The optional Target fields allow you to specify the position in the Output Part to place data mapped within the Repeat for Group action. Give the Target an alias, select a Part and specify an XPath. This alias is used as the target context for Map actions within the loop.

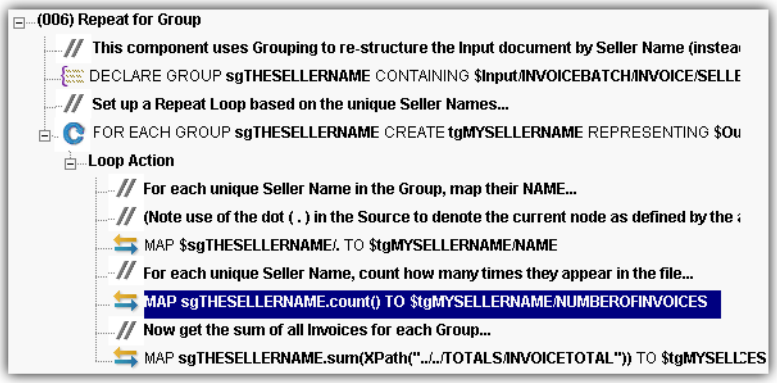
6 Click **OK** to complete the Repeat for Group action.

Once the Repeat for Group action is created, you can add one or more Map actions within the loop. The following illustration shows a Map action using groups as the Input and Output Part elements.



Notice the use of the dot. It indicates the current location, which is whatever the context "sgTHESELLERNAME" resolves to (defined in a Declare Group action earlier in the Action Model).

The Repeat for Group action and the Map action should appear in the Action Model pane as shown in the next illustration.



The Repeat While Action

The Repeat While action creates a processing loop based on any criteria you wish to define. This gives you a different kind of flexibility in creating repeat loops than do the Repeat for Element and Repeat for Group actions, both of which base their looping on data in a document or DOM.The Repeat While action allows you to base your processing loop on any valid XPath or ECMAScript expression.

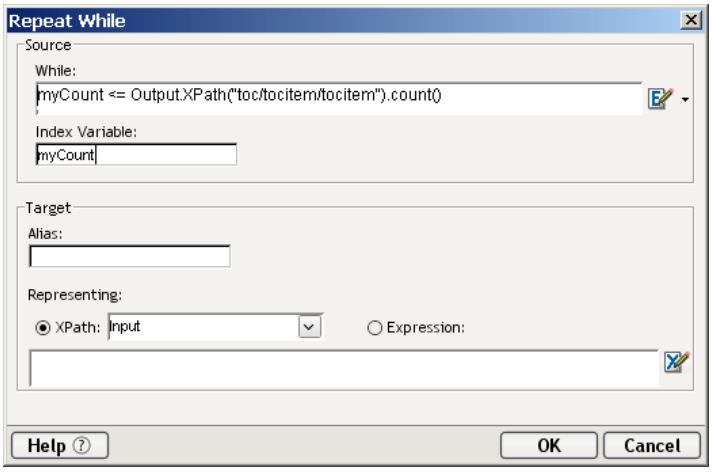
For example, you could base the execution of your loop on an ECMAScript expression that looks at the system clock to determine when to break out of the loop. In another example, you could base your loop on the existence of files in a directory. In this case, the actions within the loop will process the files and the loop will only break when no more files are present.

To create a Repeat While action, you need to perform the following tasks:

- Select a place in the Action Model pane where you wish to place the Repeat While action.
- Create the action.
- Create one or more actions inside the Repeat While action.

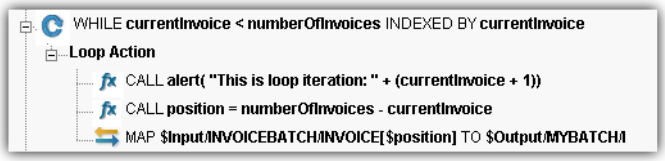
➢ **To add a Repeat While action:**

1. Select a line in the Action Model where you want to place the Repeat While processing loop. The loop is inserted below the line you selected.
2. From the **Action** menu, select **New Action>Repeat** then **Repeat While**. The Repeat While dialog box appears.



3. The **While** Script Expression field is where you type an ECMAScript expression. When it evaluates to false, the execution of the loop will stop. You can also press the **Expression Builder** button and type an expression or select from a list of pre-written expressions.
4. The **Index Variable** field allows you to create a name for a loop counter. This counter is incremented each time the loop executes. You can capture its value in the **While** Expression to further control the loop processing.
5. Optionally, you can enter **Target** information. Enter an alias and select either **XPath** and a DOM element, or **Expression** and type in a valid expression. You can also click the **Expression Builder** button and build an expression.
6. Click **OK** to complete the Repeat While processing loop.

Once the Repeat While processing loop is created, you can add one or more Map actions within the loop. The following illustration shows a Repeat While loop with two Map actions.



Performing Aggregate Calculations

The aggregate calculations include the following examples which can be found in the component named: [008] aggregate calculations in the Action Examples project.

- Calculating a sum
- Finding the highest total
- Finding a specific match for the highest total

Calculating a Sum

Suppose you have a component that is handling the processing of invoices, and you want to calculate the sum of line item totals (before taxes) across all invoices.

A simple ECMAScript expression using an XPath syntax does the trick. Create a Map action and select the Expression radio button for Source. Type the following ECMAScript expression:

```
$Input.XPath("//LINETOTAL").sum()
```

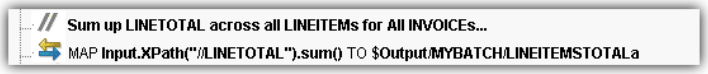
For the Target, select Output, and specify an XPath to receive the result:

```
MYINVOICEBATCH/LINEITEMTOTALa.
```

The Source expression uses the XPath // pattern symbol to select all Input nodes regardless of parentage labeled LINETOTAL, and then applies a Novell aggregate method: `sum()`.

NOTE: Because no XML template was specified for the Output, it will be built dynamically (i.e, if a Map Action does not find the element specified in the "Map To" control, it will be created).

Here is an example of what the action looks like:



Finding the Highest Total

Suppose you have a component that is handling the processing of invoices, and you want to find the highest total of the invoice amounts.

To find the maximum of an element across multiple INVOICEs, you can specify the "max" method in a Source specification. This establishes the context for the `max()` function. Then you can continue the specification down to the point of the DOM tree where the element you are interested in finding the max of resides "TOTALS.INVOICETOTAL."

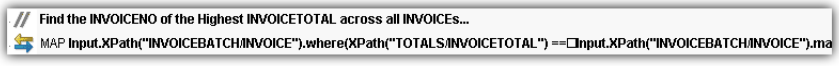
Here is an example of what the action looks like:

Finding a Specific Match for the Highest Total

Continuing with the previous invoice example, suppose you want to select the one invoice that matches the highest total.

To look across all INVOICEs but only select one of them, you can specify the "where" method in a Source specification, (the where method implies that you will be processing each INVOICE). The specification continues by comparing TOTALS\INVOICETOTAL for each INVOICE against the "max" of all the TOTALS.INVOICETOTALs. The max is found and compared against the value of each INVOICE. Once the match is found, the specification continues to retrieve the INVOICENO.

Here is an example of what the action looks like:



12 Testing and Debugging

Novell exteNd Composer provides many aids to testing and debugging. In fact, in most cases you'll find it possible to do end-to-end animation (complete roundtrip testing) of all services in your project without having to deploy to the app server. Document input/output, XML transformations, transaction-control logic, logging, etc., can be tested in real time using "live" connections to back-end systems. Since Composer's design-time/debug-time environment executes against *exactly the same Java classes* that are used in the Composer Enterprise Server environment, you can have high confidence that if a service is trouble-free in the design-time setting, it will operate reliably on the app server.

Among the powerful testing and debugging features offered by Composer are:

- Robust step-into/step-over animation capability, allowing you to execute a component's action model one line at a time (and pause or abort animation at any time)
- Assign unlimited breakpoints and use run-to-breakpoint animation to get quickly from one breakpoint to another
- Create "watch lists" of variables and XML elements whose values you want to observe in real time, at various points in an execution cycle
- Highly fine-tunable logging capability
- Fault documents and a "Throw Fault" action for flexible exception handling.
- Try/On Fault action allows you to trap exceptions, handle them gracefully, and continue executing
- "To-do lists" for keeping track of pending-action items
- Conditional enablement of debug code
- Indirection of test values through Project variables

These features and strategies for their use will be discussed in this chapter.

What are the Animation Tools?

Composer's service and component editors provide *animation tools* that allow you to test and troubleshoot actions interactively within your services and components. You can execute a service or component's Action Model step-by-step and watch the result of each action. Not only will you see any errors as they happen, but you can verify that connections and data have behaved as you had planned.

For concentrating one particular section of an Action Model, the animation tools allow you to toggle one or more *breakpoints*. When used in conjunction with the run-to-breakpoint tool, breakpoints allow you to quickly run through action-model sections that work properly, coming to a stop at exactly a particular action, From there, you can step through each action in sequence. You can also, optionally, *step over* loops, Component Actions, and other code blocks that would otherwise be tedious to "step into."

The Basic Animation Tools

The animation tools are available on the service and component editor Action Model tool bars. In addition to the Action Model tool bars, the editors provide menu options on Composer's **Animate** menu, as well as corresponding keyboard commands. The table below describes the various tools and their functions.

Animation Toolbar Button	Description
	Start Animation—Clicking this button starts the animation process. Optionally, you may select **Animate>Start Animation** or press the **F5** key.
	End Animation—Clicking this button stops the animation process. Optionally, you may select **Animate>End Animation** or press the **Shift + F5** keys.
	Step Into—Clicking this button executes the currently selected action and highlights the next sequential action. If the currently selected action is a Component, Repeat, Decision, or Try/On Fault action, then the next highlighted action becomes the details of those actions. For a Repeat Loop pressing Step Into will execute each action in the loop as well iterate through each loop. For a Decision Action, Step Into will process the next action in the True or False branch. For the Try/On Fault action, Step Into will process the next action inside the execute, and possibly the On Fault branch. For a Component action, a separate window will open and you will "step into" that component. Optionally, you may select **Animate> Step Into** or press the **F7** key.
	Step Over—Clicking this button executes the currently selected action and highlights the next sequential action. Unlike the Step Into button, clicking this button does not highlight and execute the details of Component, Repeat, Decision, or Try/On Fault actions. For a Component action a separate window will NOT open when another service or component is called. It simply executes the call and moves onto the next sequential action; you essentially "step over" the called service or component, or Repeat, Decision, or Try/On Fault action. Optionally, you may select **Animate> Step Over** or press the **F8** key.
	Toggle Breakpoint—Clicking this button sets the highlighted action in the Action Model as a breakpoint. You may set more than one breakpoint. Other ways to Toggle a breakpoint include selecting **Animate>Toggle Breakpoint,** pressing the **F2** key and clicking with the **RMB** while in the action model.
	Run To Breakpoint/End—Clicking this button runs the animation to the next breakpoint or to the end of the Action Model. Optionally, you may select **Animate>Run To Breakpoint/End** or press the **F9** key.
	Pause Animation—Clicking this button pauses the animation. Optionally, you may select **Animate>Pause Animation** or press the **F6** key.

Starting Animation

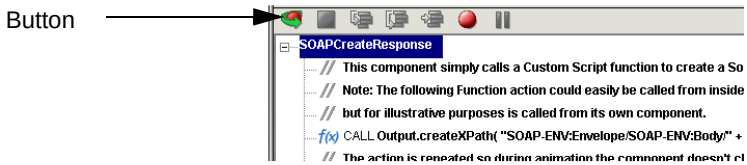
When you first open a service or component, the Start Animation and Toggle Breakpoint tools are the only "enabled" tools; the others are dimmed. Once you click the Start Animation button, the remainder of the Action Animation tools become enabled and you can click them at any point. If you want to halt the animation temporarily, you can use the Pause button. Likewise, if you want to abort the animation, you can do so at any time by clicking the red "End Animation" button.

CAUTION: *Although Copy, Paste, and action editing operations (including adding new actions) are all available at animation time, we recommend that you not edit the action model during animation. If you do, exceptions and/or unpredictable behavior may occur. If you need to edit the action model, use the Stop button to abort the animation first, then apply your edits; then begin the animation again.*

➢ **To start the animation:**

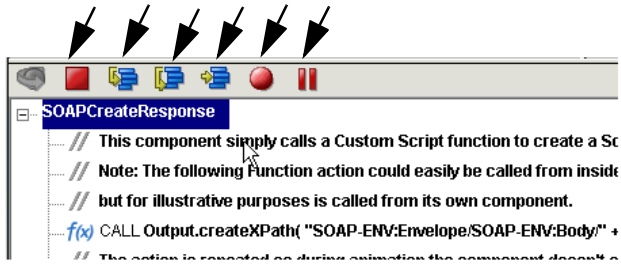
1 Open a service or component. The service or component appears in its respective editor.

Start Animation Button



2 Click the **Start Animation** button in the Action pane's tool bar, or press **F5** on the keyboard. All of the tools on the action tool bar become active, except for the Start Animation button, which is now dimmed.

Activated tools



3 Follow the instructions in the sections below to perform the desired Animation activity.

Toggling a Breakpoint

The Toggle Breakpoint tool allows you to set a breakpoint in the Action Model where you'd like the process to stop. This is especially helpful if you have a lengthy Action Model with long sections that work properly. You can set the breakpoints at the beginning of each action that is causing a problem and then step through the action to troubleshoot it.

➢ **To toggle a breakpoint:**

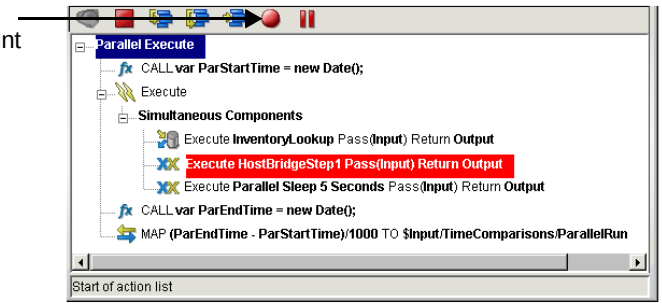
1 Open a service or component. The service or component opens in its respective editor.

2 Adjust the Action pane as necessary to view its content.

3 In the Action pane, select the action where you'd like the breakpoint to be. This is where the animation will stop.

4 Click the **Toggle Breakpoint** button on the Action Model tool bar, or press **F2** on the keyboard. The action you select changes to a red background with white text.

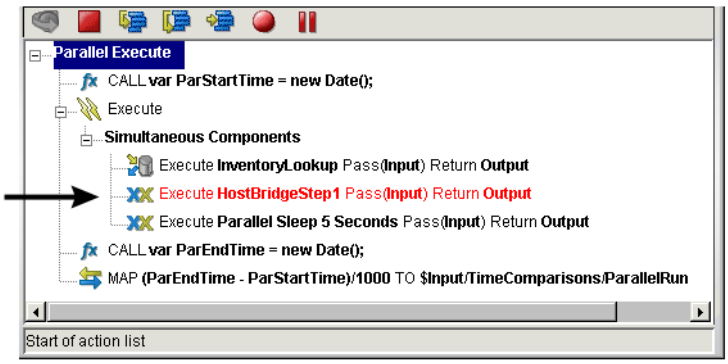
Toggle Breakpoint



5 If desired, repeat steps 3 and 4 to select additional Breakpoints.

6 If it is not grayed out, click the **Start Animation** button on the Action Model tool bar. (If Start Animation is grayed out, you can choose another animation tool to complete the animation process from the current breakpoint.) The following changes occur when you click Start Animation:

- The Start Animation button becomes inactive
- The remainder of the buttons on the Action Model tool bar become active
- The cursor moves to the beginning of the Action Model and highlights the object's name, for example, "ProductInquiry."
- The action you selected as your breakpoint changes to a white background with red text



7 Follow the instructions in the sections below to perform additional animation processes.

Running To a Breakpoint

The Run to Breakpoint tool works in conjunction with the Toggle Breakpoint tool to let you control the animation process. When you have a lengthy Action Model it is helpful to be able to control how long you want the animation process to run and at which point you'd like the process to stop. The Run to Breakpoint allows you to do just that.

➢ **To run the animation to a breakpoint:**

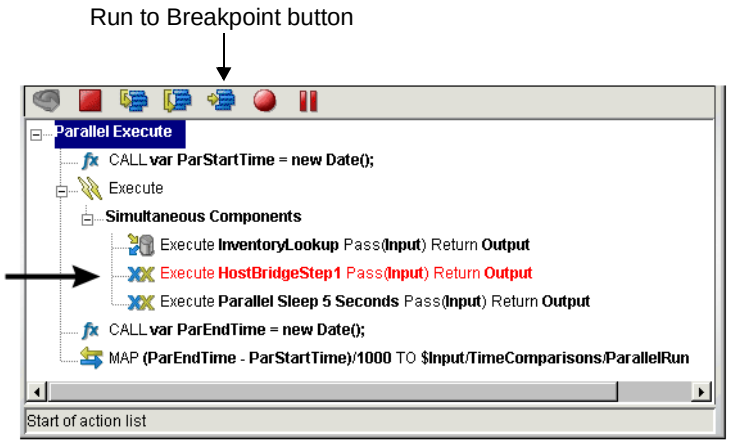
1 Open a service or component. The service or component you open appears in its respective editor.

2 Adjust the editor's panes as necessary to view the Action pane's content.

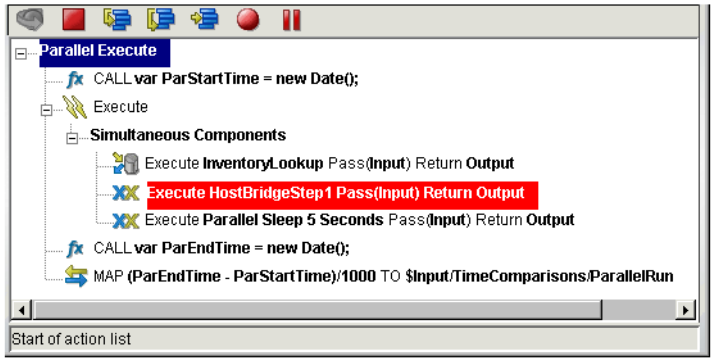
3 Select the action that you'd like to be the breakpoint for your test.

4 Click the **Toggle Breakpoint** button on the Action pane tool bar, or press **F2** on the keyboard.

5 Click the **Start Animation** button on the Action pane tool bar, or press **F5** on the keyboard. The Action pane of the service or component you've opened should look similar to this:



6 Click the **Run to Breakpoint** button on the Action pane tool bar, or press **F9** on the keyboard. The animation process runs all of the actions prior to the breakpoint, then stops and highlights the breakpoint in red, as shown.



7 Follow the instructions in the sections below to continue the animation process.

Stepping Into an Action

The Step Into tool runs the highlighted action in the Action Model and then moves to the next action in the sequence, even if it is inside another component. (In other words, if the next action is a Component Action, the target component opens and animation continues at the first action in that component's action model. You can use the Step Into tool to step through each action in the entire Action Model, or you can use it in conjunction with the Run to Breakpoint tool. Execution stops at the next breakpoint or when the action model ends, whichever comes first.

A possible scenario for using a breakpoint would be if you have ten actions that you know work properly but have doubts about the eleventh. You could set the eleventh action as a breakpoint, execute the Run to Breakpoint tool, and then step through the eleventh (and subsequent) action(s) by executing the Step Into tool.

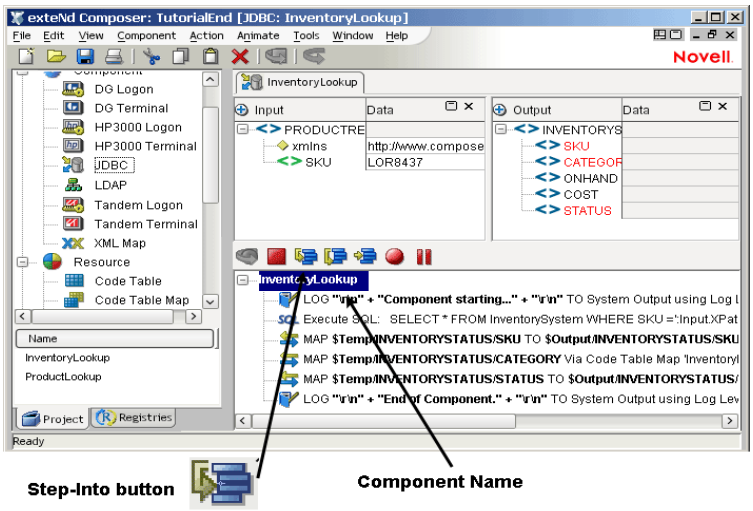
NOTE: When a service or component is called from an Action Model and a separate editor opens to display the called object, you must step through that object's Action Model to completion, at which point it closes and you return to the original Action Model.

➢ **To run the Step Into tool:**

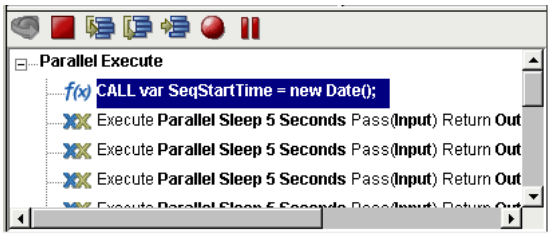
1 Open a service or component. The service or component appears in its respective editor.

2 Adjust the panes as necessary to view the Action Model's content.

3 Click the **Start Animation** button on the Action pane tool bar, or press **F5** on the keyboard. The Animation tools become active and the object's name is highlighted in the Action Model.



4 Click the **Step Into** button on the Action tool bar, or press **F7** on the keyboard. The first action becomes highlighted.



5 Click the **Step Into** button again. The action executes and the next action becomes highlighted.

6 Continue to work through the Action Model by clicking the **Step Into** button after each action executes and the subsequent action becomes highlighted.

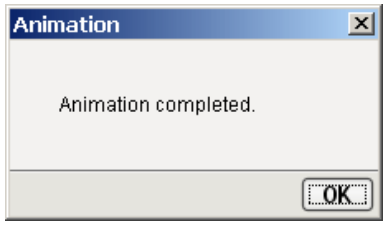
7 When an action that calls another service or component becomes highlighted, click the **Step Into** button. The following results:

- A new window opens and displays the appropriate editor (i.e., service or component).
- The Action pane displays with all tools (except "Start Animation") active.

8 Click the **Step Into** button on the Action pane tool bar of the called component.

9 Continue to click the **Step Into** button to execute all of the actions in the called component. When you've executed all of the actions, the window closes and you are returned to the point in the original Action Model where you left off and the next action is highlighted.

10 Continue to click the **Step Into** button to execute all of the actions in the original service or component. When you are done, a message appears.



NOTE: An Action Model may call one or more components, and each component may call components as well. In each occurrence of a called component, the animation tools work exactly the same. For example, you may want to Toggle a Breakpoint within a called component and then perform a Run to Breakpoint in the original service or component. The Action Model begins to execute its actions, opens the called component, and then stops at the breakpoint you've set.

Stepping Over an Action

The Step Over tool is useful when you don't want to step into the details of the Component, Repeat, Decision, or Try/On Fault actions.

For a Component action, it means you avoid the potentially timeconsuming opening of a separate editor to continue animating through the target component. The Step Over tool simply executes the target component and then highlights the next action in the Action Model.

Similarly, for blocks of code wrapped in Try/On Fault, Repeat, or other control-flow actions, using Step Over means you can execute an entire block of code at once without stepping individually through each action (which could get tiresome inside a loop).

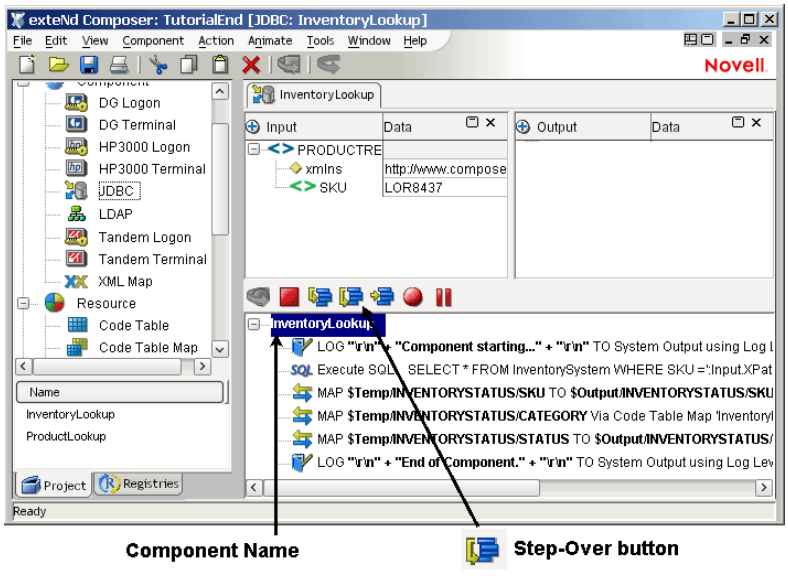
You'll often find it handy to use the Step Over tool in conjunction with the Run to Breakpoint tool. For example, you could toggle a Breakpoint, execute the Run to Breakpoint tool, and then use the Step Over tool to execute the action you've designated as the breakpoint. The Step Over tool can save a great deal of time when testing lengthy Action Models, since you can avoid tediously stepping through individual actions that might not be of interest to you.

➤ **To use the Step Over tool:**

1 Open a service or component. The service or component appears in its respective editor.

2 Adjust the panes as necessary to view the Action Model's content.

3 Click the **Start Animation** button on the Action pane tool bar. The Animation tools become active and the object's name is highlighted in the Action Model.

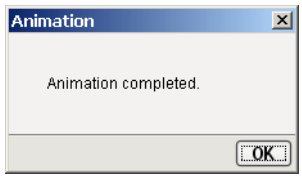


4 Step through the Action Model with the **Step Into** button until you reach a loop or other line of code that precedesan indented code block.

5 Click the **Step Over** button on the Action tool bar, or press **F8** on the keyboard. The first action *after* the block of indented code becomes highlighted. (All of the indented code will execute normally and you will be taken straight to the next "outdented" action, without needing to step through the indented action lines individually.)

6 Continue to work through the Action Model by clicking the **Step Over** button as necessary.

7 Continue to click the **Step Into** and/or **StepOver** buttons to execute all of the actions in the Action Model. When complete, a message appears.



Pausing Animation

The Pause Animation tool allows you to pause the execution of an action in the Action Model. This is especially helpful in cases where Action Models contain lengthy loops.

➢ **To pause the animation:**

1 During the execution of an action, click the **Pause Animation** button on the Action pane's tool bar, or press **F6** on the keyboard.

2 To resume the animation process, click the **Step Into**, **Step Over**, or **Run to Breakpoint** (if a breakpoint has been set) as desired.

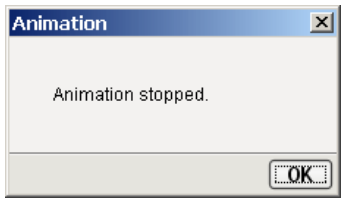
CAUTION: *Although Copy, Paste, and action editing operations are available when animation is paused, we recommend that you not edit the action model during animation. If you do, exceptions may occur.*

Aborting Animation

The Stop Animation tool simply stops the animation process. Once you stop the animation, you cannot restart from the place where you left off: you must restart from the beginning of the Action Model.

➢ **To stop the animation:**

1 While the animation is in progress, click the **Stop Animation** button on the Action pane's tool bar, or press **Shift + F5** on the keyboard. The following message appears.



2 Click **OK**.

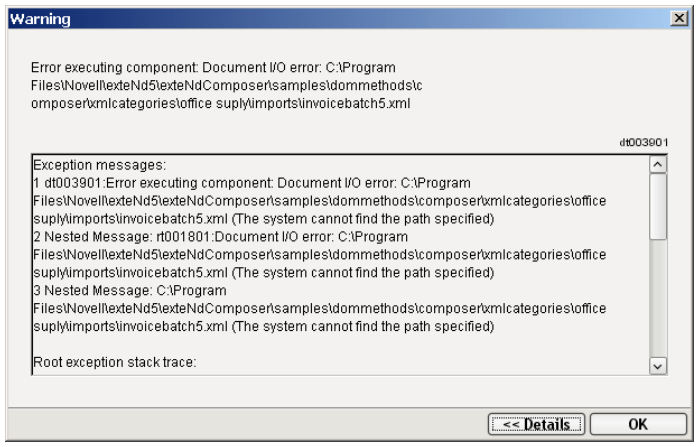
Execution Errors

If an action does not execute correctly, an error message appears.

You can click the **Details** button to read more about the problem encountered. This will give you the full Java stack trace.

Error messages are also written to the System Log, which is viewable from the Composer **View** menu.

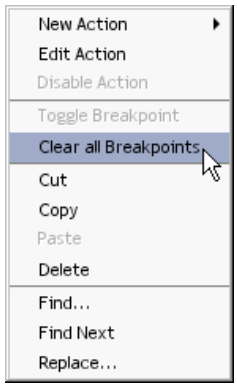
NOTE: Composer's Log (or Output) pane will also show messages. To get the most comprehensive reporting of messages to the Log tab, set the Log Threshold to one in the **General** tab under **Tools > Preferences**. Also, check the Show Stack Trace checkbox in the same dialog.



Clearing All Breakpoints

Once you have your action model working properly you will want to run through it from start to finish without interruption. To do this, you'll need to eliminate any breakpoints you had previously set. All your breakpoints can easily be removed at once using the Clear All Breakpoints menu option. This can be accessed in two ways:

1. From the **Animate** menu, select **Clear All Breakpoints**
2. While in the Action Panel of the Component Editor, right-click to bring up the contextual menu. Select **Clear All Breakpoints**.



Resetting All Documents

You may find it desirable or necessary, during a testing or debugging session, to reset the component's input, output, temp, and fault documents to their original states. You can do this manually by using the **Reload XML Documents** command under the **Component** menu on the main menubar.

Clearing a Document

You can completely clear a document, including its root node, programmatically, using ECMAScript. Insert a Function Action that contains code similar to the following:

```
Temp.removeChild( Temp.firstChild );
```

Execute the statement using the Execute button on Composer's main toolbar, or use the Apply button in the Function Action dialog. You will see the contents of the `Temp` document (in this case) "zero out."

NOTE: The `removeChild()` method and the `firstChild` property are standard DOM ECMAScript extensions defined by W3C. You can use these and other methods to remove, add, or modify any portion of a document's structure in any action that uses ECMAScript. See Chapter 10, "Custom Scripting and XPath Logic in exteNd Composer", for more information.

Testing Tips

You may find it useful to leverage one or more of the following techniques when testing and debugging action models:

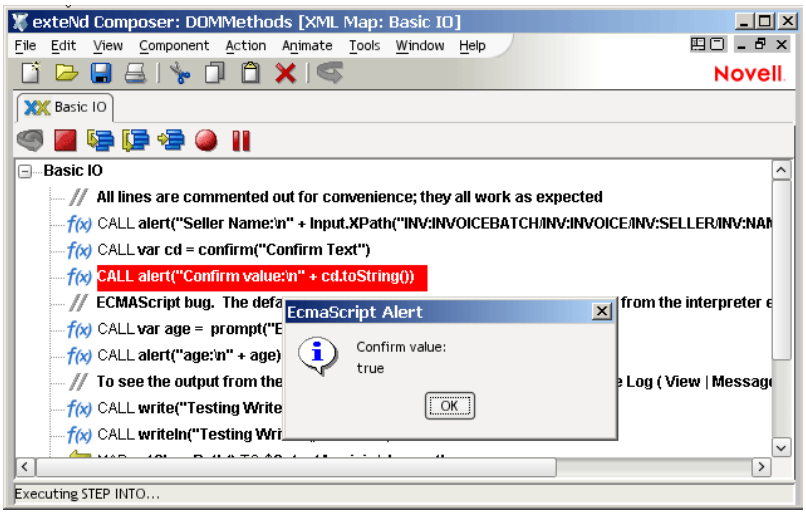
- The ECMAScript `alert()` function, as well as `java.lang.System.out.println()`
- Project variables as a way of conditionally enabling debug code
- Watch list (data values that you want to watch at animation time)
- Try / On Fault actions
- Fault documents

Each of these is discussed below.

Using the ECMAScript alert() Function

You may want to inspect a data value before and/or after running a map or other type of action. To do so, you can create Function actions that contain the ECMAScript `alert()` function. The alert function displays a message box with a value you specify.

In the following example, an alert function action has been constructed so as to display a confirmation value of true.



NOTE: You will want to disable any actions that use the `alert()` function prior to deploying your project to the app server environment. The `alert()` functionality is of use in design-time testing only. It should not be allowed to execute after deployment to the app server.

Using a Project Variable to Turn Debugging On or Off

If your component or service contains many debugging related actions, you can ensure they do not run when your project is deployed. One way to do so is to create a *project variable* that can be used in a Decision action to decide whether to execute your debugging actions. Then place all your debugging-related actions inside Decision actions in your components.

Another tactic is to use a direct call into one of Composer's internal methods to determine whether the current environment is a runtime one versus a design one:

```
gDebugMode = !Packages.com.sssw.b2b.rt.GNVxObjectFactory.isRuntime();
```

This is an example of using ECMAScript to call a custom Java method. The result here is that the (component-scoped) variable `gDebugMode` contains true if the component is running in Composer at design (or animation) time, but false if the component is running in a deployed project on a server.

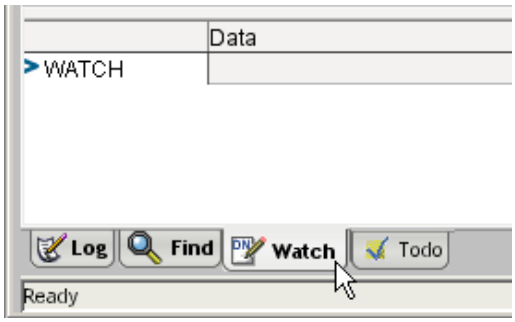
An example of using this "sentinel variable" to decide whether it's okay to call `alert()` is shown below:

NOTE: Log Actions are also invaluable for debugging and can be controlled easily via the General tab of the Preferences dialog. See "The Log Action" beginning on page 136.

Watch Lists

You can watch specific data values change at animation time by adding one or more variables or data nodes to a *watch list*.

The watch list is visible when you bring the Watch tab (in the output pane, at the bottom of Composer's main window) to the front.



The Watch list is essentially an in-memory XML document (DOM) to which "watched" data items can be added or deleted. Watch-data updates in real time as you step through a component or service. Of course, the Watch view is strictly a design-time aid: It does not exist at runtime (on the server).

Watch-List Persistence and Scope

You can create a Watch list for each component and switch back and forth between components; each list is scoped to its own component.

If you add a Watch list variable or variables to a component, then re-Save the component, the Watch list will be there again the next time you open the component. The list is persisted along with the component.

You can also Save a Watch list as an XML file at any time. (Use the **Save As XML** menu command that appears in the context menu when you right-click the WATCH node.) This can be useful for troubleshooting purposes, since it lets you compare Watch list values at the same point in two different execution runs in two different design sessions.

Types of Variables You Can Watch

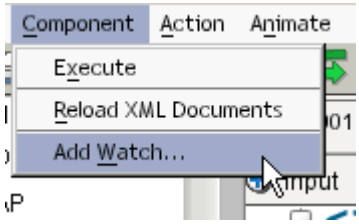
You can add two types of variables to the Watch view: ECMAScript variables and DOM nodes (for example: elements). If the DOM node is not a leaf node, it can be added to the Watch list either as a single node or with all its children. (The Watch target can be defined using XPath.)

NOTE: Watch lists are read-only. You can add and remove items in the Watch window, but you cannot *change the values* of those items yourself.

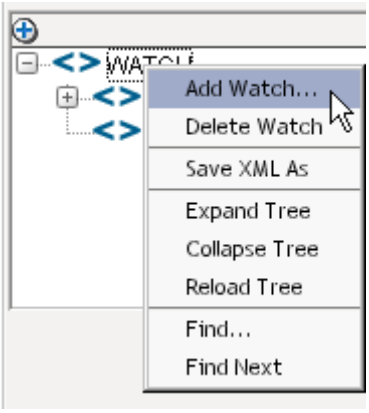
How to Add Items to a Watch List

You can add items to the Watch tree in any of three ways:

- By using the **Add Watch...** menu command under Component (in Composer's menubar), as shown below.



- By using the **Add Watch** command in the context menu that appears when you right-mouse-click on any node in the Watch view itself. Note that in addition to the **Add Watch** command, the context menu provides **Delete Watch** and other commands, as shown here:



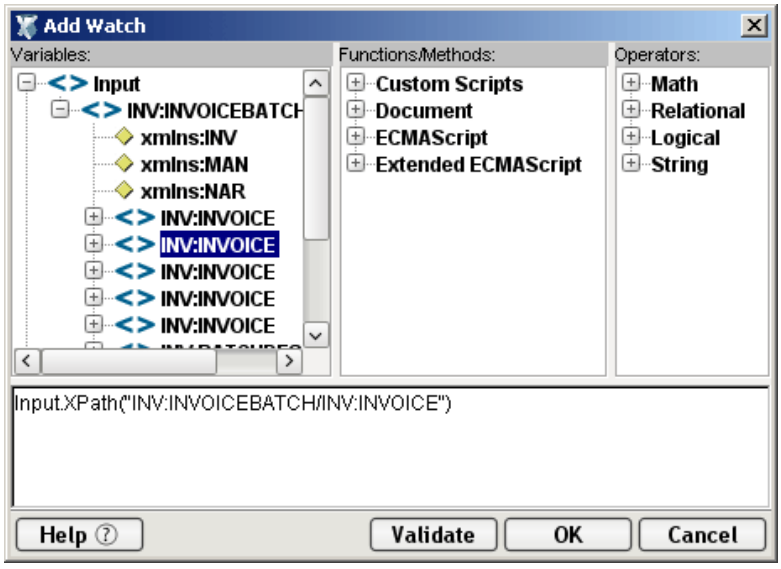
- By using **drag-and-drop**: Just pull any node of any XML document that's visible in tree-view mode down to the Watch window and let go out of the mouse. The node(s) in question will be added to the Watch list.

  NOTE: The drag-and-drop method is useful for document nodes only. To add ECMAScript variables to the Watch window, you must use the **Add Watch** menu command. See example below.

Step-by-Step Example

The following steps show how to add a variable to a watch list, animate through a Component so as to see the variable change value, and delete an item from the list.

1. Open a Component if one is not already open.
2. Click the **Watch** tab (at the bottom of the Composer main window) to bring it forward.
3. Right-mouse-click the root node ("WATCH") of the Watch view. A context menu appears. (See illustration above.)
4. Select **Add Watch** from the context menu. A dialog appears:



This dialog is an expression builder: It lets you construct an ECMAScript expression that evaluates to an XPath value. You can do this in point-and-click fashion by choosing appropriate items in the pick-lists in the top three panes of the dialog. For example, if you want to add the Input element at `INV:INVOICEBATCH/INV:INVOICE` to your Watch list, using the above example, you would simply open the Input tree nodes as shown and doubleclick the appropriate node. The corresponding ECMAScript statement appears automatically in the text-edit field, as shown above.

NOTE: If you are interested in watching an ECMAScript variable, simply type the variable's name. (You must enter it by hand since ECMAScript user variables are not shown in the pick-lists.)

5 Optionally use the **Validate** button to check the syntax of the expression.

6 Close the dialog by clicking **OK**.

7 Notice that the node or variable in question has been added to the Watch view. If the variable is a DOM node with children, doubleclick the node (or single-click the plus sign next to it) to toggle the node open, exposing its children and their values.

8 Now click the **Start Animation** button to begin stepping through your action model.

9 As you step through actions that change the values associate with the Watch variable(s), notice how the values change.

10 To delete the Watch variable, right-mouse-click the node in question and choose **Delete Watch** from the context menu. The item disappears.

Environmental Differences between Animation Testing and Deployment Testing

There are significant environmental differences between Animation testing in Composer and Deployment testing. Both types of testing are needed to adequately verify the components and services you build. The differences are detailed in the table below.

	Testing in Composer	Deployment Testing
OS	Win98 or WinNT or Win 2000	WinNT, Sun Solaris, etc.
Platform	JRE (Java Runtime Environment)	Application Server JRE
Component or Service Startup	Directly from Composer	By Service Triggers only (i.e., deployment Servlets or EJBs).
xObject access	From disk files	From a JAR file in Application Server
Runtime Context	Test individual components or components running within a service	Always from within a service
Service and Component Inputs	Input documents frequently come from sample XML documents on the local machine as well as DOMs from other services or components	Input documents are passed into the services and components via Service Triggers, or DOMs from other services or components

	Testing in Composer	Deployment Testing
Project Variables for: ◆ Log File Paths ◆ DTD URLs ◆ XSL URLs ◆ Send Mail Server ◆ XML Inter-change URLs	Usually point to locations on local machine (but could be on Servers or Web)	Should point to locations on production Servers and Web
Testing Tools	In addition to Log actions, you can use dialog boxes (ECMAScript alert() function) to display runtime values	No dialog boxes can be used
JDBC Connection	Doesn't use Server Connection Pools	Uses Server-provided Connection pools
HTTP and LDAP Connections	May or may not be pointing to local machine(s) or test servers	Should be pointing to production server(s)

13 Working with Services

A *service* is Composer's basic unit of execution: It is a Composer object (an xObject) that wrappers the various components you build, so as to create a logical unit of processing within the application server environment—one that's initiated with a request and results in a response. A service typically responds to a request by executing one or more components in a sequential and/or conditional manner (and can even execute other services). It can be, but doesn't have to be, exposed on a URL and triggered by a servlet.

Because the *service* xObject is the entry point for all web apps built with Composer, it's important that you understand the design philosophy behind services and how Composer's runtime architecture handles services. This chapter will tell you what you need to know in order to build and use services effectively.

Terminology

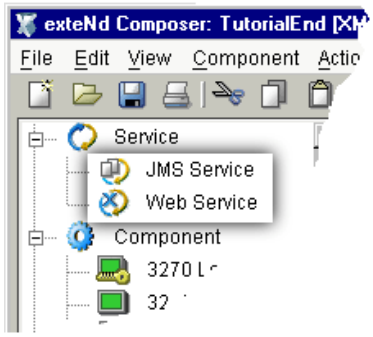
The term Web Service, as used in this discussion, is generic. It refers to any Composer-created service running on the app server, whether triggered by a servlet request in an HTTP session, arrival of e-mail, direct invocation by a custom Java class, or some other mechanism. It *may* (but doesn't always have to) refer to a web-facing service that is described by WSDL.

A *SOAP service*, in Composer, is not a service type *per se*, but a way of specifying how a service needs to be invoked (and how its data needs to be marshalled or unmarshalled) on the server. In Composer, you specify SOAP-HTTP as a *trigger type*. A specialized type of servlet is used to trigger the service.

It's also important to note that a given service can be associated with different trigger types. For example, it is possible (though perhaps not likely) that you would deploy a service with several triggers: a servlet-based trigger that handles data passed via HTTP GET; another servlet (on another URL) that handles data arriving via a form field using HTTP POST; and another that expects data passed in a String object, programmatically, via a custom application running locally on the server.

What Are the Available Service Types?

In exteNd, there are two types of services: Web Services, and JMS Services. (JMS stands for Java Messaging Service, a Sun-defined interface for message oriented middleware.) Your project, deployed (typically) as an EAR file, might contain one or more of *either or both* kinds of services. The two service types are referenced by two different icons under the "Service" heading of Composer's navigation frame (see below).



JMS Services

The JMS Service type will not be visible to you in Composer if you have not installed the Novell exteNd Composer JMS Connect (which comes as part of the Novell exteNd Enterprise Edition suite). The Web Service category, however, is always visible.

The defining characteristic of a JMS Service is how it is triggered. If a service that uses enterprise messaging will be triggered via the web, it must be created as a Web Service. If it will be triggered by arrival of a message, it must be created as a JMS Service.

Service Architecture

A service is actually a specialized type of exteNd component. As mentioned earlier, a service has an action model and can perform most of the tasks that components perform, including XML mapping, looping, logging, fault-trapping, conditional processing based on Decision and Switch actions, etc. For reasons of good design, you should limit these tasks to an exception basis only, delegating business logic to underlying components. The main actions you should use in a *service* are Component, Log, Decision, Function, Try/On Fault, and Throw Fault. (See “Building a Service with Components” on page 319 for examples of how a service uses these actions.) Anything connectivity-related, data-related, or implementing business logic, should happen at the component level.

NOTE: You can execute any number of components of any type (JDBC, XML Map, LDAP, etc.) in a service; and you can fire off those components synchronously (in serial, one-by-one fashion) or asynchronously (all at once). Also bear in mind that a service can invoke another service.

Using the *service* as the basic unit of processing in an application server should be a major goal in the design of your Composer applications.

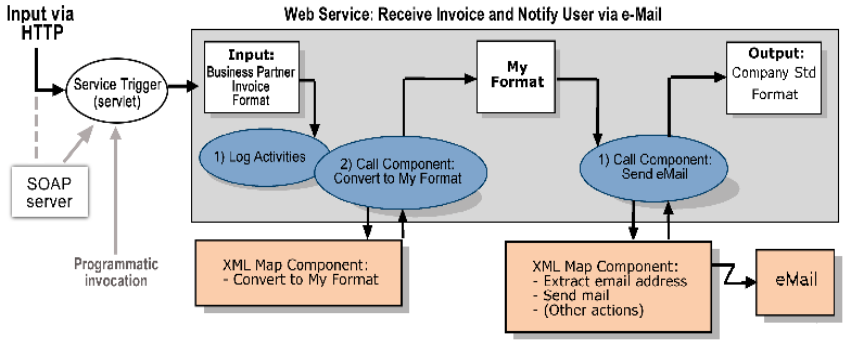
Composer Web Services and WSDL

A Composer Web Service can be, but doesn’t have to be, a WSDL-described service deployed on a URL. In simplest terms, a Composer Web Service is merely a component that calls other components. What makes it a “service”instead of a component is that the Web Service xObject can be triggered via a servlet or Java object on the server, whereas Component xObjects are not triggerable this way. (Components are *called by services*.) What makes a service a “Web Service” in the conventional sense of the term is exposure of the service as an endpoint as described in an associated WSDL file.

A Composer Web Service can implement any of the interaction patterns alluded to by WSDL: notification, one-way, request-response, or solicit-response. It can be deployed on a public URL or it can be executed as a local app. It can be associated with WSDL, or not; and it can accept SOAP requests, or not.

Looking at an Example Web Service

The following picture shows the parts and function of a Web Service and is explained below.



In the drawing, the large rectangular grey box represents a Web Service. The shaded oval shapes with numbered text represent actions in the Action Model. The input and output XML files (squares) and the called components (small rectangles outside the service) are visible.

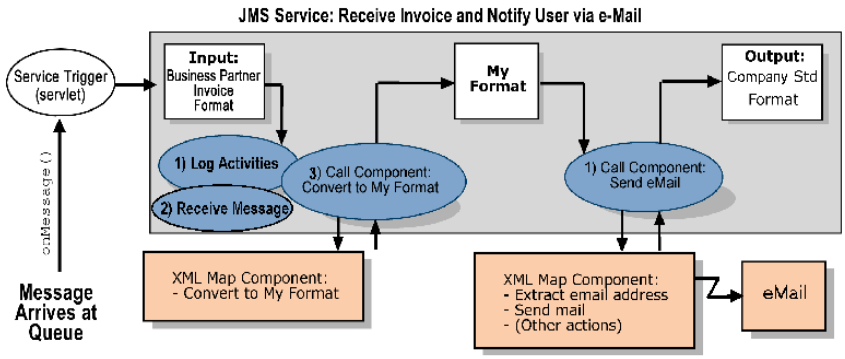
The purpose of this service is to receive an invoice (in an industry standard format) and to notify the sender that the invoice was received. Accomplishing the service requires some manipulation of the invoice, which is received as an XML document.

Here is how the service works.

1. The service is invoked by its Service Trigger (an object created at deployment time, designed to start a service in response to some external event). The servlet can be started by a business partner's application server issuing an HTTP Post to the servlet; or, alternatively (as shown by the longer grey arrow), the servlet could be invoked programmatically by a Java process on the host server.
2. The first job of the service, in this example, is to execute a Log action to write a file to record the activities of the service as they are executed.
3. The service then executes a Component action to call the *Convert to My Format* component.
4. The *Convert to My Format* component uses an industry standard invoice format as input and returns an XML file formatted to the company's internal format (*My Format*), as output.
5. The service executes another Component action to call the *Send Email* component. The *My Format* file is the input for the *Send Email* component.
6. The *Send Email* component executes several actions (extracts an email address from the invoice using the XML Interchange action, sends an email using the Send Mail action, and so on) and returns an XML file, *eMail*.
7. The Company standard format file is output by the service.

Looking at an Example JMS Service

The following graphic shows the parts and functions of a JMS Service.



It is important to note that the JMS Service does not differ substantially from the Web Service discussed earlier; it differs mainly in that it is invoked by the arrival of a message on a queue (or *topic*, in Publish/Subscribe parlance). The JMS Service implements a *MessageListener* object whose `onMessage()` method is called automatically when a message arrives at a queue or topic with which the listener has registered. The `onMessage()` method executes the service.

The JMS Service must, by its nature, contain one (and only one) Receive Message action, created using the JMS Connect. The Receive Message action allows the service to gain access to the incoming message's data and properly acknowledge its receipt.

The remainder of the Action Model is the same for this service as for the preceding Web Service.

NOTE: This example is relevant only if you have purchased and installed the Novell Composer JMS Connect.

Creating a New Service

You create a new service just like you create a new XML Map component. If you have not yet created any XML Map components, you must create any required XML templates before creating a service. For more information, see "Creating an XML Template" on page 81.

About Specifying XML Templates for a Service

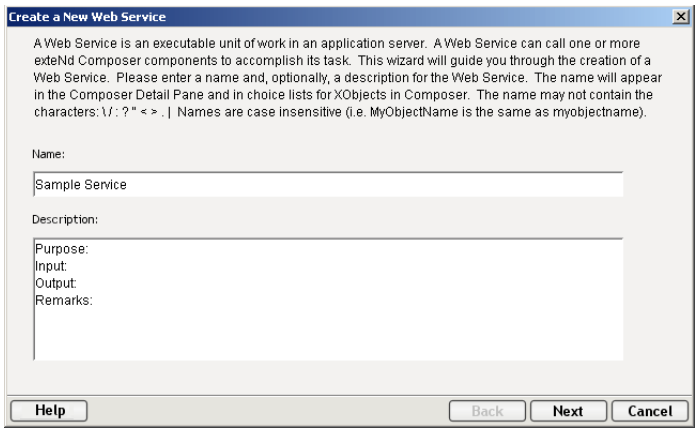
When you create a service, you specify input and output templates, just as you do for a component. If your service is designed to call components, rather than process data directly, the input template you choose for the service will often be the same template that the first component uses. The output template will often be the same one that is output for the final component in the sequence.

If you intend to create a SOAP service that uses custom SOAP headers, you should create XML Templates for the headers separately

➢ **To create a new Web Service:**

1 From the Composer window **File** menu, choose **New,** then **xObject**, then from the **Process/Service tab, select Web Service.**

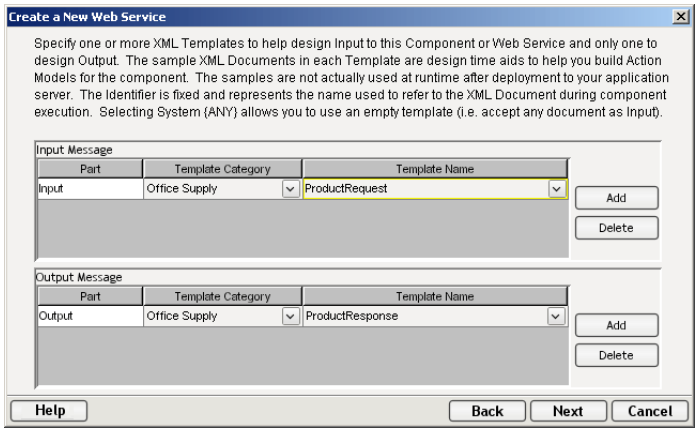
The Create a New Web Service Component wizard appears.



2 Type in a **Name** and an optional **Description**.

You can use the optional description fields to describe the tasks the service performs.

3 Click **Next** to display the Input/Output Templates panel.

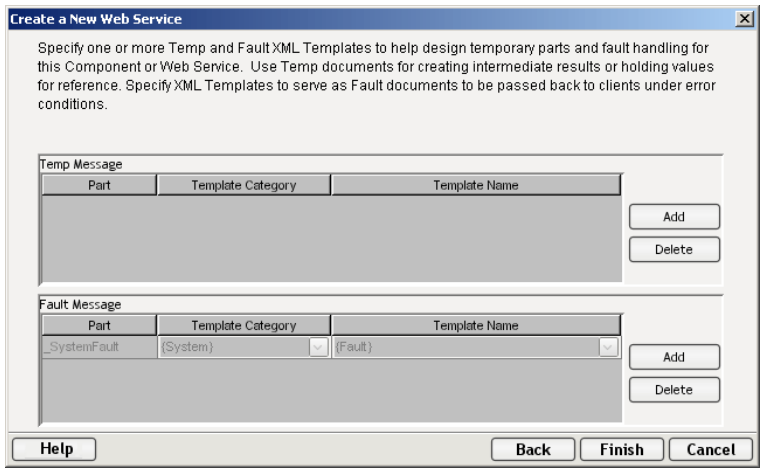


4 Specify the input and output templates as follows. See "About Specifying XML Templates for a Service" on page 314 for some tips.

- Type in a name under **Part** if you wish the Message Part name to appear in the Component Editor as something other than "Input" or "Output."
- Select a **Template Category** if it is different than the default category.
- Select a **Template Name** from the list of XML templates in the selected **Template Category**.
- To add additional input XML templates, click **Add** and repeat steps 2 through 4.
- To remove an input XML template, highlight an entry and click **Delete**.

5 Select an XML template as an output.

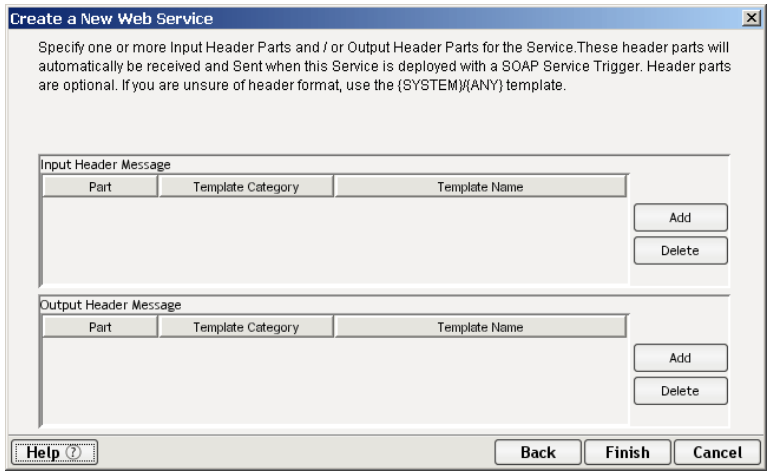
6 Click **Next**. The Temp/Fault Templates panel displays.



If desired, specify a template to be used as a scratchpad under the “Temp Message” pane of the dialog window. This can be useful if you need a place to hold values that will only be used temporarily during the execution of your component or are for reference only. Under the “Fault Message” pane, select an XML template to be used to pass back to clients when an error condition occurs.

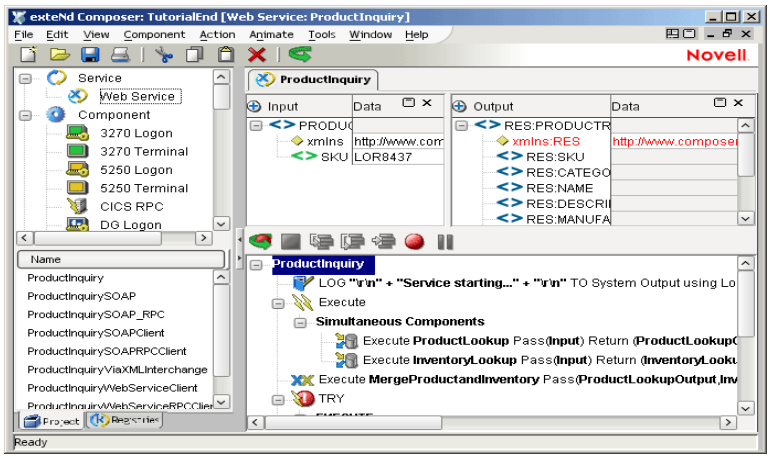
7 As above, to add additional XML templates, click **Add** and choose a Part Name, a Template Category and Template Name for each. Repeat as many times as desired. To *remove* an input XML template, select an entry and click **Delete**.

8 Click **Next**. The Input/Output Headers panel displays:



Using the methods described above for adding Input, Output, Temp and Fault Documents, specify Input and Output Header Parts for your service if it will be used with a SOAP Service Trigger.

9 Click **Finish**. The component is created, and the Service Editor appears.



If your templates have namespace declarations, Composer will generate a Declare Namespaces action for you automatically, at the top of your new Action Model.

Creating a JMS Service

Creation of JMS Services occurs via a wizard that has much in common with the Web Service wizard. For step-by-step instructions, see the *exteNd JMS Connect User's Guide*.

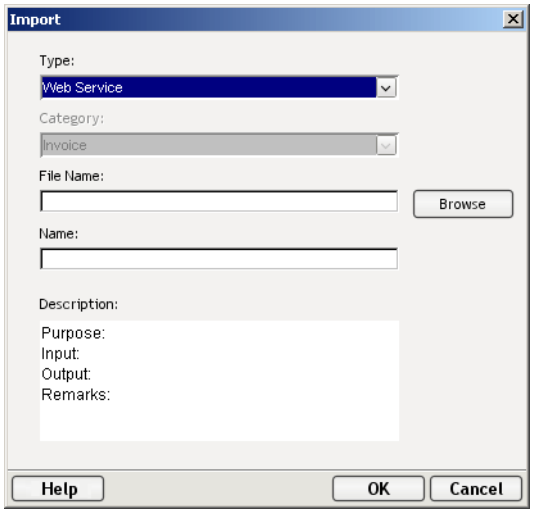
Importing a Service

The import feature allows you to create a copy of an Composer service created in another project. Once imported, you can customize the service for use within the current project.

➢ **To import a service:**

1 Right-click on the **Service** item in the exteNd Composer window, or choose **Import xObject** from the main **File** menu.

The Import xObject window appears.



2 Select **Web Service** as the **Type**, if it is not selected.

3 In the **File Name** field, type in the name of the service you wish to import, or use the **Browse** button to find it. If you import a file from a URL, you must explicitly type “http://,” “https://,” or “ftp://.”

4 Modify the **Service Name** if desired.

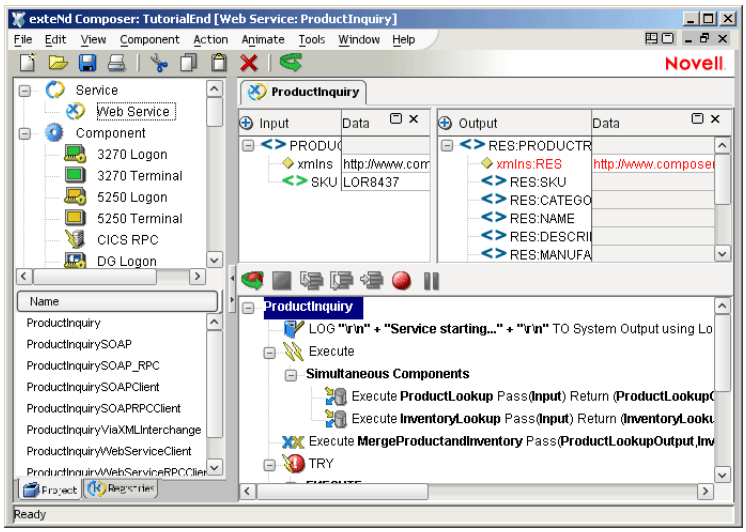
5 Type in Descriptive information if desired.

6 Click **OK** to import the service.

Understanding the Service Editor

The Service Editor is (usually) where you specify the execution of components and services as well as perform error logging, decisions, and functions. You can also map, transform, and transfer input and output structure and data.

The Service Editor provides a logical working environment for visualizing and manipulating the inputs, output, and actions of your service. The Service Editor is composed of multiple mapping panes and a single Action Model pane. The mapping panes display the DOMs of your sample input and output documents. The Action Model displays actions that operate on the DOMs.(This environment is essentially the same as the XML Map Component editor.)



Using the Service Editor

The Service Editor has all the same functions as the XML Map Component Editor. For more information on using the Service Editor, see the following topics:

- “Creating an Output Document without Using a Template” on page 113
- “Creating a Temporary Message Part” on page 115
- “Reloading an XML Document” on page 118
- “Loading a Sample Document” on page 119
- “Saving Your Component” on page 120
- “Saving a DOM as an XML Document” on page 121
- “Viewing Component Properties” on page 124
- “Printing a Component” on page 125

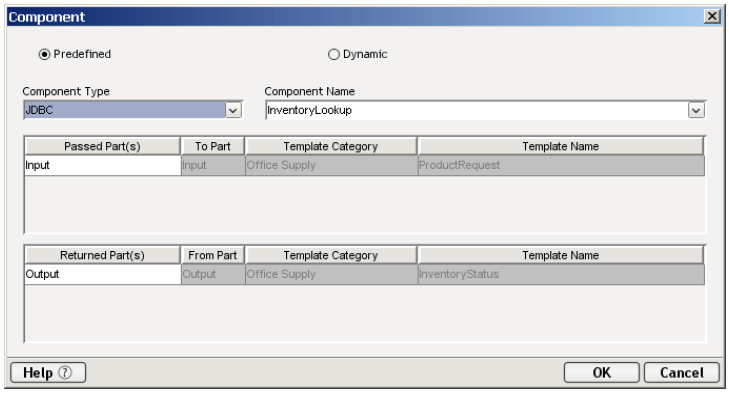
Building a Service with Components

A service is usually made up of one or more Component actions, each of which performs a specific task to map, transfer, and/or transform data for use by the next component or service called in the application.

You use the Component action to call and execute a component or service with runtime input DOMs and outputs DOMs that you specify.

➢ **To add a Component action:**

1. Select a line in the Action Model where you want to place a call to a component or service. The new action is inserted below the line you select.
2. From the **Action** menu, select **New Action** then **Component**. The Action Component Information dialog box appears.
3. Select **Predefined**, by clicking on the radio button, if it is not already selected. (See Chapter 7 for a discussion of Predefined versus Dynamic Component Actions.)

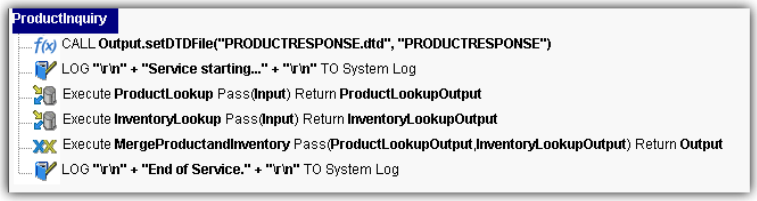


4. Select a **Component Type** from the pulldown menu on the upper left.
5. Select a **Component Name** to execute.
6. In the **Passed ID** field, select a Message Part.
7. In the **Returned ID** field, select either Output or Temp for the Message Part.
8. Click **OK**.

Looking at an Example Service Action Model

When you add Component actions to a service, they appear in the Action Model pane of the Service Editor. A service's Action Model represents the sequence in which components are called.

An example Action Model is shown below.



The Action Model functions has some logging functions and executes components as follows:

1. The first component action calls a component (**ProductLookup**). It specifies the DOM to be passed as the input document handle (**Input**) through which the component receives data from the service, and specifies the DOM to receive the component's output (**ProductLookupOutput**).
2. The second component action calls a component (**InventoryLookup**). It specifies the DOM to be passed as the input document handle (**Input**) through which the component receives data from the service, and specifies the DOM to receive the component's output (**InventoryLookupOutput**).
3. The third component action calls a component (MergeProductAndInventory). It passes the DOMs ProductLookupOutput and InventoryLookupOutput which the component receives data from the service as its Input and Input1 DOMs, and specifies a service DOM to receive the component's Output (Output).

Service FAQ

How Do I Pass Data Between Different Types of Components?

exteNd provides a variety of Connect components that access different computing environments. The inputs and outputs of all component types are simply XML documents, This means that the communication between different component types is straightforward and simple.

There are two basic methods for passing data between components. The first method uses a service to pass and receive the inputs and output from individually called components. In this method, the components don't interact directly, but instead use the service as their point of contact. The second method uses the components to call one another directly. Which method you choose depends on how you design your services and the types of tasks they perform.

Can Composer Services Accept More than One Input Document?

It depends how the service is deployed. If it is deployed as a SOAP service, your SOAP server may pass multiple input documents to your service (if multiple inputs are specified in your service's WSDL). In all other cases involving the four canonical Composer service trigger types—Params (URL/Form), XML (MIME multipart), XML (HTML form field), and XML (HTTP POST)—only *one* XML document can be accepted as input.

For information on deployment, see your *Composer Enterprise Server User's Guide.*

Can a Component Be Executed that is not Called Directly by a Service?

If you create a project with one service which calls two components, it is an acceptable design to have the first component call a third component before returning its output to the service which then calls the second component. Technically speaking, the third component is not "contained within a service" or called directly from it. The key idea to understand about a service is that only a service can be called by a "Service Trigger" object on the application server. Components don't have to be directly linked to a service, but if a component is not called somehow in the chain of events, it will never execute.

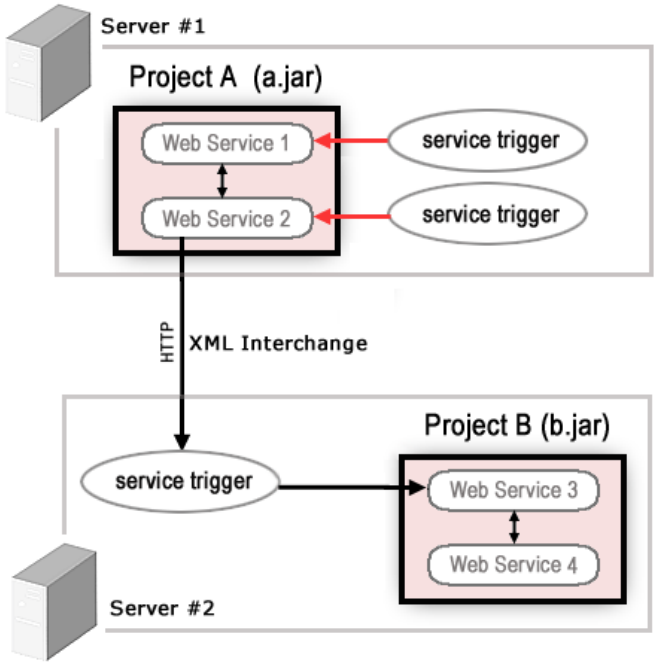
A Service Trigger object is the Java Servlet, EJB, or MessageListener (in the case of JMS) that you create with exteNd's deployment framework. This object is triggered by a URL either embedded in a Web page or called from another program on the Web. Once triggered, the Servlet or EJB starts an Composer service.

Again, for information on deployment, service triggers, framework objects, etc., see your *Composer Enterprise Server User's Guide.*

How Do I Call a Service Deployed in a Different JAR File?

Projects are deployed as JAR files, and normally, if any services or components in your project need to call on *other* services and/or components, the "called" services/components will reside in the same JAR file. But on occasion, you may find it convenient (or necessary) to have a service call another service that exists in a *different* JAR file (that is, another deployed project). You can do this by means of the XML Interchange action.

The XML Interchange action allows your component or service to output an XML document via HTTP GET, PUT, or POST protocols. By supplying the URL for another Composer service, you can have your XML Interchange action fire the servlet trigger for the service in question. See the diagram below.



In this diagram, Web Service 2 in Project A wishes to call Web Service 3 in Project B. Although Web Service 2 can call Web Service 1 directly, since it resides in the same project JAR, it cannot reach Web Service 3 directly. Instead, it must execute an XML Interchange action, which fires the service trigger for the remote service.

How Do I Log Activity in a Single File for Each Component Called from within a Service?

Log Actions write information about the activities of components within services. To create a single log file in which to record the activities of all components within a service, simply specify the same file name in the **Log to:** field for every Log Action used in the service and each component. Refer to "The Log Action" on page 136 for more information about Log Actions.

NOTE: When specifying the same file name for multiple log actions, make sure you do not select the **Clear the Log File** checkbox. Doing so will erase the log file before each log action writes to it. You may, however, want to select this option for the first Log Action encountered in the service; this clears the log so you can troubleshoot or animate an action model multiple times without continuously appending messages to the end of the file.

Loading Sample Documents as You Test a Service

Similar to using components, the XML template(s) you use as the Input(s) to your service may contain multiple sample documents. During testing, as you step through the actions in the service, you can load the appropriate sample documents to verify that the service can handle each instance.

For more information, see “Loading a Sample Document” on page 119.

14 Working with Registries

This chapter discusses the registry browsing functionality provided in the exteNd Composer. There are currently three different models in popular use for registry browsing: UDDI, ebXML Registry Services and WSIL. Composer supports all three of these specifications. They are compared briefly below, with reference made to sources of additional information.

The current business registry standard covering Web Services is UDDI (Universal Description, Discovery and Integration), which was designed to give businesses a uniform way to describe their services, discover other companies' services, and understand the methods necessary to conduct e-business in an automated or semi-automated way with remote partners. If you need to learn more about UDDI, the complete standard can be obtained at **http://www.uddi.org**.

In addition to UDDI, Composer supports ebXML Registry Services. ebXML stands for (Electronic Business using eXtensible Markup Language). The ebXML Registry and Repository, like UDDI, was developed to enable the storing and sharing of information between parties to allow e-business collaboration. Composer's implementation of ebXML is made possible using JAXR (Java XML Registries). The specification for ebXML can be found at: **http://www.ebxml.org/specs/#technical_specifications.**

Finally, Composer also supports WSIL (Web Services Inspection Language), yet another specification for the discovery and publishing of Web services. In the past few years since its inception, UDDI has been criticized for its lack of moderation and an inadequate quality of service (*Web Services Architect*, "WSIL: Do we need another Web Services Specification?"). WSIL was designed to be more lightweight and portable, and, in a sense, to pick up where UDDI leaves off. Although this standard has yet to be submitted to one of the standards bodies (W3C and OASIS) it is certainly widely-used and gaining in popularity. To find out more, see **http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html**.

UDDI, ebXML and WSIL form the basis for the registry management functionality described in the pages below. Familiarity with these standards, as well as a general understanding of the publishing and discovery of web services will be assumed here. Your web service may use one or all of these models, depending on the nature of the application you are developing.

Capabilities of the Registry Manager

The exteNd Composer incorporates a Registry Manager, accessible via the Registries tab at the bottom of the Composer main navigation frame. There is also a facility for defining registries through the Profiles capability (available via **Tools> Profiles...** in the Composer main menubar).

The capabilities of the Registry Manager include:

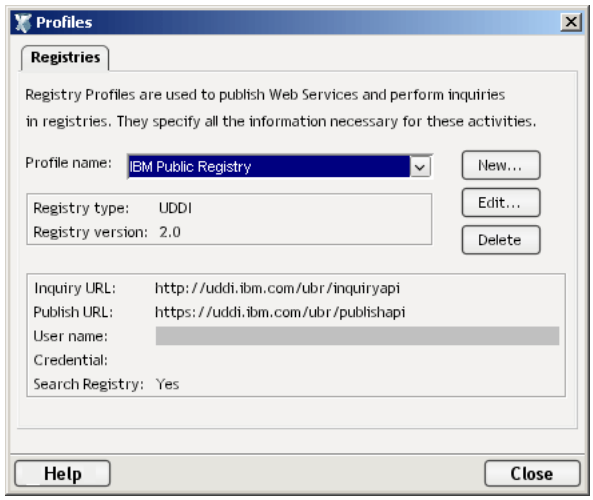
- Adding/removing registries
- Selecting registries to include in the search process
- Viewing business information on selected businesses in a given registry
- Viewing information on Web Services offered by a given business

- Searching for businesses or services within a registry or group of registries, optionally using extended query parameters
- Publishing new services to a registry

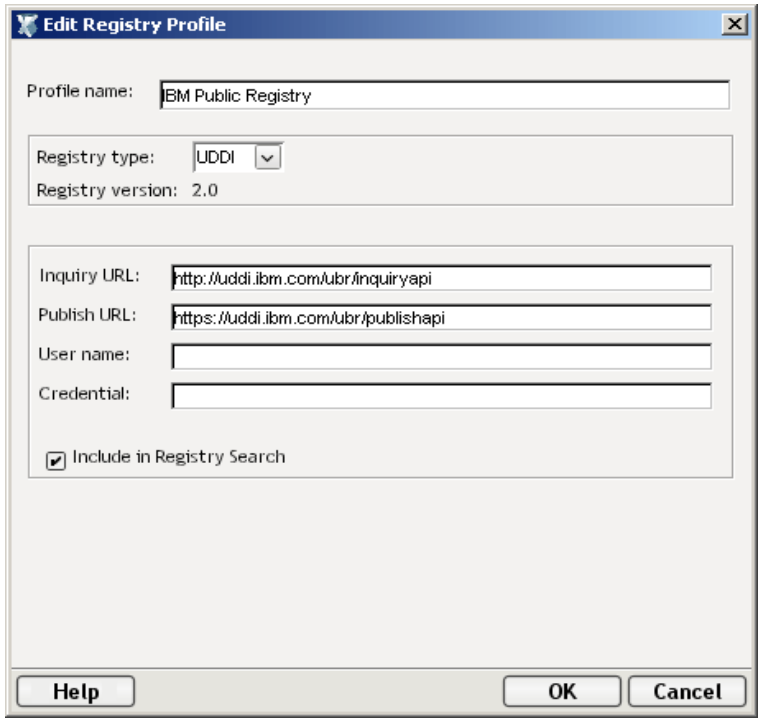
Registries are specified by URL and can be local or web-based. You can add or delete registries via the Profiles dialog (**Tools** menu, **Profiles**).

➢ **To edit or delete a registry:**

1 Select **Tools**> **Profiles . . .** from the exteNd Composer main menubar. The Profiles dialog appears.

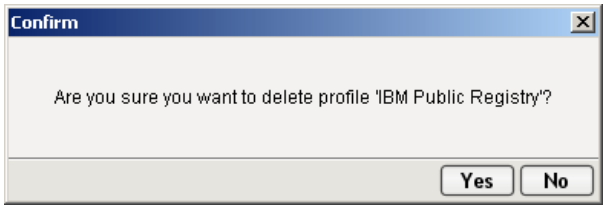


2 If you are editing an existing entry, select it from the Profile name pulldown menu, then click the **Edit** button. The Edit a Registry Profile dialog will appear, as shown below. After editing your selection, click on **OK** to save.



NOTE: If you have changed the name during editing, a new registry is created. If you do not want to keep the old one, then you must delete it.

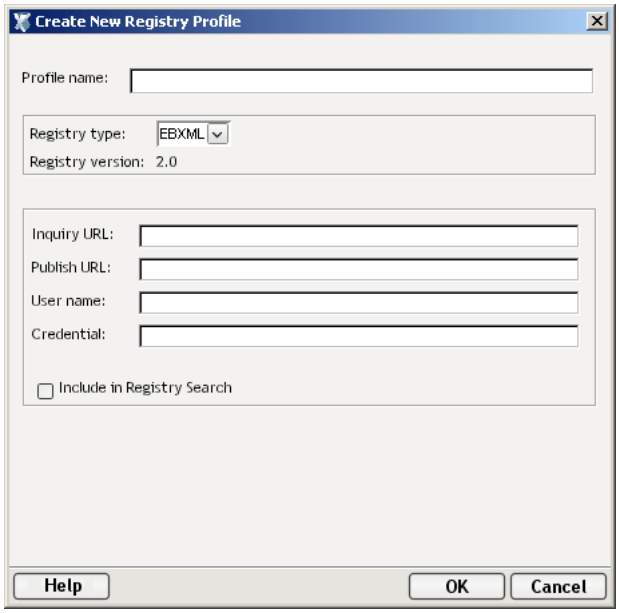
3 If you are deleting an existing entry, select it from the Profile name pulldown menu, then click the **Delete** button. A message will appear to confirm if your selection is the one you intend to delete. After deleting your selection, click on Close to save.



➢ **To define a new UDDI or ebXML registry:**

1 From the **Tools** menu, select **Profiles**.

2 In the Profiles dialog window, click on **New**. The Create a New Registry Profile dialog will appear.



3 Enter a name for the profile in the **Profile name** field (required).

4 Select the **Registry type** from the pulldown menu. The choices are: ebXML, UDDI and WSIL (which is described in a separate procedure, since the fields required for WSIL are unique). If you select ebXML or UDDI, the screen will look like the one above.

5 In the **Inquiry URL** field, enter the URL through which the registry can be queried (required).

6 In the **Publish URL** field, enter the URL via which new services can be published to the registry.

7 Enter the **User name** and **Credential** information, if any, that the registry provider assigned to you for publishing access.

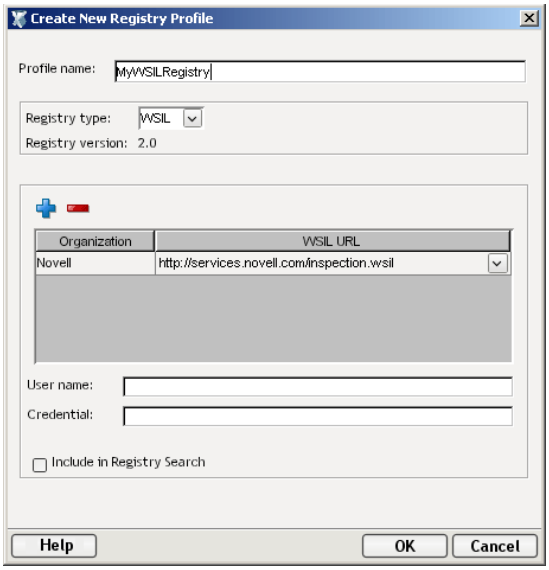
8 Check the **Include in Registry Search** checkbox if you wish to include this registry automatically in the default search set.

9 Click **OK** to close the dialog.

➢ **To define a new WSIL registry:**

1 From the **Tools** menu, select **Profiles**.

2 In the Profiles dialog window, click on **New**. The Create a New Registry Profile dialog will appear.



3 Type in a Profile name.

4 Select the **WSIL Registry type** from the pulldown menu.

5 Use the blue + icon to add additional WSIL registries.

6 Type in a name for the **Organization**.

7 Type in the fully qualified WSIL URL, ending in “inspection.wsil.”

8 To delete an organization, use the red - icon.

9 Enter the **User name** and **Credential** information, if any, that the registry provider assigned to you for publishing access.

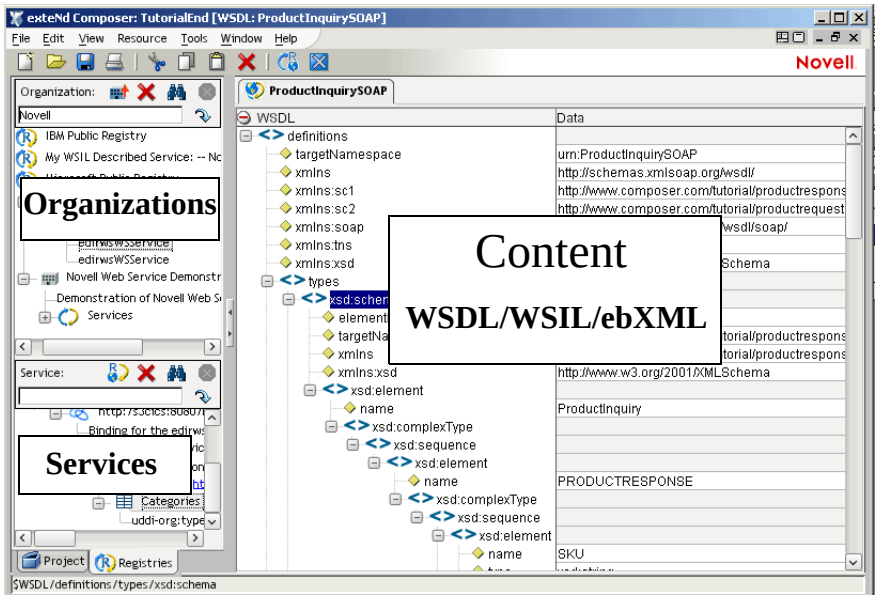
10 Check the **Include in Registry Search** checkbox if you wish to include this registry automatically in the default search set.

11 Click **OK** to close the dialog.

Once you have defined an ebXML, UDDI or WSIL Registry Profile in the above fashion, you will be able to use it in the Registry Browser tab on the Navigation Pane of the Composer main window. You can also publish services to the registry.

Registry Browsing

Registry browsing is available via the Registries tab in the Navigation Pane of the Composer main window. There are two subpanels within the navigation pane: one for organizations (top) and one for services (bottom). To the right is the Editor Pane. See illustration.

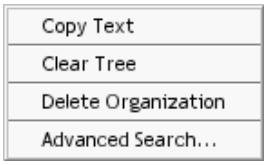


Context Menu Items

Context menus, specific to each pane in the Registry Manager, are available when using the Composer.

Organization Context Menu

To view the context menu for **Organization**, place your cursor in a field in the Organization pane and click the RMB (right mouse button). The context menu appears as shown.



The function of the context menu items are as follows:

Copy Text—Allows you to copy text from the currently selected business tree node to another area or file.

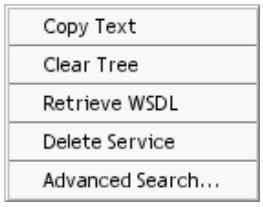
Clear Tree—Allows you to clear the pane of business information that you retrieved from your search.

Delete Organization—Allows you to delete the selected organization from the registry.

Advanced Search—Allows you to set advanced search criteria via the Set Browsing Criteria dialog.

Services Context Menu

To view the context menu for **Services**, place your cursor in a field in the service pane and click the RMB (right mouse button). The context menu appears as shown.



The function of the context menu items are as follows:

Copy Text—Allows you to copy text from the currently selected tree node to another area or file.

Clear Tree—Allows you to clear the service pane of information that you retrieved from your search.

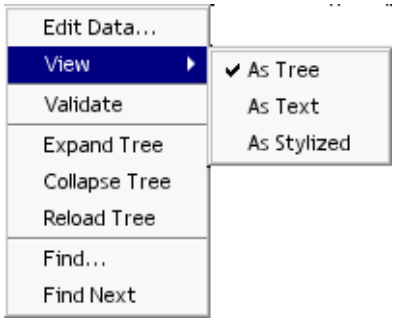
Retrieve WSDL—Allows you to retrieve the WSDL for the currently selected service from the registry. This can also be done via the Retrieve button. If the service you selected has no WSDL definition, a message will notify you of this condition.

Delete Service—Allows you to delete a service that you highlighted in either the business or service Registry.

Advanced Search—Allows you to set advanced search criteria via the Set Browsing Criteria dialog.

Content Pane Context Menu

To view the context menu for **Content pane**, place your cursor in a field in the pane and click the RMB (right mouse button). The context menu appears as shown.



The functions of the context menu items are as follows:

Edit Data—Allows you to change text from the information contained in the pane to another area or file.

View—There are three choices for viewing the information in the Content pane. They are: **Tree**, **Text** and **Stylized**. Click on your preference and the information will appear in the pane as such.

Validate—Runs a validation routine to ensure that your XML is sound.

Expand Tree—Displays all nodes in the pane.

Collapse Tree—Hides all nodes except the root node in the pane.

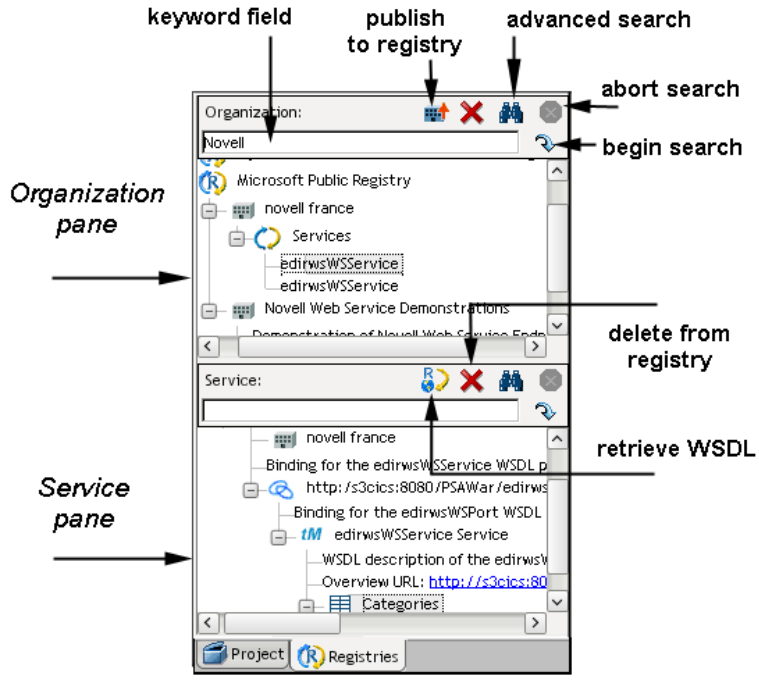
Reload Tree—Allows you to load the original tree.

Find—Allows you to search, via a dialog box, for a specific word or part of a word within the tree.

Find Next—Allows you to search, via a dialog box, for the next word or part of a word within the tree.

Action Buttons

The following illustration shows the location of the various action buttons on the Organization and Service panes.



Searching by organization

Searching for an organization (or organizations) is a simple matter of entering a complete or partial business name in the text field next to Organization, then clicking on the Search button (or “Go” button, shaped like a downturned arrow). A list of matching organizations will appear in tree-view form, in which each top-level node in the tree is a registry, each child of a registry is an organizationname, and underneath each business is detail information consisting of Descriptions, Categories, and Services. You can also enter a group of organization names separated by a vertical bar (pipe character), which allows you to search for multiple groups of businesses. For example, Silverton|Silicon etc.

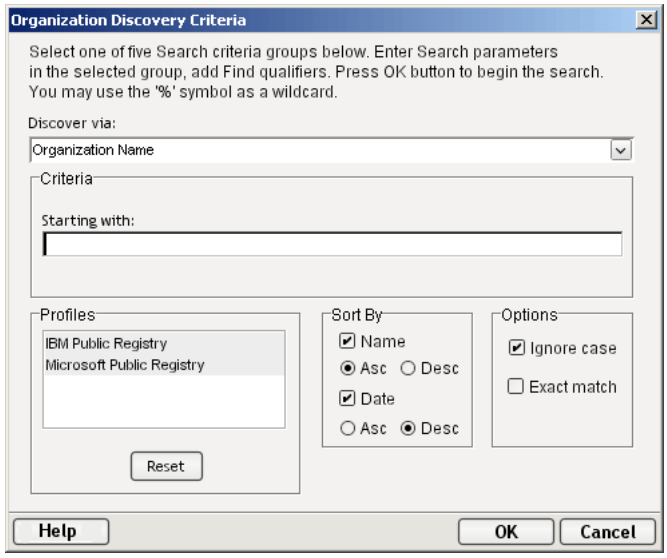
➢ To search organizations by keyword

1. To search on an organization name or partial name (or other keyword), enter text into the keyword field, then click the Go button (which looks like a downturned arrow). The search will begin. Note that while a search is underway, the Abort button (normally greyed out) is red.
2. Searches can take several minutes. If you want to interrupt a search prematurely, click the **Abort** button. Partial search results will show up in the Organization pane.
3. Wait until the search is complete. You will know it is complete when results have shown up in the Organization pane and the Abort button has returned to its normal, greyed-out (disabled) appearance.

➢ **To set advanced search criteria**

1 If you want to set advanced search criteria, do not enter anything in the text field; merely click the Advanced Search Button (shaped like binoculars).

The following dialog box appears.



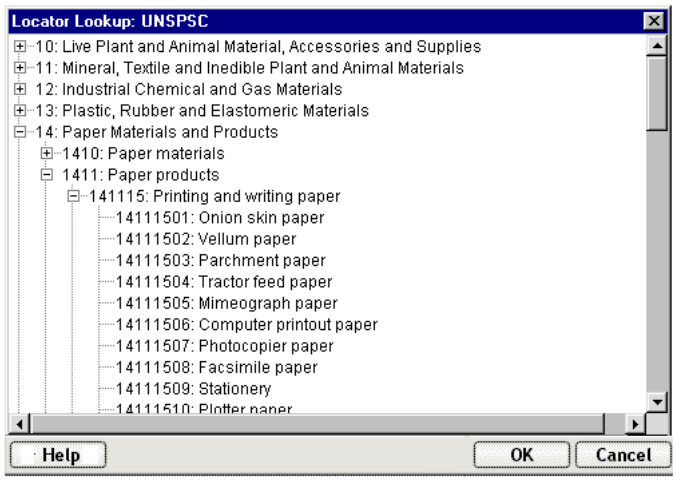
2 You can select only one of the search-criteria groups at a time. The available options are:

Organization Name: Enter enter a complete or partial organization name or list of names separated by a vertical bar (|) in the text field next to **Starting with**.

Identifier: If you choose this option, a new field called Identifier will appear. From the pulldown list, select one of the following: D-U-N-S, or Thomas Register (catalog names). Enter a key from the catalog (partial or complete) in the text field next to **Starting with**. This entry can contain numeric values and dashes.

Locator: When you choose this option, a new field called Locator appears. From the pulldown list select one of the following: NAICS (North American Industry Classification System), UNSPSC (United Nations Standard Products and Services Classification) or GEO (geographical). Enter a key from the catalog (partial or complete) in the text field next to Starting with, if you selected NAICS or UNSPSC. This entry can contain numeric values. Enter a country (region) abbreviation for GEO.

Alternatively, click the button at the far right of the control, to bring up a “key picker” in which you can doubleclick full or partial key names from a prepopulated list. See below.



Service Type Name: This allows the search of organizations associated with a particular UDDI *tModel*. Enter a key word for this *tModel* in the text field next to Starting with.

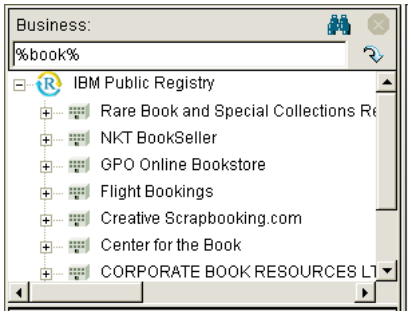
Discovery URL: Enter an IP address or portion of an IP address for the URL in the text field next to Starting with.

3 Select the **Registry Profile**(s) you want to use for this search. The Profiles box contains a list of Registries from which you can search. Registries you have previously selected in the Profiles dialog box (see description above) will already be highlighted. However, you may override them by selecting or de-selecting one or all of the registries within the list. If you decide to return to your original (default) registries, click the Reset button at the bottom of the dialog pane.

4 Under **Sort By**, you. can select how you want to sort—by Name, or by Date—in either Asc (Ascending) or Desc (Descending) order. The most common technique is to sort on Name (alphabetically) by ascending order or on Date (numerically) by descending order. Sorting by Date works within groups of businesses with identical names.

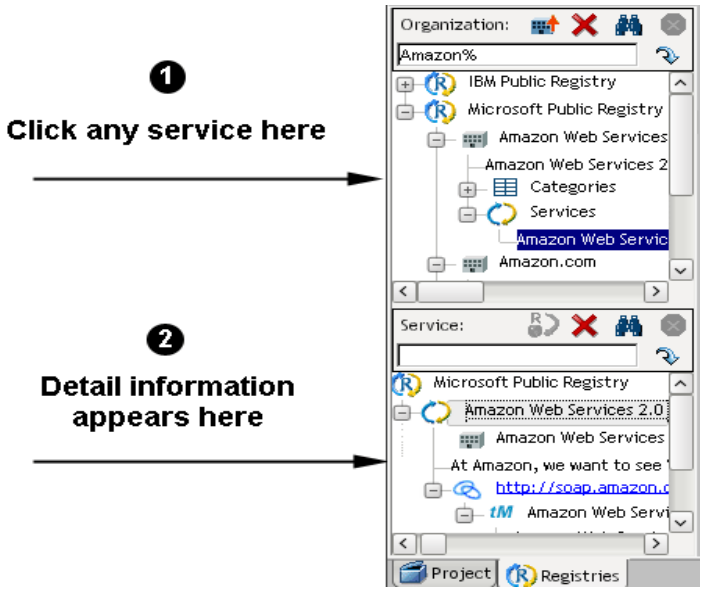
5 Under **Options**, you can select Ignore Case and/or Exact Match by clicking in the appropriate checkbox.

6 Click **OK**. The dialog goes away and your search begins.



After a search, a tree of matching businesses will be built in the Organization pane; the Service subpane will be cleared.

NOTE: Clicking a Service entry in the Organization tree causes that Service's detail information (binding, etc.) to appear in tree form in the lower Service pane. See below.



Searching by service

Searching for a service (or group of related services) is a matter of entering a complete or partial service name or keyword in the text field next to Service, then clicking on the Search button (or "Go" button, shaped like a downturned arrow). A list of matching services will appear in tree-view form, in which each top-level node in the tree is the registry that was searched; each immediate child of a registry is a service name; and children of the service node(s) contain detail information consisting of the Organization name associated with the service, a Description of the service, and bindings for the service.

Wildcards in Registry Searches

The Composer registry search engine supports the use of the percent sign (%) as a wildcard symbol, meaning one or more of any character. This is a particularly useful tool when you want to search for business or service names that *contain* a particular word but might not *start* with that word.

NOTE: The default search logic is "Start With." Thus a search on "Books" will turn up "BooksRUs" but not "ABC Booksellers" nor "Used Books". The way to override this behavior is to search instead on "%Books%", which will turn up all three.

The Composer registry search engine also supports the use of the | symbol as a logical-OR symbol, meaning "look for hits that contain any combination of these words." You can chain together any number of keywords this way. For example:

```
%Booking% | %Travel% | %Airline%
```

would return all names that contain *at least one* of the words, no matter where in the name that word might appear.

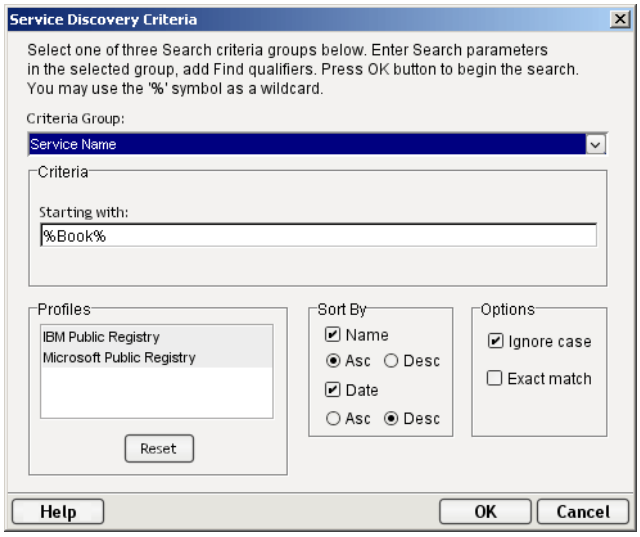
➢ **To search services by keyword**

1. To search on a service name or partial name (or other keyword), enter text into the keyword field, then click the **Go** button (which looks like a downturned arrow). The search will begin. Note that while a search is underway, the Abort button (normally greyed out) is red.
2. Searches can take several minutes. If you want to interrupt a search prematurely, click the **Abort** button. Partial search results will show up in the Service pane.
3. Wait until the search is complete. You will know it is complete when results have shown up in the Service pane and the Abort button has returned to its normal, greyed-out (disabled) appearance.

➢ **To set advanced search criteria**

1 If you want to set advanced search criteria, click the Advanced Search Button (shaped like binoculars).

The following dialog box appears.



2 As indicated by the presence of radio buttons, you can select only one of the search-criteria groups at a time. The available options are:

Service Name: Click on the radio button next to Service Name. Enter enter a keyword in the text field next to **Starting with**.

Locator: If you select this search criteria, a new pulldown menu appears from which you must select a Locator. You have the following choices: : NAICS (North American Industry Classification System), UDDITYPE, UNSPSC (United Nations Standard Products and Services Classification) or GEO (geographical). Enter a key from the catalog (partial or complete) in the text field next to Starting with, if you selected NAICS or UNSPSC. This entry can contain numeric values. Enter a country (region) abbreviation for GEO.

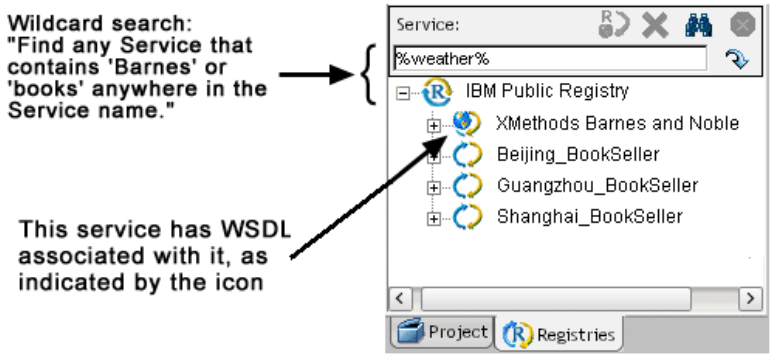
Service Type Name: Allows the search of businesses associated with a particular tModel. Enter a key word for this Model in the text field next to Starting with.

3 Select the **Registry Profile**(s) you want to use for this search. The Profiles box contains a list of Registries from which you can search. Registries you have previously selected in the Profiles dialog box (see description above) will already be highlighted. However, you may override them by selecting or de-selecting one or all of the registries within the list. If you decide to return to your original (default) registries, click the Reset button at the bottom of the dialog pane.

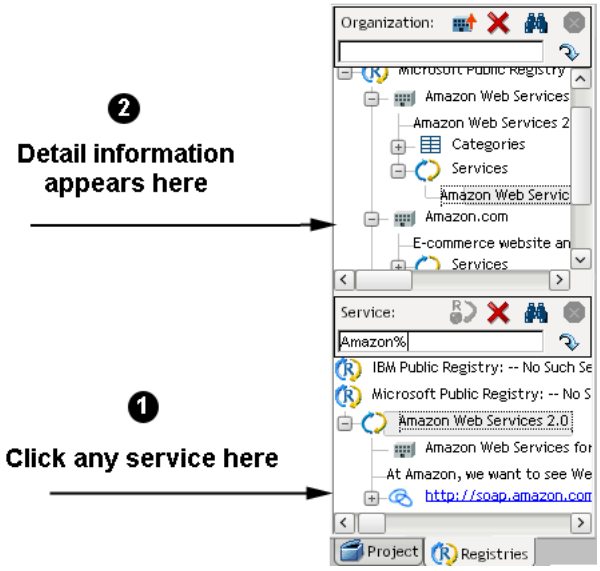
4 Under **Sort By**, you. can select how you want to sort—by Name, or by Date—in either Asc (Ascending) or Desc (Descending) order. The most common technique is to sort on Name (alphabetically) by ascending order or on Date (numerically) by descending order. Sorting by Date works within groups of businesses with identical names.

5 Under **Options**, you can select Ignore Case and/or Exact Match by clicking in the appropriate checkbox.

6 Click **OK**. The dialog goes away and your search begins.



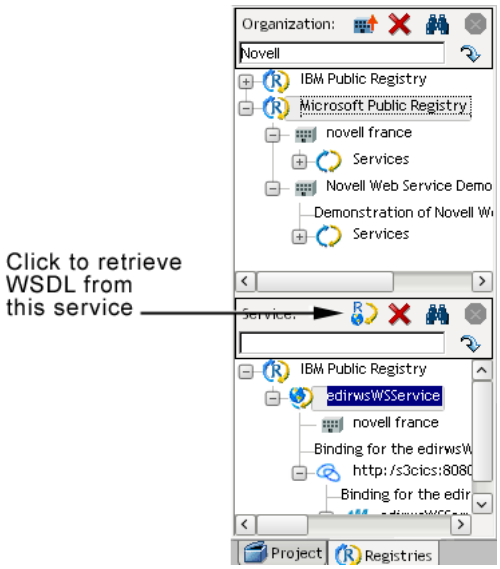
After a search, a tree of matching services is built in the Service pane; the Organization pane is cleared. Clicking a service node in the lower (Service) tree causes that business's detail information to appear in tree form in the upper (Organization) pane.



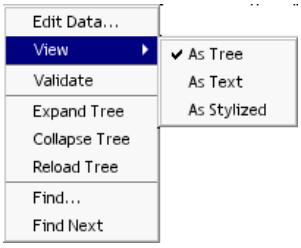
Retrieving WSDL from the Registry

After you have found the service that you searched for, now you can retrieve the WSDL definition for this service from the Registry. Just highlight the desired service node and click the Retrieve WSDL button, or click with the RMB and select Retrieve WSDL from the context menu. If a definition for the service exists, you see the Contents pane fill with the WSDL information in a tree format (see illustration). If no WSDL exists for the service, an alert dialog will appear, advising you of that fact.

NOTE: You can tell whether a given service listing has WSDL or not by looking at the service icon to its left. A ring icon with a globe in it means the service has WSDL. A ring icon with no globe means it is not a WSDL web service.



You can view the information in the Contents Pane as text or in stylized form by simply clicking on the RMB and selecting the view you wish to see.

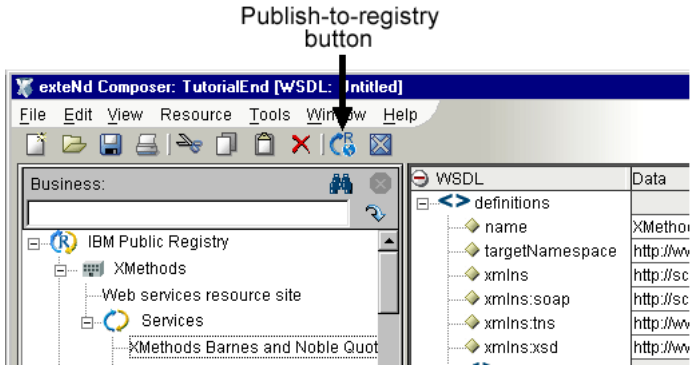


Publishing to a registry

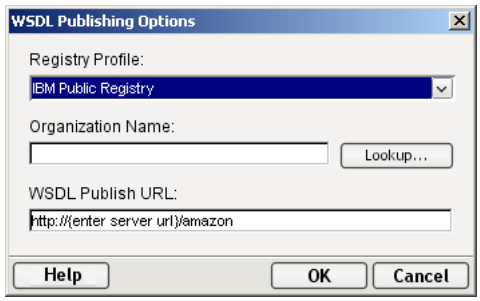
When you have created WSDL using the Composer editor, you can publish it to a registry by following the procedures outlined below.

➢ **To publish WSDL to a registry**

1 Click on the **Publish to Registry** button on the toolbar as shown below.

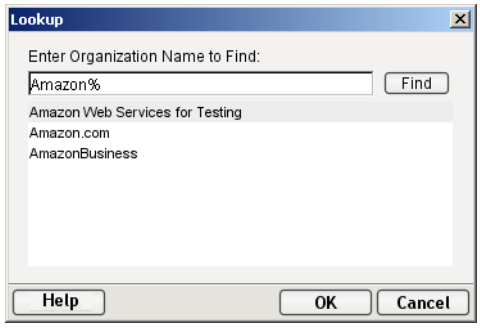


2 A dialog screen, WSDL Publishing Options, appears.



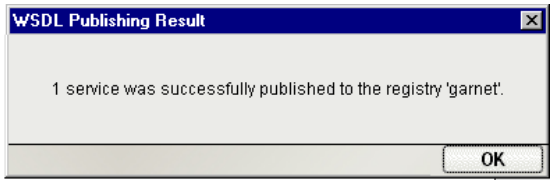
3 **Registry Profile**: Select the registry from the pulldown list you wish to publish.

Organization Name: Allows you to lookup organizations and select which one to associate the service with. If you click the Lookup button, the following dialog appears:



WSDL Publish URL: Shows the URL to which the service will be published to. You can edit this if you desire.

4 Click **OK** and if your service was successfully entered into the registry you selected, you will see a message like the one shown below.



15 Deploying Your Project

When you've completed the design, building, and testing of your project, the next step is to deploy it to the application server, where it will execute.

Some of the major topics discussed in this chapter include:

- Deploying Directly from Composer (Enterprise Edition only)
- Deployment from exteNd Director (all editions)
- Director Wizards for Composer Code Generation
- Server Profiles
- The Deployment xObject
- Defining SOAP Triggers (Composer UI for)
- Composer Web Service Wizard: SOAP Service Deployment (Director UI for)

Both editions of the Novell exteNd suite (Professional and Enterprise) offer tools for deploying Composer projects. In the Professional Edition suite, you will use exteNd Director to carry out your deployments. In the Enterprise Edition suite, you also have the option of deploying projects directly from Composer to any supported app server (Novell exteNd, WebSphere, WebLogic, Tomcat) without the need to switch between Composer and Director environments.

The sections immediately following this one will tell you about:

- Architectural considerations relevant to deployment
- How Composer handles packaging of EAR files and resources
- Composer's design-time UI for creating and managing deployment artifacts (applicable to the Enterprise Edition suite)
- Composer-related deployment aids and wizards available in Novell exteNd Director (applicable to all editions of the Novell exteNd suite)

App-server-specific issues, including administration issues pertinent to deployed Composer services, are discussed in a separate document: the *Composer Enterprise Server User's Guide*. Consult that guide if you have concerns about issues that are not addressed in this chapter.

NOTE: This discussion assumes some prior knowledge of J2EE deployment concepts, such as JAR (Java archive), WAR (Web archive), and EAR (Enterprise application archive) packaging, deployment descriptors, etc. If such concepts are unfamiliar to you, you may want to consult books or articles on J2EE deployment architecture before proceeding.

Planning your Deployment

Before deploying a Composer service, you should consider:

- **Packaging requirements**—Do you want to deploy services individually, straight from Composer, into the app-server environment; or will you instead be packaging whole projects (containing multiple services) into WAR or EAR files created in Novell exteNd Director (or possibly some other environment)?

- **Triggering needs**—How will your service(s) be fired off? The trigger object can be a servlet, an EJB, an EJB triggered through a servlet, a custom Java class that calls your service programmatically (directly), or a JSP that uses Composer tag library routines. If you have the JMS ConnecForm Resourcet installed, you can also fire a service from a (JMS) MessageListener object that "listens" for incoming requests on a queue. If you have the SAP Connect, you can trigger a service off an SAP function. (See the appropriate Connect user guides for more information.)
- **How arriving *data* might be packaged**—Will the incoming data be in the form of urlencoded param/value pairs appended to a URI (i.e., HTTP Get)? Will your service trigger handle incoming XML via HTTP Post with multi-part MIME attachment? Will your service be a SOAP service? Will the SOAP payload be encrypted or digitally signed?
- **Shared resources**—Do you want to deploy some resources, such as WSDL, JSP, JAR, XSL, or XSD files, as published resources so that other Composer services can share them?

These are just some of the considerations will affect how you deploy your Composer-created services. For some types of services, you will also want to consider connection pooling, container-based transaction management, directory storage of passwords and public key info, and perhaps other items as well.

About Service Triggers

A service trigger is a process (such as a servlet or bean) that initiates execution of a Composer service in response to some kind of input. The input *may* arrive via HTTP, but could also be an e-mail arriving via SMTP or a JMS message arriving at a queue. Various kinds of Composer triggers are available to handle various kinds of requests arriving via various transports. Composer can create triggers for you, or you can generate them using exteNd Director's code wizards. s

The following trigger types are supported by Composer:

- **E-mail**—A process on the server polls a mailbox at a specified interval and kicks off a Composer service when an e-mail meeting certain size limits arrives in the mailbox.
- **EJB**—The trigger is an Enterprise Java Bean (which responds to programmatic requests).
- **EJB with servlet**—The trigger is an EJB invoked by a servlet. The servlet "listens" on a URL, handles HTTP requests, and uses the bean to mediate interaction with a Composer service.
- **File**—A process on the server watches for the arrival of files in a particular location on a physical drive. When a file arrives, the Composer service fires.
- **JMS**—The trigger is a JMS listener. Arrival of a message at a topic node causes the Composer service to fire. (This trigger type is available only in the Novell exteNd Enterprise Edition version of Composer.)
- **JSP**—The trigger is a Java Server Page containing scriplets that can call a Composer service directly.
- **SAP Service**—Execution of an SAP function causes a Composer service to fire.
- **Servlet**—Arrival of a request over HTTP causes the trigger to invoke a Composer service.
- **SOAP HTTP**—Arrival of a SOAP request causes the service to fire.
- **Timer**—A daemon process on the server causes a Composer service to fire at a set interval.

In addition to these "event-oriented" trigger types, it is possible for a custom Java class to invoke a Composer service directly. Novell exteNd Director has wizards, in fact, that can generate the necessary code. (See "Director JSP Wizard" and "Java Class Wizard" below.)

Triggers and Input Data

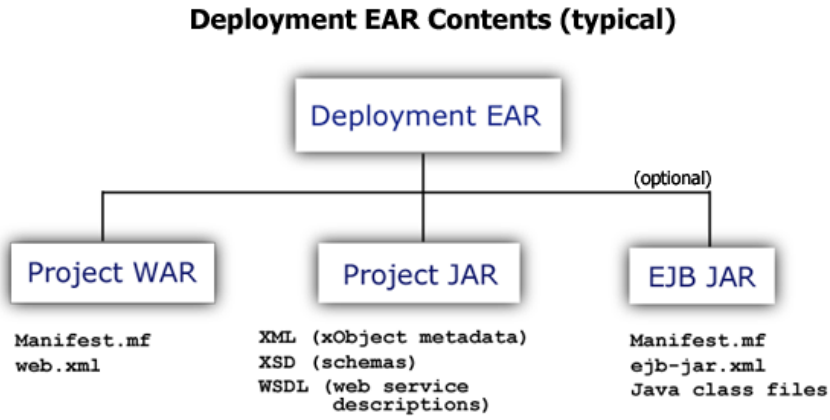
In a contractual sense, triggers have two responsibilities: They not only listen for specific events (whether it's the arrival of an HTTP request, arrival of e-mail, etc.) and take action on them; but they also must harvest and "hand off" XML data to the Composer service. To do the hand-off, the trigger has to know how to *collect* the data over the transport in question and *package* it in a form Composer can understand. Various helper classes (included in the Composer Server installation) are available to help a trigger servlet marshall/unmarshall data appropriately. (Those classes are described in more detail in the discussion at "Converter Classes" in the appendix on JSP tag-library methods.)

For example, a servlet can acquire data as part of an HTTP GET, or as part of HTTP POST. In the simplest case, user data consists of name/value pairs attached to the end of a URL (i.e., a request arriving by HTTP GET). But data can arrive in other ways as well, such as a SOAP request; XML via HTTP POST where XML is embedded in a form field; XML via HTTP POST with XML constituting the entire content portion of the stream; or XML via HTTP POST as multipart-mime attachments. Composer will also let you define triggers that fire when e-mail is received. In that special case, XML arrives in the form of an e-mail attachment.

In the discussions that follow, familiarity with the basic trigger types available in Composer is assumed. If you are not already familiar with Composer's trigger architecture, consult the Composer Enterprise Server User's Guide.

About Composer-Built Deployment EARs

At deployment time, Composer packages all of the deployable services in a project into an *EAR* (Enterprise application archive). This is the so-called "deployment EAR." Its contents are as shown below.



When you deploy directly from Composer, the following steps happen automatically (in the order shown):

1. Composer packages your deployable resources—including the metadata (XML) describing your services—into a Project JAR. This JAR, along with the files mentioned below, is written to a *staging directory* (typically a subdirectory in your project directory).
2. Composer creates a WAR file containing two items: a manifest that points to the foregoing JAR, and a **web-xml** file that describes all of the trigger servlets that apply to your deployed services (as well as URL bindings for them), so that the app server knows how to find and expose your services.
3. The JAR and WAR files are packaged into a deployable EAR.
4. The EAR is uploaded from the staging area to your app server.

The last step varies in implementation depending on the type (and version) of app server to which you are deploying. In some cases, Composer will create and execute a batch file that carries out the steps needed to put the EAR (and any deployment descriptors) on the app server. In other cases, Composer Enterprise Server will "pull" the deployment EAR onto the server. In all cases, Composer will (as part of Step 4) launch your web browser and lead you through a short series of forms, where you will provide any user-ID, password, or other information the server may need in order for the deployment to finish normally.

It may or may not be necessary to restart the server after deploying the EAR (this varies by app server; consult your app server's documentation).

NOTE: You should ensure that the app server *and* Composer Enterprise Server are running before undertaking any kind of Composer deployment.

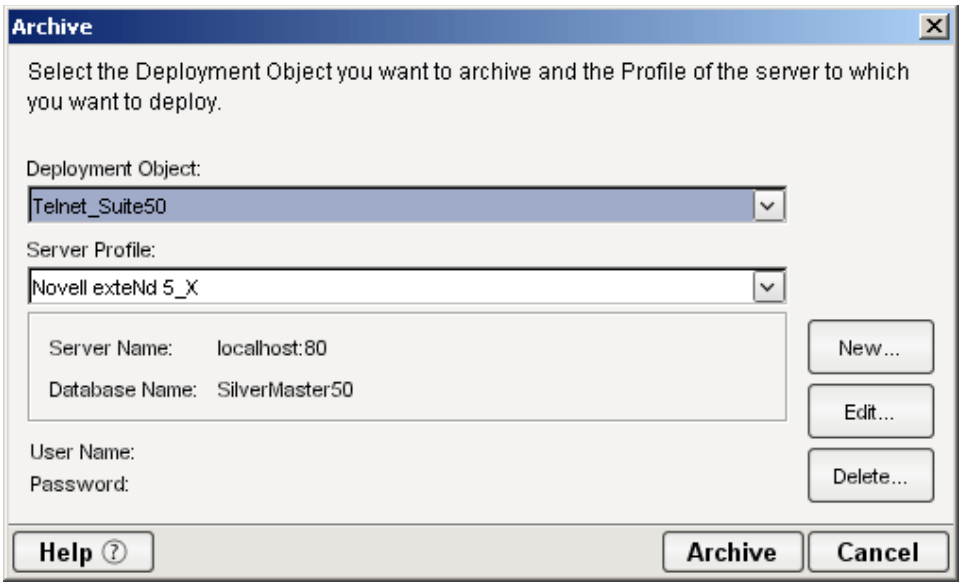
Creating EAR, WAR, and JAR Archives

In Composer 5.2, you have the option of letting Composer package your project into EAR, WAR, and JAR archives and write them to disk. You can later work with these archives in a J2EE tool of your choice or install the files manually into an app server environment of your choice, etc.

NOTE: Deployment objects (in the discussion below) are supported only in the Enterprise Edition of the product.

➢ **To create EAR, WAR, and JAR archives from within Composer:**

1 Using the **File** menu, select the **Archive Project** command. A dialog appears.



2 Use the **Deployment Object** dropdown menu to select the Deployment Object applicable to the application you wish to archive.

3 Choose the **Server Profile** that corresponds to the intended deployment environment. (Use the **New, Edit**, or **Delete** buttons as necessary.)

4 Click the **Archive** button to initiate the archiving process, or **Cancel** to abort.

When Composer has finished creating archives, you will find them under the staging folder that you previously specified when creating your deployment object. (To see this preference, open the Deployment Object in question and use **File > Properties** to view its setup params.)

Deployment Options

When it comes to building and deploying the deployment EAR, there are several options.

The first option is to use Composer's native design-time UI to package and install deployable objects for a given project. (This requires the standalone version of Composer, *or* the exteNd Suite Enterprise Edition. It is not an option in the Professional Edition suite.) The "Composer-direct" method is the easiest and quickest way to deploy Composer-built web applications.

The second option is to use exteNd Director's utility-tools UI for building and/or customizing a deployment EAR. This option should be considered when your Composer project needs to be bundled into an EAR with other Java objects (such as portal components) as part of a large deployment.

A third option is to build JARs, WARs, and/or EARs manually (or with a third-party tool) and install your deployment objects "by hand," following the procedures recommended by your app-server vendor. This option should be considered only when you need low-level control over the deployment process for one reason or another.

We'll discuss the first two options in depth, beginning with Composer-direct deployment.

Deploying Directly from Composer

If you are using Composer Enterprise Edition, you can deploy projects directly from Composer. (The Professional Edition does not support this capability. If you will be using Professional Edition, you should skip this section and proceed directly to "Deployment from exteNd Director" further below.)

The basic steps involved in deploying your project from the Composer design-time environment are:

1. Create a Server Profile
2. Create a Deployment Object
3. Set up Service Triggers and Resources for your Web Service
4. Prepare Objects for Deployment
5. Deploy the EAR to the server

Server Profiles

Server profiles define a target server and the necessary server-specific information required for deplyoment to that server. Creation of a server profile is a necessary prerequisite for deployment of a Composer project to the app server (regardless of app server type: Tomcat, Novell exteNd App Server, WebSphere, or WebLogic).

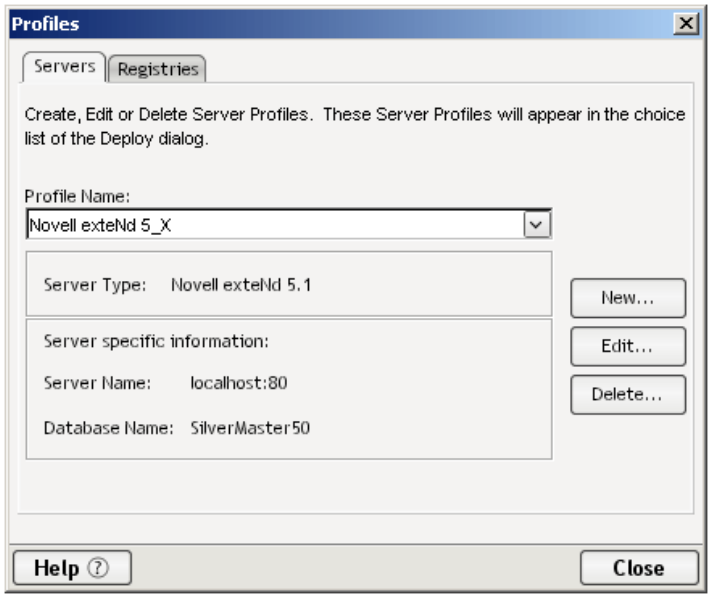
NOTE: Server profiles are not project-level resources. They are stored in a properties file and are available for use with *all* projects you create using a given installation of Director or Composer.

Before creating your server profile, you should make sure that the application server to which you will be deploying is running and has the exteNd Composer Enterprise Server (and any necessary Connects) installed and running as well.

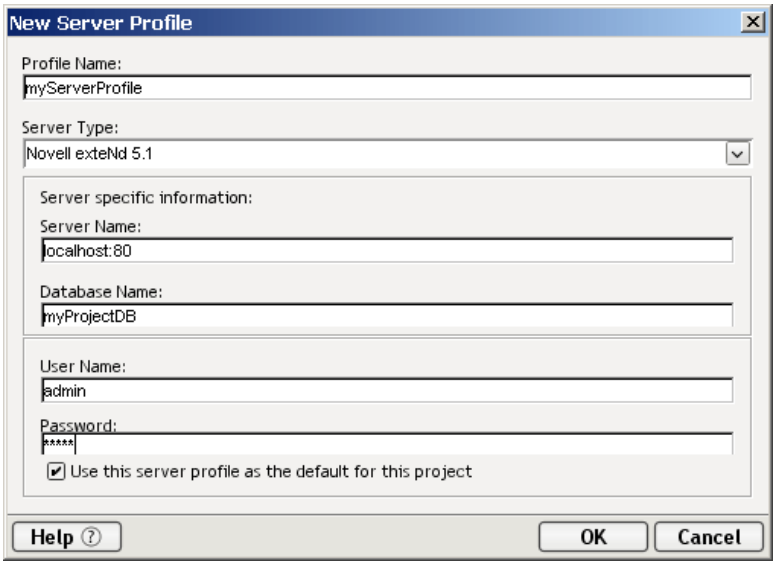
➢ **To create a Server Profile**

1 Select **Tools>Profiles** from the Composer menu.

2 From the Profiles dialog, select the **Servers** tab.



3 Click on **New...** to create the new Server profile. A dialog appears:



4 Specify a **Profile Name**. The name you type will be used to identify this particular profile when you deploy your project.

5 Select a **Server Type** from the drop-down list. The choices are:

- WebSphere 4.0 and 5.0
- Tomcat 4.1
- WebLogic 8.1
- WebLogic 6.1
- Novell exteNd 4.0 or 5.2

NOTE: These were the choices as of the exteNd 5.2 release timeframe. Consult the Novell website for the latest upgrade and patch information.

The fields in the “Server Specification Information” area of the dialog will change according to which Server Type you specify in this field.

6 In all cases, you will need to provide a **Server Name**. In the example above, `localhost:80` was entered, since this refers to a locally installed exteNd Application Server.

- If you selected a WebSphere or WebLogic Server, you will also need to identify **Target Servers**.
- If you select an exteNd Application Server, you will also need to specify a database name to be used for deployment.
- For Tomcat Servers, no other Server Specification information is required.

7 Enter a **User Name**and **Password** if your server requires authentication.

8 Click the checkbox if you would like to use this Server Profile as your default.

9 Click **OK** to create the new server profile.

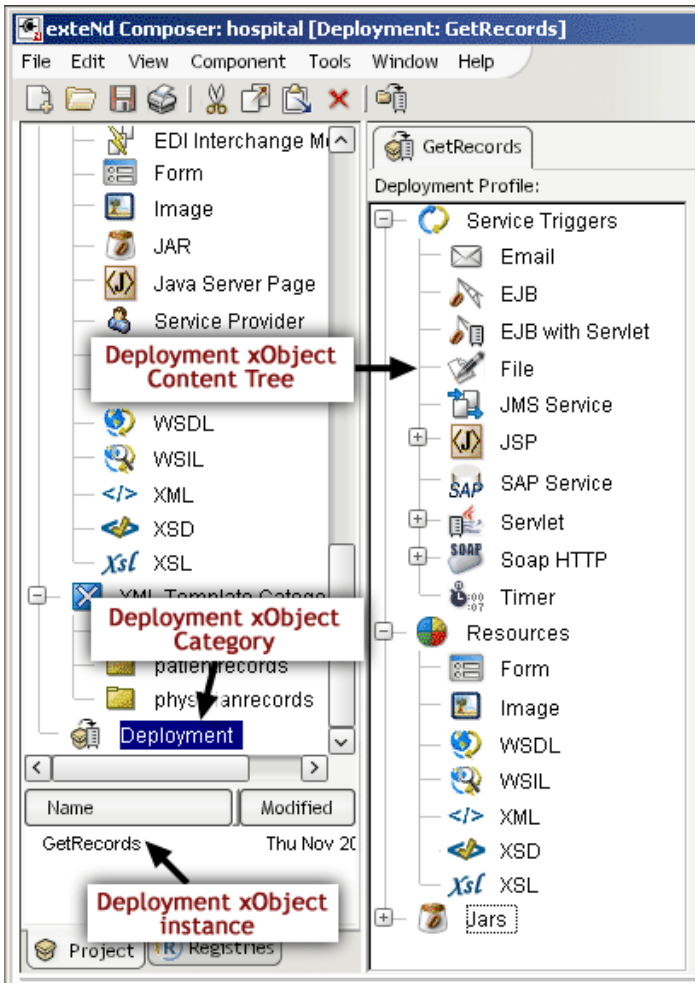
NOTE: To modify existing server profiles, go to **Tools > Profiles** and select the **Servers** tab. Select the profile you wish to change and click on **Edit**. Similarly, if you wish to delete a Server profile, click on the **Delete** button.

The Deployment xObject

Deployment objects, like Services, Components and Resources, are another species of Composer xObject. They contain metadata about your deployment: information about which service triggers to create and which resources to deploy.

NOTE: This discussion does not apply to the Professional Edition suite. The Deployment xObject is available in Composer only in the Enterprise Edition suite. If you are using the Professional Edition, you will create and manage deployment artifacts in the Director environment. See “Deployment from exteNd Director” further below.

Like other xObjects, Deployment objects are accessed through Composer’s Navigator Pane.

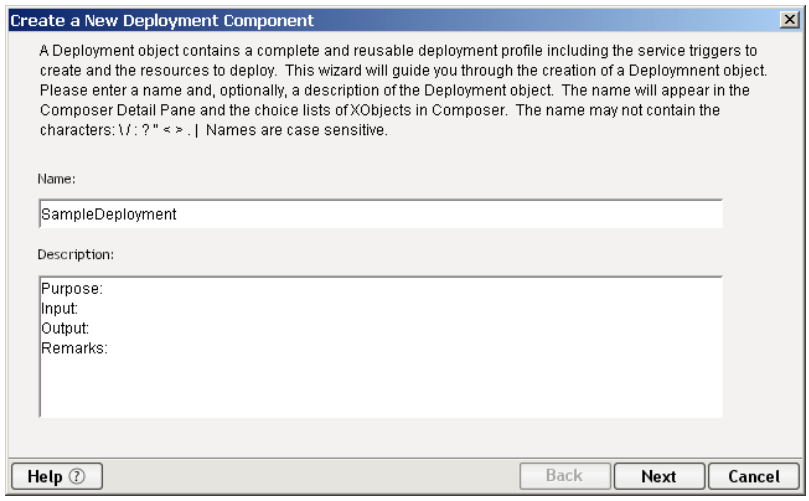


The procedure for creating a Deployment xObject will probably feel familiar to you, since it involves a wizard that operates much like Composer's other xObject-creation wizards.

➢ **To create a Deployment Object:**

1 Click with your right-mouse button on Deployment in the Navigator Tree and select **New**. (Alternatively, use **File > New > xObject**, and select **Deployment** on the Component panel.)

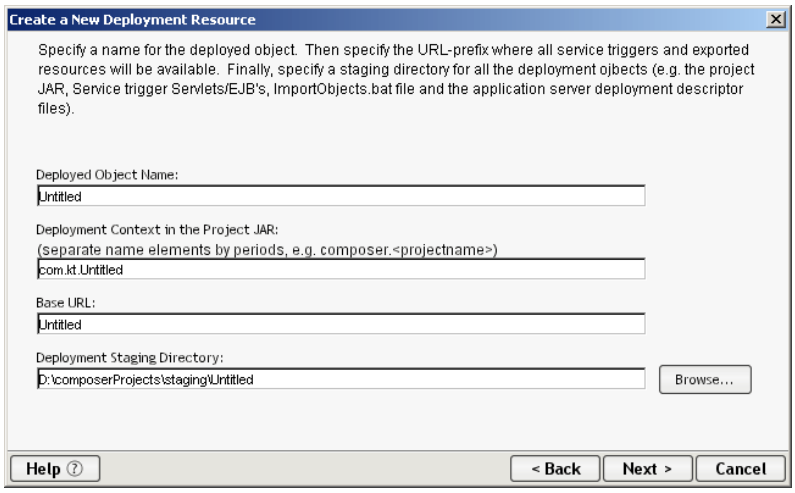
2 The first screen of the New Deployment Wizard appears:



3 Provide a **Name** for the Deployment object.

4 Optionally, provide a **Description**.

5 Click **Next** to proceed to the next screen:

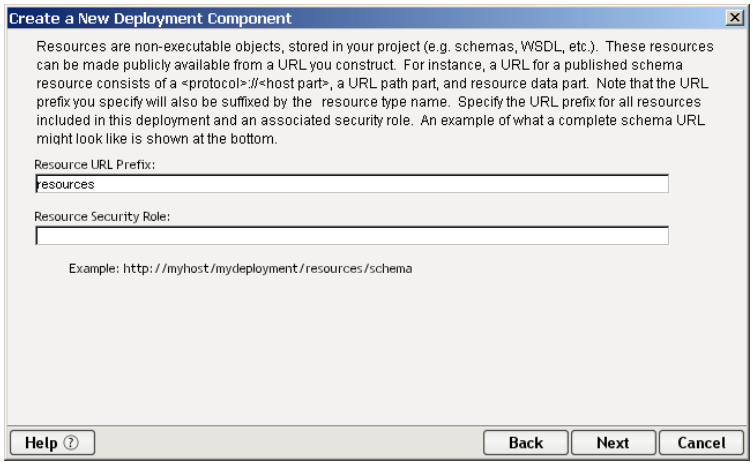


6 Specify a **Deployed Object Name** (this will default to the name you entered on the previous screen).

7 Specify a **Base URL** by entering the URL-prefix where your service triggers and other resources (JSPs, images, etc) will be available.

8 Finally, browse your file server to designate a **Staging Directory** to hold your deployment objects and descriptor files.

9 Click **Next** to proceed to the next screen:

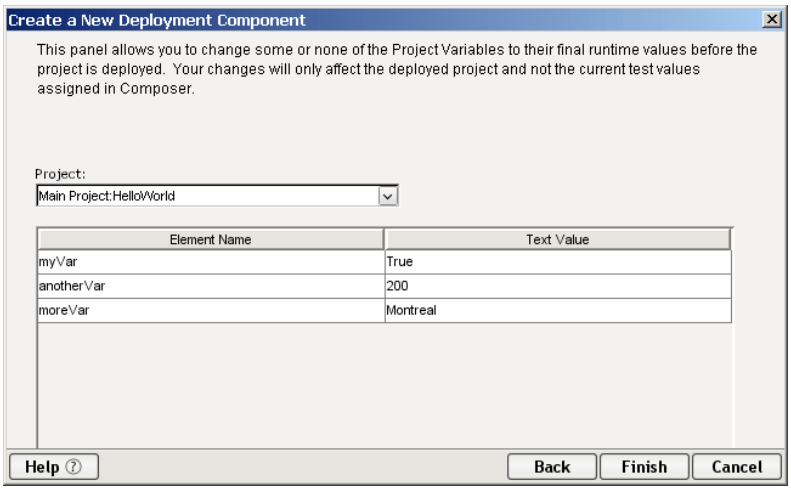


10 Specify a **Resource URL Prefix** to be used to access any resources used by your project which you would like to be publicly available.

11 You can also designate a **Resource Security Role**, if you are using J2EE 1.3 (or higher), to prevent users from surreptitiously accessing resources

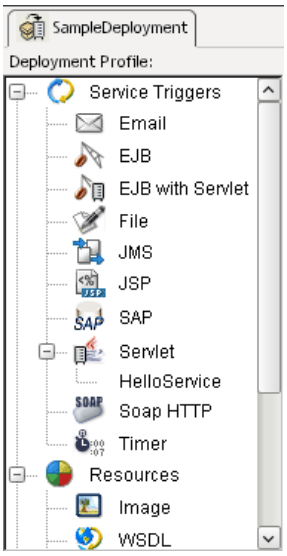
NOTE: The security role is ignored for Composer Services in the project that use the resource.

12 Click **Next** to proceed to the final screen of the Deployment Object creation wizard:



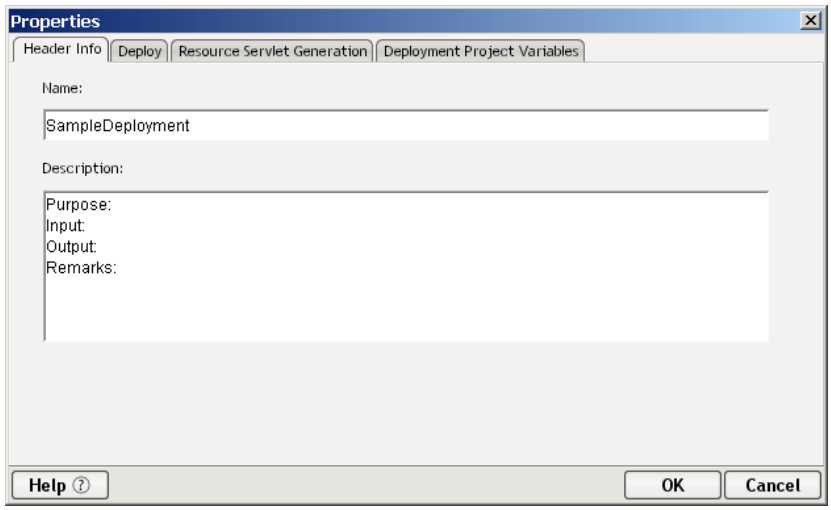
13 This panel provides you with an opportunity to override project variables that exist in the project (and any subprojects called by it) and set them to the values they will need at runtime. The table is pre-populated with the project variables and their values currently defined in the project.

14 Click **Finish** to create the deployment object and have it appear in your work area in the form of a Deployment Content Tree, as shown below:



Editing Existing Deployment Object Properties

Like all other xObjects, once a Deployment object is created, you can access a tabbed dialog containing all the wizard panels by selecting a Deployment object and using the RMB to select **Properties**. Values can be modified, as necessary, using the Properties dialog. An example of the tabbed interface is shown below.



Configuring a Deployment

The discussions below describe how to set up and carry out various kinds of deployment operations from the Composer design-time environment using standalone Composer *or* the Enterprise-Edition suite (not Professional Edition).

NOTE: If you are using the exteNd Suite Professional Edition, you should skip the following discussion and go directly to the discussion at *"Deployment from exteNd Director"* further below.

IMPORTANT: Before attempting any of the following procedures, you should already have created an applicable *server profile* (see "Server Profiles" above), as well as a Deployment Object (see discussion at "The Deployment xObject") to contain this project's deployment contents. *You should also have already created (and added to your Deployment Object) any special resources needed by your service(s), such as WSDL resources for SOAP services.*

Service Triggers

One of the most important configuration choices regarding the deployment of services is deciding what kind of *triggering mechanism* to associate with the service. Composer can create many different kinds of trigger objects. All you do is decide which kind(s) of triggers to associate with which individual services, and specify a few parameters appropriate to each trigger.

Note that you can associate more than one trigger type with a given service. You can also associate more than one service with a given trigger type.

In the sections to follow, you'll learn about:

- Defining EJB-Based Triggers (including EJB-with-Servlet)
- Defining E-mail Triggers
- Defining File-Based Triggers
- Defining JSP-Based Triggers
- Defining Servlet-Based Service Triggers
- Defining SOAP Triggers
- Defining Timer-Based Service Triggers

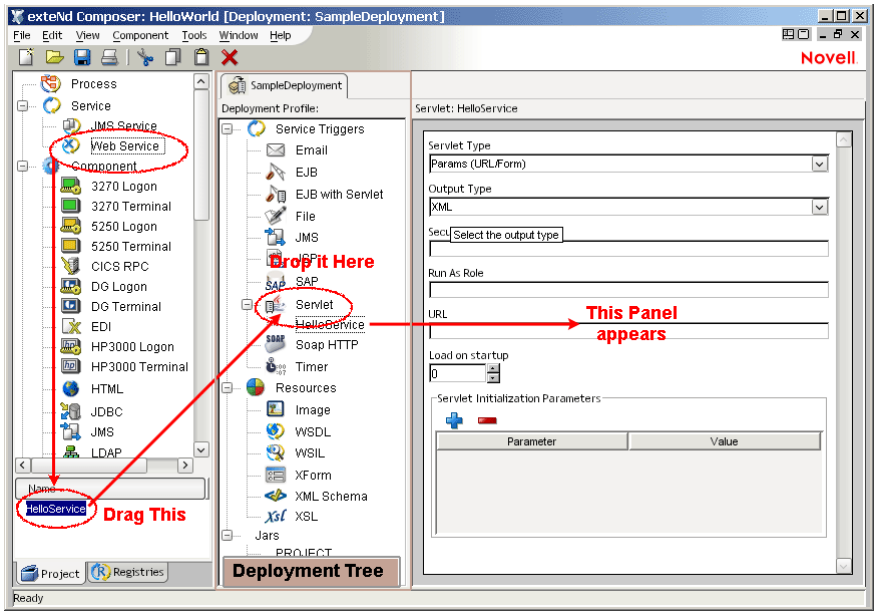
NOTE: For information on how to set up JMS-based or SAP-gateway-based service triggers, see the separate *JMS Connect User's Guide* or the *SAP Connect User's Guide*, as appropriate.

Drag-and-Drop Creation of Service Triggers

Most of the procedures described in this chapter use the *drag-and-drop* GUI metaphor extensively. Drag-and-drop affords an easy, quick method of associating triggers with services. But it should be noted that the same associations can also be created using *menu commands* instead of drag-and-drop. (See next section.)

The drag-and-drop procedure is easy:

- Open a Deployment object
- Select the Web Service category in Composer's navigator
- Drag a particular service instance over to the Deployment tree and drop it on the trigger type of interest



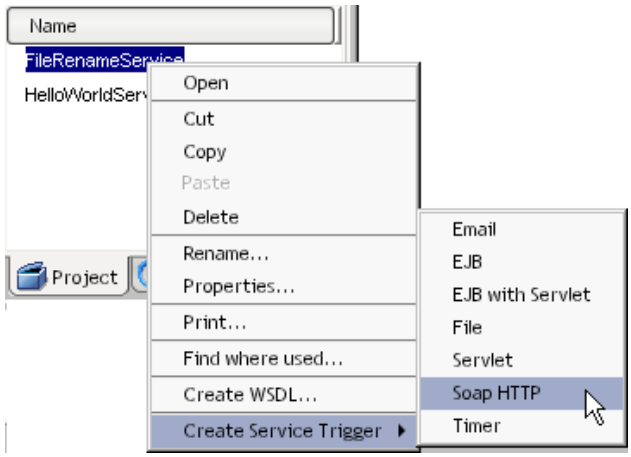
Creating Service Triggers Using Menu Commands

To create a Service Trigger, simply right-click on the object in the Navigator pane. There are caveats, however. In order to create a Service Trigger, the following two conditions must be met:

- A deployment xObject must be open and active in the native environment panel editor.
- The object that was right-clicked on must actually be able to be used to create a service trigger.

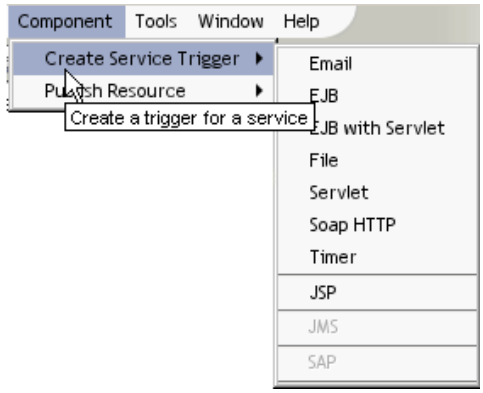
If these conditions are met, then the items listed in the sub-menu will be the service triggers that can be created using the object that was right-clicked on.

NOTE: A service must have an associated WSDL resource to be deployable as SOAP HTTP and thus for the menu item to appear.



Clicking on one of these items will create the appropriate entry in the deployment tree pane (as if the user had dragged it there manually) and cause its property sheet to appear in the deployment properties pane.

You can also create a Service Trigger using the **Create Service Trigger** submenu in the **Component** menu (in Composer's main menubar), as illustrated below.



When you make a selection from one of the trigger options shown in the submenu, a small dialog appears:

Use the pulldown menu control to select the service to which the trigger will be bound. Then click OK.

NOTE: The JMS and SAP service-trigger options are not available unless the relevant Connect products are installed.

Defining E-mail Triggers

You can configure a Composer service to fire when an e-mail arrives in a particular mailbox. The e-mail becomes the payload, such that if you were (for example) to write `Input.getXML()` to `System.out`, using a Function Action, you would see the entire message appear in your system console, in XML format, similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<Message>
  <X-Auth-OK>joeblow@smtp-send.myrealbox.com</X-Auth-OK>
  <Return-Path>&lt;jblow@myrealbox.com&gt;</Return-Path>
  <Received>from JBLOW-DT1 jblow@smtp-send.myrealbox.com [12.23.52.5]
        by smtp-send.myrealbox.com with NetMail SMTP Agent $Revision:   3.42  $
on Novell NetWare;
        Mon, 29 Sep 2003 13:09:23 -0600</Received>
  <Message-ID>&lt;13140405.1064862444476.JavaMail.JBlow@JBLOW-DT1&gt;
</Message-ID>
  <Date>Mon, 29 Sep 2003 15:07:24 -0400 (EDT)</Date>
  <From>joeblow@myrealbox.com</From>
  <To>joeblow@myrealbox.com</To>
  <Subject>Trigger Test Mail</Subject>
  <Mime-Version>1.0</Mime-Version>
  <Content-Type>text/plain; charset=ASCII</Content-Type>
  <Content-Transfer-Encoding>7bit</Content-Transfer-Encoding>
  <Body charset="ASCII" encoding="7bit" subtype="plain" type="text">This is a test
message
</Body>
</Message>
```

Before creating an e-mail trigger, you should know:

- The IP address of the target mail server (e.g., **pop3.myrealbox.com**)
- The protocol (POP3 or IMAP)
- The name of the mailbox (typically INBOX)
- The account-holder's name (such as the "myname" in *myname@mydomain.com*) and any associated password

The account name and password info should exist in the form of a *Mail Simple Authentication* connection resource. (Instructions for creating this type of connection resource can be found under "Mail Simple Authentication Connection Resource" in the discussion of Connection Resources.) If you have not created a resource of this kind to hold your account info, do so before using the procedure shown below.

You should also have some idea of how often you would like the target mailbox to be checked, and whether or not e-mails larger than a certain size should be ignored.

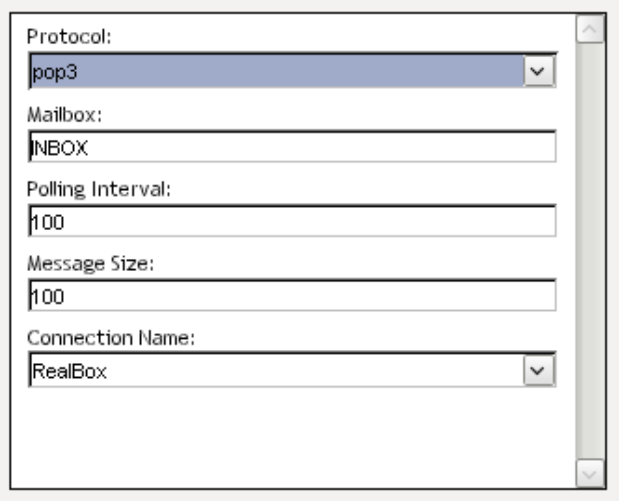
Note that as e-mails are detected and processed, they are consumed (removed) from the mailbox. Any e-mails that are detected but not processed (due to size constraints—see below) will be left in the box unread.

➢ **To associate an E-mail Trigger with a service:**

1 In the Navigation Category pane, under Services, click on **Web Services**.

2 Find the service you wish to deploy in the instance pane and highlight it by clicking.

3 Drag the service onto the **E-mail** node of the Deployment tree, under Service Triggers. When you let go of the mouse, a property sheet similar to this one should appear:



4 Select IMAP or POP3 from the **Protocol** menu.

5 Enter the name of the **Mailbox** (typically INBOX).

6 Under **Polling Interval**, enter a numeric value representing the time, in *seconds*, to wait between checks of the mailbox.

NOTE: If the box contains e-mail and the trigger fires, Composer waits until the service finishes executing before checking the box again. So in other words, if the service should happen to take two seconds to execute, and the polling interval is 10 seconds, it might take as long as 12 seconds for the box to be checked again after the previous e-mail has been detected.

7 Under **Message Size**, enter a numeric value representing the size, in *kilobytes*, of the largest e-mail that will be processed. Any e-mail that is smaller in size than this number will be processed: the service will fire and the mail will be passed to it (and removed from the mailbox in the process). Any e-mail larger in size than this value will simply be ignored.

8 **Save** your Deployment object.

Defining EJB-Based Triggers

Associating an EJB trigger with a service can be done in two slightly different ways. One way is simply to designate an EJB (session bean) as the object through which service access occurs. In this case, there is no URL to hit: instantiation of the EJB has to occur *programmatically* (perhaps through a custom trigger object of your own design). The only deployment parameters you can set in this case are the JNDI name, session type, and transaction attribute. Your service, after deployment, becomes available through normal JNDI/EJB mechanisms, but otherwise has no "web-facing" layer.

The other way of deploying a service to the EJB container is to choose the "EJB with Servlet" trigger option and let Composer produce the EJB as well as a servlet that knows how to access the EJB. In this case, Composer "front-ends" your EJB-based service with a web-tier component (a servlet) that can act on HTTP requests.

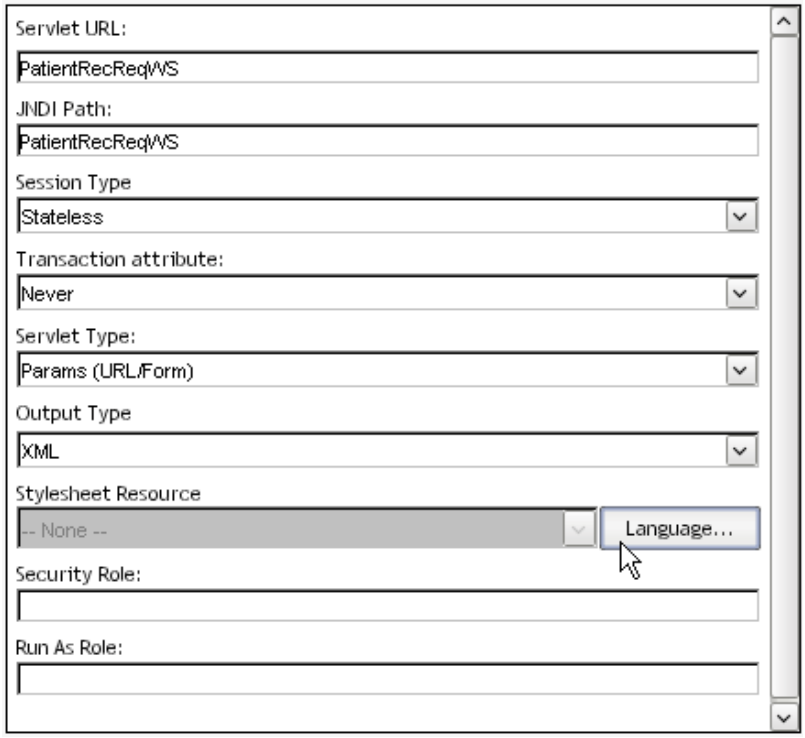
➢ **To associate an EJB (or EJB-and-Servlet) with a service:**

1 In the Navigation Category pane, under Services, click on **Web Services**.

2 Find the service you wish to deploy in the instance pane and highlight it by clicking.

3 Drag the service onto the **EJB** or **EJB with Servlet** node of the Deployment Profile tree, under Service Triggers.

- If you drop the service onto the **EJB** trigger node, you will get a property sheet that has fields only for three items: *JNDI Path*, *Session Type*, and *Transaction Attribute*. See explanations below.
- If you drop the service onto the **EJB with Servlet** category node, the property sheet that appears will have the aforementioned three items as well as several more:



4 Specify a **Servlet URL**, as applicable.

5 Fill in the **JNDI Path** that will be used to find the object.

6 (Enterprise Edition only) For **Transaction Attribute**, select the applicable JTA transaction behavior from the dropdown list. Choices include:

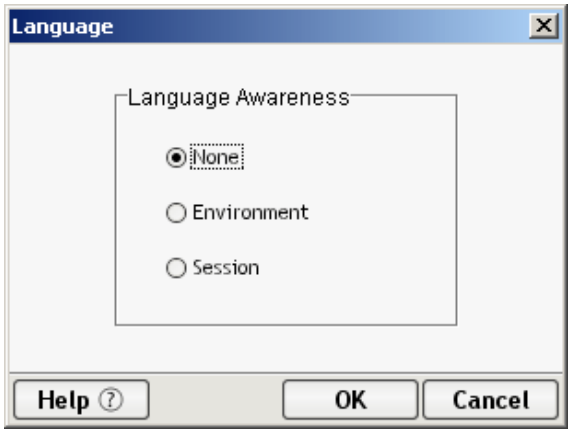
- Bean-managed
- Mandatory
- Never
- Not supported
- Required
- Requires New
- Supports

7 Select an **Output Type** of XML, HTML via PI or XHTML.

8 If a **Stylesheet Resource** is to be used for transforming this service's output, select the appropriate Stylesheet Resource from the pulldown menu provided.

NOTE: You will typically use this option when the Output Type specified in the previous step is XHTML.

9 If you are using a stylesheet (per the previous step) and you have implemented multiple language versions of the stylesheet, and you wish to specify the language version to use, click the Language button. A dialog will appear.



Choose one of the radio buttons:

- **None**: Applies no preference.
- **Environment:** Choose the language of the host machine.
- **Session:** Chooses the language specified in the servlet request.

NOTE: Please refer to “Support for Language Versioning of Resources” for a more detailed discussion of this dialog and its intended use.

10 If this is a J2EE 1.3 (or higher) application, optionally specify a **Security Role.**

NOTE: Security roles are a J2EE-defined mechanism for managing access control. The implementation of this layer is app-server-dependent. It is not implemented by Composer. For more information on J2EE security role concepts, consult the Sun web site and/or your app-server documentation.

11 If this is a J2EE 1.3 (or higher) application, optionally indicate a **Run as Role**.

12 **Save** your deployment-object changes.

Defining File-Based Triggers

The File trigger enables you to set up a scenario in which the appearance of a new file on a particular path on the local hard drive will fire a service. This can be useful for situations in which documents that need to be processed on a timely basis (as part of a workflow) can be handled in automated fashion.

When a File trigger is used, a process on the server monitors a given folder (or subdirectory) on the local hard drive, checking for the appearance of new files at regular intervals. (You can specify any interval you want.) As documents appear in the target directory, the trigger detects them and sends them, one by one, to your Composer service. Each time a file is detected, the following events take place:

- The file is read into memory.
- A copy of the file is written to a destination directory. You will specify a destination directory when you set up your trigger at design time (see setup procedure below). The destination might, for example, be called **\dest**. At runtime, each time a file is processed, Composer creates a *new* folder within the **\dest** folder. The new folder’s name will be its timestamp: e.g., **2003.09.30_08.54.48**. Thus, a copy of the original file (bearing the original name) is written to **\dest\2003.09.30_08.54.48.**
- Depending on the type of file-handling you specified in your trigger, your service will receive, in Input, either the contents of the detected file *or* a URI that points to a copy of the file. The file-handling options you can specify are as follows:

- **Content as XML**—The file itself is assumed to be well-formed XML. The unmodified file becomes the Input message part to your service.
- **Embed Content in XML**—The file contents are copied into a CDATA section of an XML skeleton file. That XML file becomes the Input message part in your service.
- **File Reference**—A URI (relative path) that points to the new copy of the file is placed in an otherwise-empty Input message part.

- The original file is deleted from the source directory (the directory that the trigger process monitors).
- The timer restarts. That is, if the polling interval is ten seconds, the clock will start at zero again as soon as the service has finished running.

File-Handling Options

You can set up a File trigger to pass data to your service in any of three ways.

Content as XML

When you specify this option, each new file that appears in the target directory is assumed to consist of well-formed XML.

Embed Content in XML

The Embed Content in XML option is appropriate when arriving files are not in XML format. (For example: EDI files.) Composer merely wraps the content of the file in a CDATA section, so that your service's Input document looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<Root>
   <![CDATA[ the input file's raw content appears here ]]>
</Root>
```

By default, the root element is named **Root**. But you can override this behavior, as shown further below.

NOTE: Binary content is not appropriate for CDATA, since binary streams can contain XML-illegal characters. If you will be processing binary files, or files that may contain illegal characters, you should *not* use the "Embed Content in XML" option. Instead, use "File Reference," and read the file via a URL/File-Read action inside a component, with base64-encoding enabled. (If you need the data in raw form, rather than base64-encoded, you will need to perform the necessary file I/O operations yourself, in a custom Java class.)

File Reference

When you use the "File Reference" option, your Input will look similar to:

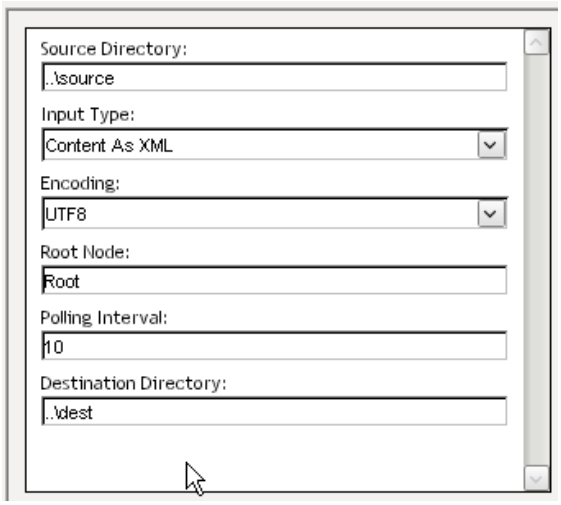
```
<?xml version="1.0" encoding="UTF-8"?>
<Root>..\dest\2003.09.30_09.10.15\myIncomingFile.dat</Root>
```

Again, by default, the root element is named **Root**. But you can override this behavior.

NOTE: All pathnames are relative to the app-server **\bin** directory by default. You can override this behavior, however.

➢ **To create a File-based trigger:**

1. In the Navigation Category pane, under Services, click on **Web Services**.
2. Find the service you wish to deploy and highlight it in the instance pane by clicking.
3. Drag the service onto the **File** node of the Deployment tree, under Service Triggers. When you let go of the mouse, a property sheet similar to this one should appear:



4 Under **Source Directory,** enter a URI pointing to a directory on a local storage drive that should be checked for arriving files. The URI can be a relative path (in which case it will be treated as relative to the **\bin** directory of your app-server installation), or it can be a fully qualified path, such as **d:\temp**.

NOTE: The source directory need not already exist before you deploy the service.

5 Use the dropdown menu under Input Type to specify how each file's contents should be handled. (See the discussion under "File-Handling Options" above.)

- **Content as XML**—Arriving files are assumed to be well-formed XML. The unmodified file becomes the Input to your service.
- **Embed Content in XML**—File contents are copied into a CDATA section of an XML file. That XML file becomes the Input message part in your service.
- **File Reference**—A URI (relative path) that points to the new copy of the file is placed in an otherwise-empty Input message part.

6 Choose an **Encoding**. The default is UTF-8.

7 If you are using Embed or File-Reference handlers, enter the name you would like Composer to use for the root node of the Input document. The default is **Root**. Change this to any XML-legal element name.

8 Under **Polling Interval**, enter the number of *seconds* Composer should wait between inspections of the Source Directory.

NOTE: Polling is suspended when your service is executing. It resumes again when the service has finished running.

9 Under **Destination Directory**, enter the path to the directory that will receive copies of processed files as described earlier. (Composed will create datestamped folders in this directory at runtime: one per file processed.)

10 **Save** your Deployment object.

Testing Considerations

The file-I/O portions of a service that uses the File trigger cannot be tested using ordinary design-time debugging techniques, since the polling, file-handling, and output functions of the trigger will only work on the server. Deployment to a server is a necessary part of testing File-triggered services.

Of course, regular “action model” actions can be tested and debugged as usual in the components that implement your service’s business logic. In order for your action model to operate against realistic Input data, you may need to “dummy up” some specimen XML documents of the type Composer will create in actual operation. (See the XML examples shown under “File-Handling Options” further above.) Use sample documents for action-model debugging; then deploy the service and test.

Defining JSP-Based Triggers

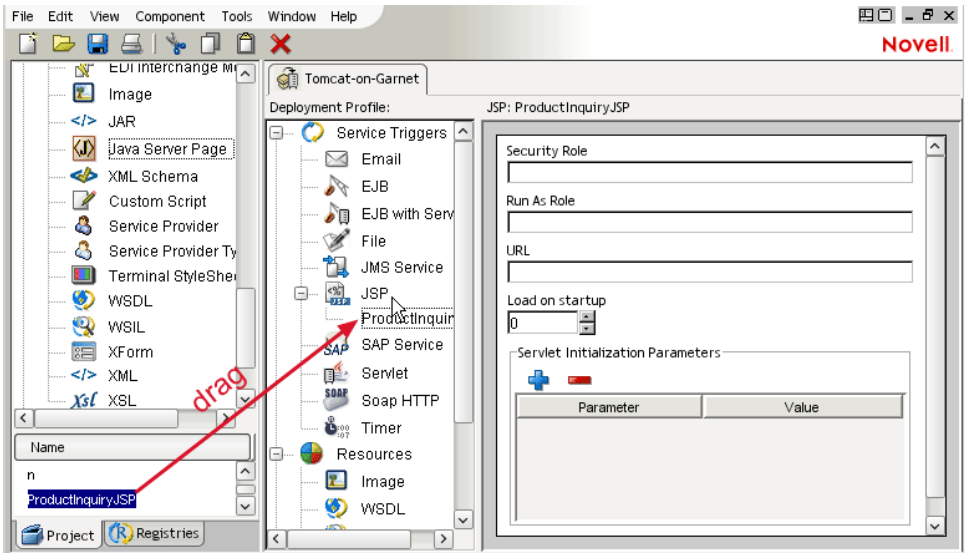
If you are “front-ending” your service with a JSP, you can specify the JSP-to-service binding using the following procedure. Note that you should already have created a JSP Resource for the Java Server Page in question, prior to beginning this procedure. (For information on how to create JSP Resources, see “About JSP Resources” in the section on Resources.)

➢ **To associate a JSP with a service:**

1 In the Navigation Category pane, under **Resources**, click on **Java Server Pages.**

2 Find the particular JSP you wish to use as a trigger in the instance pane and highlight (select) it by clicking.

3 Drag the selected JSP Resource onto the JSP node of the Deployment Profile tree, under Service Triggers.



4 The JSP Resource you selected now appears as a node under the JSP node in the trigger tree, and the JSP Properties sheet is displayed in the editor pane on the right (see illustration).

5 Optionally enter a name to use in **Security Role**.

NOTE: Security Roles are valid for J2EE 1.3 only.

6 Fill in a base **URL** for deployment.

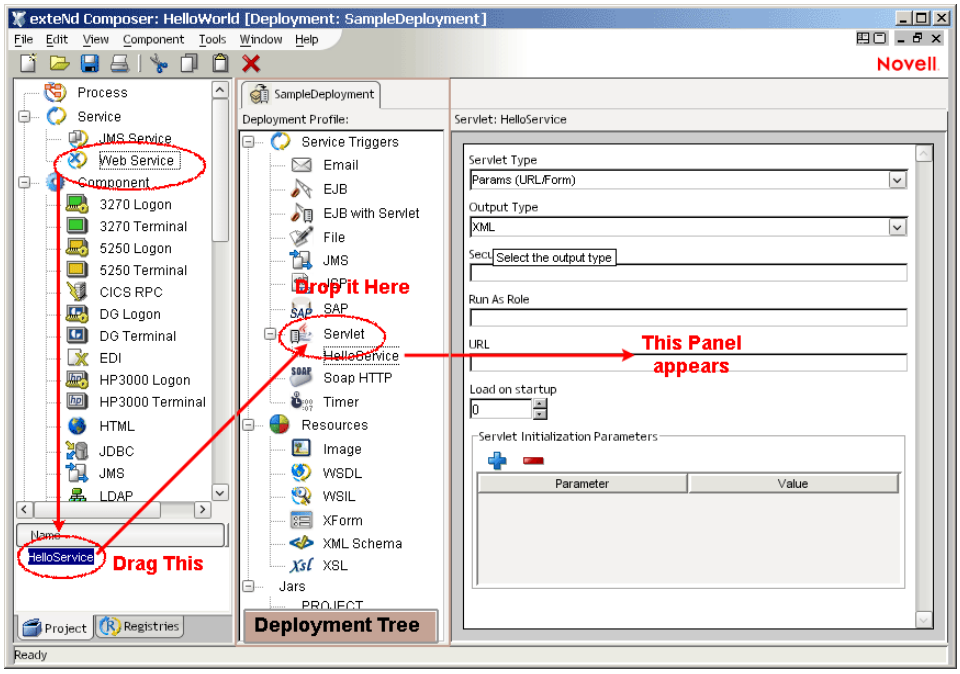
7 **Save** your deployment-object changes.

Defining Servlet-Based Service Triggers

In many cases, you will simply trigger a service off a servlet that handles requests on a given URL. (The servlet can be exposed through a JSP or not. In this case, we will assume not. If you wish to bind a JSP to a particular service, see the discussion at “Defining JSP-Based Triggers” elsewhere.) The servlet may be configured to handle data arriving via HTTP GET or POST; and in the latter case, the XML data might be contained in a particular form field, or it might arrive as multi-part mime content, or it may comprise the content stream of the HTTP POST.

➢ **To define a servlet-based trigger for a service:**

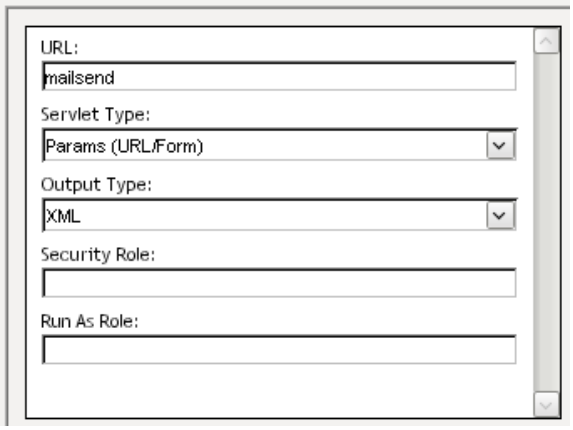
1 In the Navigation Category pane, under Services, click on Web Services. The names of existing services will appear in the instance pane.



2 In the instance pane, find the service you wish to deploy and highlight (select) it by single-clicking.

3 Drag the selected service onto the Servlet node of the Deployment tree, under Service Triggers, as shown above.

4 The service you selected now appears as a node under the Servlet branch, and the Servlet Properties sheet is displayed in the editor pane.



5 Fill in a **URL** for deployment. This will form the tailmost fragment of the URL for your service. The complete URL will be something like:

http://localhost:80/[MyDataBase]/[MyDeploymentEAR]/myurl

where **[MyDataBase]** is the database in which the deployment will occur on the app server (Novell exteNd app servers only); **[MyDeploymentEAR]** is the name of your Composer project (Professional Edition) or Deployment xObject (Enterprise Edition); and **myurl** is the value you entered above.

6 Select an appropriate **Servlet Type** from the dropdown list as the source of data which will be used as input to the service. The choices are:

- Params (URL/Form)
- XML (MIME/Multi-Part)
- XML (HTML Form Field)
- XML (HTTP/Post)

7 Select an **Output Type** denoting the MIME type of the response data for the service. Valid types include:

- XML
- HTML via PI
- XHTML

8 Optionally enter a **Security Role** name.

NOTE: **Security Roles** are valid for J2EE 1.3 only.

9 Under **Run As Role**, type the name of a Role to use while running the service.

10 **Save** your deployment-object changes.

Defining SOAP Triggers

The procedure for associating a SOAP trigger with a Web Service involves a process similar to the ones described above.

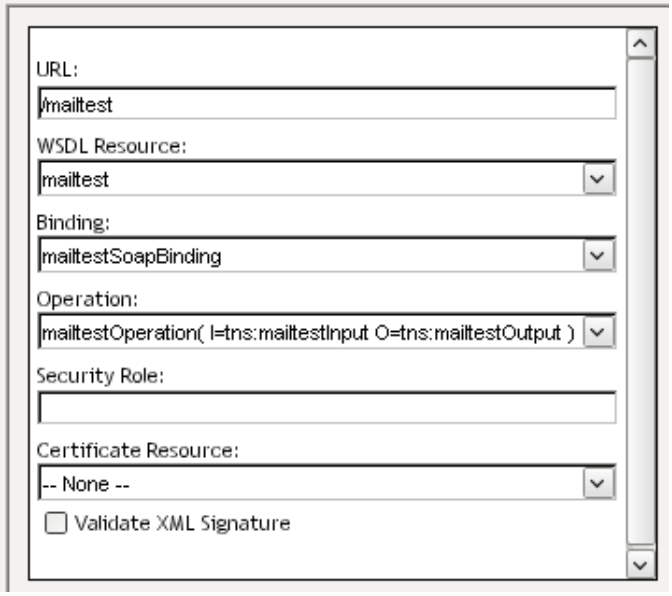
NOTE: The following procedure assumes familiarity with SOAP, WSDL, and XML Signature concepts. If you are not familiar with these technologies, consult the **http://www.w3.org** web site and/or other resources as necessary before proceeding.

➢ **To associate a SOAP trigger with a service:**

1 In the Navigation Category pane, under Services, Click on **Web Services**.

2 Find the service you wish to deploy in the instance pane and highlight it by clicking.

3 Drag the service onto the **SOAP HTTP** node of the Deployment Profile tree, under Service Triggers. The property-sheet pane changes to the following appearance:



4 In the property sheet, enter a **URL** name for the service. This is the final portion of the URL (not the complete URL).

NOTE: An HTTP GET on this URL will return the WSDL for this service. An actual SOAP request on the URL will trigger the service.

IMPORTANT: The value you enter here will be reflected through to the <service> element of your service's WSDL after deployment. Changes will also be reflected in child elements <port> and <soap:address>. In other words, the <service> element is updated dynamically to use the URL you specify here.

5 Select the appropriate **WSDL Resource** if it is not already displayed.

6 Select the appropriate **Binding** if it is not already displayed. (Some WSDLs define more than one binding; when this is the case, you can choose the binding that applies to the URL you specified earlier. In the vast majority of cases, you will simply accept the default value shown.)

7 Specify the **Operation** that this service will handle.

8 Optionally enter a **Security Role** (for J2EE 1.3+ applications).

9 Optionally use the dropdown list to select a **Certificate Resource** (if any) from the ones currently associated with the project. This is necessary only if the service will sign outgoing responses using the XML Signature scheme. (See "About Certificate Resources" for more information on Certificate Resources and their usage.)

10 If the service requires inbound SOAP requests to be digitally signed, check the **Validate XML Signature** checkbox.

NOTE: If you check this checkbox, it means that every inbound request must contain digital signature information in the SOAP header, in accordance with the XML Signature standard. If a request does not contain such a signature, a SOAP fault will result.

11 **Save** your deployment-object changes.

Defining Timer-Based Service Triggers

A Timer trigger causes a particular Composer service to be executed on a periodic basis, independent of any events. This type of triggering mechanism lends itself to many use-cases. For example:

- You may want a service to generate daily or weekly reports: activity reports, inventory reports, payroll, financials, etc.
- You may have "batch jobs" that need to run every night.
- You may have a need to run maintenance jobs on a scheduled basis, such as scanning a database for stale data every 48 hours, pruning "opt-outs" from a mailing list every day, etc.

Scheduled Tasks versus Repetitive Tasks

Composer's Timer trigger supports two types of periodic invocation: two notions of repeat-processing. At one level is the idea of *scheduling*: tying the execution of a process to one or more fixed dates on a calendar. This is *calendar-based recurrence*, or simply *recurrence*. Composer supports recurrence in the general case. You can schedule a job to occur once (i.e., *recur* zero times), at a given date/time, or you can schedule it to occur at multiple dates/times (which might not be evenly spaced). The key intuition is that the job follows a *schedule* which begins on a certain *date* and recurs zero or more times at other dates.

The second notion of timing supported by the Timer trigger is periodic invocation of a process until a certain number of invocations has occurred. The use case here is "This service needs to execute X times, with a wait-time between executions of Y seconds" (or minutes, days, etc.), independent of calendar date. The implied parameters are an execution *count* and an execution *interval*.

By combining these two notions of timing, it's possible to achieve a high degree of customization of execution schedules. For example:

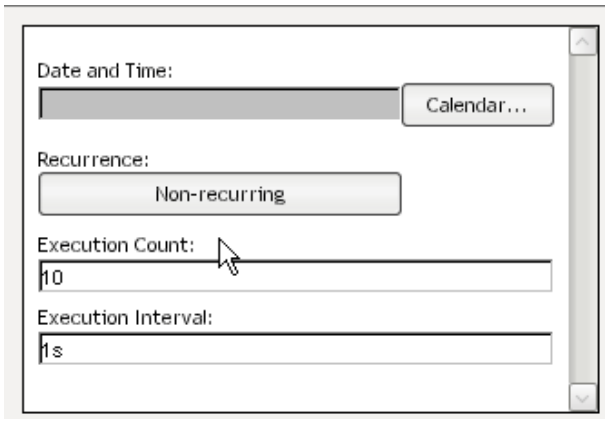
- To execute a service once an hour during business hours (e.g., 9:00 to 5:00), you can schedule the service to recur daily, beginning at 9:00, with an execution count of nine and an execution interval of 60 minutes.
- To execute a service once an hour, continuously, schedule it as a daily recurring item, with an execution count of 24 and an execution interval of 60 minutes.

Composer allows you to configure recurrences based on daily, weekly, or monthly invocations. In the weekly case, you can further specify recurrence by one or more specific days (such as every Sunday, or Tuesdays-and-Saturdays, or Monday-Wednesday-Friday). When the recurrence is monthly, you can further specify that invocation occurs on the first day of the month, the last day of the month, or a specific day.

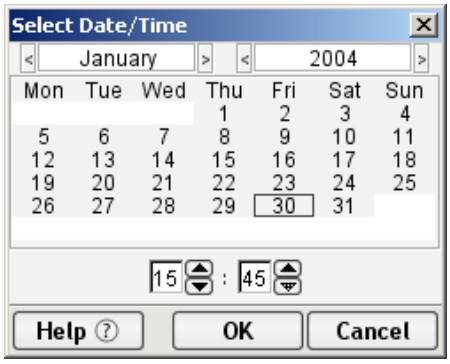
➢ **To create a Timer-based trigger:**

1. In the Navigation Category pane, under Services, click on **Web Services**.
2. Find the service you wish to deploy and highlight it in the instance pane by clicking.

3 Drag the service onto the **Timer** node of the Deployment tree, under Service Triggers. When you let go of the mouse, a property sheet similar to this one should appear:



4 If you want the trigger to begin on a certain date, click the **Calendar** button. A dialog will appear, similar to the one below.

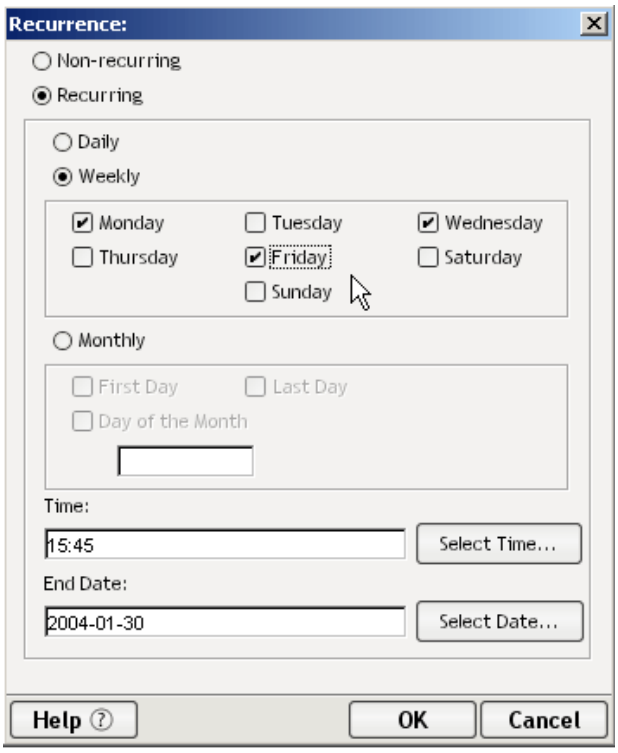


Use the various controls in this dialog to select the date and time for initial invocation of the service. Then click **OK** to dismiss the dialog. The date and time you chose will appear in the **Date and Time** field of the trigger's property sheet.

5 If this is to be a recurring process, click the **Recurrence** button. (It will be labeled "Non-recurring" until you have specified more information.)

NOTE: The Recurrence button will be enabled only if a date is showing in the Date and Time field. If the button is disabled, go back to the previous step.

When you click the Recurrence button, a dialog will appear as shown below.



6 Click the **Recurring** radio button. Other controls in the dialog become enabled.

7 Click the **Daily**, **Weekly**, or **Monthly** radio button as appropriate. Additional controls become enabled.

- Under **Weekly**: Check any checkboxes that apply. Your service will execute every week on the days you have checked.
- Under **Monthly**: Check any checkboxes that apply. Your service will execute on the days specified.

8 Under **Time** (near the bottom of the dialog), enter a time (in 24-hour-clock format) representing the time of day when the service should kick off. If desired, use the **Select Time** button to bring up a time chooser dialog with spin controls for specifying hour and minutes.

9 Under **End Date**, enter a date (in YYYY-MM-DD format) representing the final date beyond which no more executions will occur. If desired, use the **Select Date** button to bring up a date chooser dialog with point-and-click controls for specifying a calendar date.

10 Click **OK** to return to the Timer-trigger property sheet.

11 Under **Execution Count**, enter the total number of repetitions for execution of this service.

12 Under **Execution Interval**, enter the time interval to wait between executions. (Composer will wait this amount of time, *after the service finishes*, to execute it again.) Append ‘s’ to the numeric value for seconds; ‘h’ for hours; ‘d’ for days. For example, **8h** means “eight hours.” The default (if no units are specified) is seconds.

13 **Save** your Deployment object.

Note that it is possible for you to specify recurrences, execution intervals, and counts that, taken together, make little sense (such as daily recurrence of a process that is to repeat 10 times at 25-hour intervals). Composer won’t complain: It will simply add points to the execution timeline, and try to execute your service when each point arrives. The results may or may not be what you expect. It’s up to you to sanity-check your trigger’s parameters.

Specifying Other Project Resources for Deployment

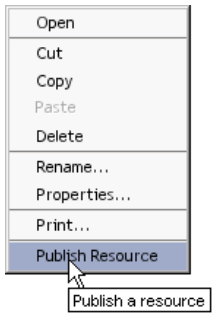
Resource objects listed in the object detail pane can be added to a deployment by simply dragging them onto the appropriate tree node in the deployment-object tree pane. This will add a new child to the category in question. You will be able to drop resources only onto targets that are appropriate for the category. When you hover over a disallowed drop target, the cursor changes to a circle-slash symbol.

The drag-and-drop UI metaphor makes resource deployment simple and quick. For example: To deploy an image resource, just drag an image resource from the object detail pane to the "Image" node of the deployment tree pane. Since resources have no associated properties, the deployment properties pane would be blank in this case. Default URLs are automatically assigned based on both the resource type and the base URL that were defined when creating the deployment object.

Under some circumstances, you can highlight a resource in the instance pane and then right-click on it to expose a context menu from which to deploy the object. Two conditions must be met:

- The resource you've selected must be one that can be "made publicly available" (published to the outside world) in a deployment object. For example, Image resources can be published on a URL. But Custom Script resources generally would not be since they are used internally by Composer.
- A Deployment Object must be open in the editor pane.

If (and only if) these conditions are met, then right-clicking on a resource will bring up a context menu, in which the last (bottommost) command is a Publish command.



Deploying Your Project to the Server

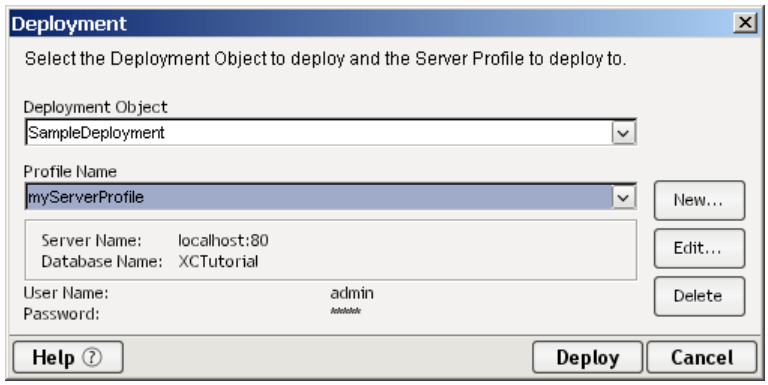
Once all the required services and resources have been added to the Deployment xObject, and all trigger associations have been specified, the actual deployment of your project to the app server can occur.

NOTE: The features described in this section are available only in Composer Enterprise Edition.

➢ **To deploy your project:**

1 Choosing **Deploy Project** from Composer's **File** menu.

2 This will take you to a dialog that allows you to associate a Deployment xObject with a server profile.

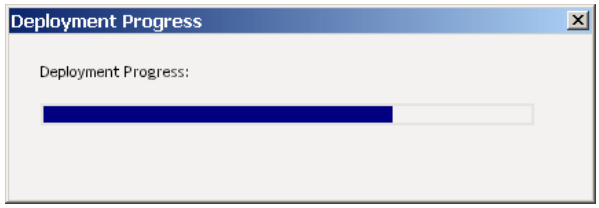


3 Select a **Deployment Object** from the dropdown list.

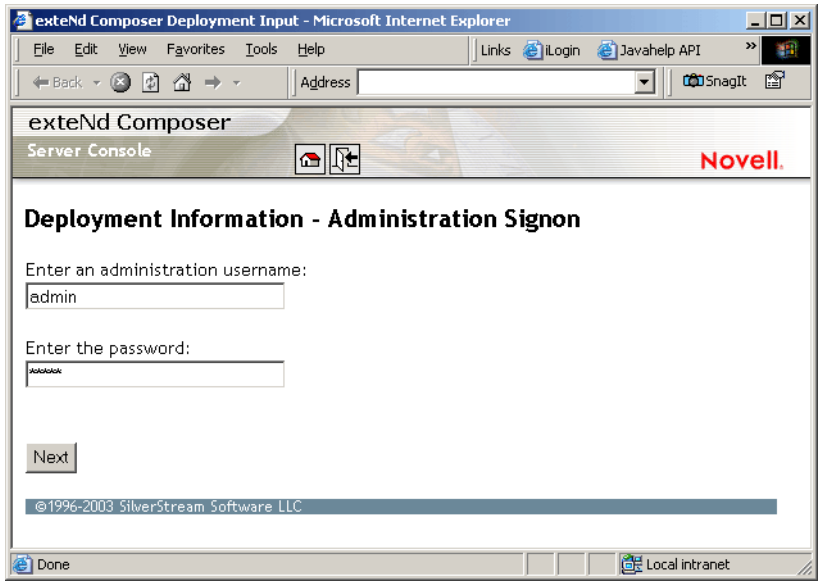
4 Select a **Profile Name** from the dropdown list of server profiles.

5 Click on **Deploy** to deploy your project.

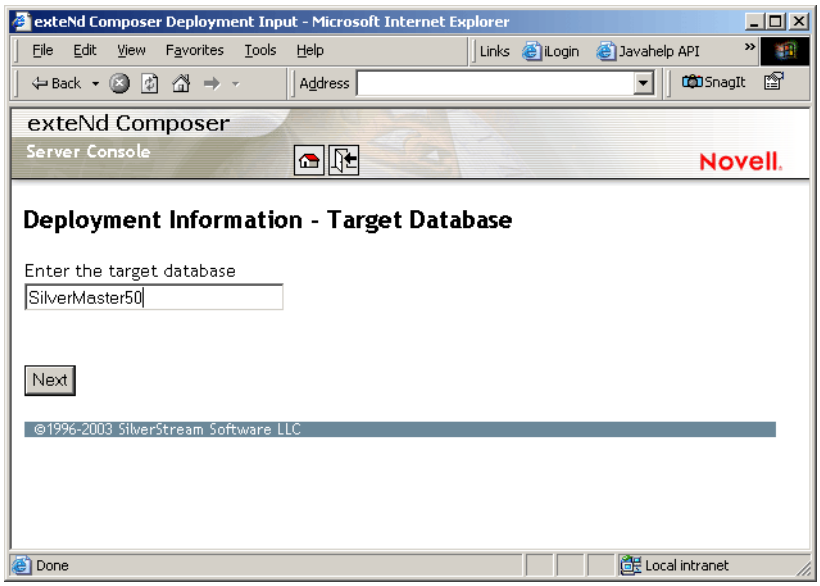
6 You will see a status bar indicating the progress of the deployment:



7 When the progress thermometer is finished, your web browser will launch and you will see a screen similar to the following. (This example assumes that you are deploying to the Novell app server. Somewhat different screens apply in the case of other app servers.)

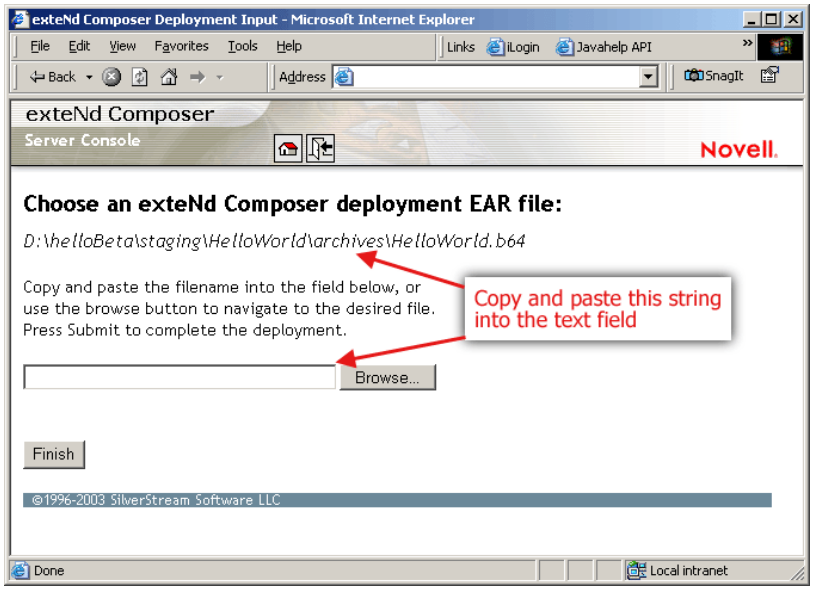


8 Enter the name and password information appropriate to your app server in the spaces provided, then click the **Next** button. A new screen will appear.



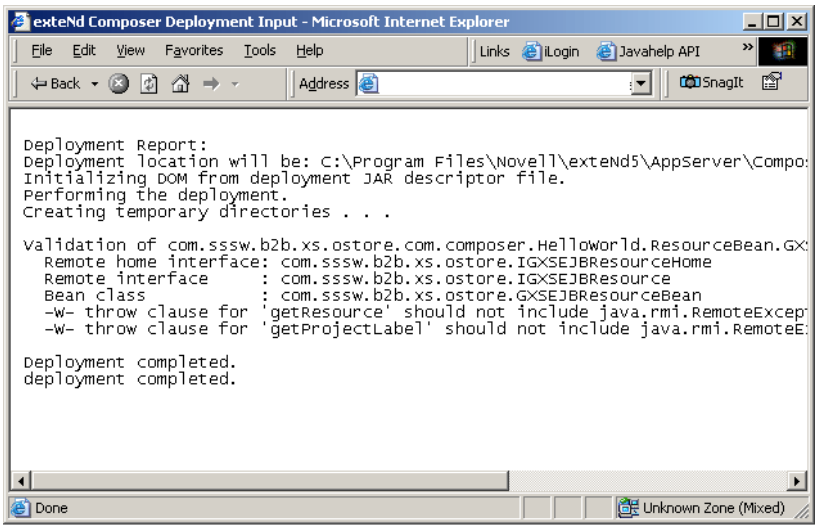
9 If you are deploying to a Novell 5.x app server, this screen prompts you for the name of the database into which the project EAR will be deployed. The default is SilverMaster.

10 Click the **Next** button. A third and final browser screen appears.



11 Copy and paste the deployment-archive name (i.e., the file name shown in italics near the top of the page) into the text field shown. (Alternatively, use the Browse button to browse your file system until you've located the deployment archive you wish to deploy.) This is the delpoyment EAR that Composer created in a staging area (the staging area described earlier in "To create a Deployment Object:", above).

12 Click the **Finish** button. A status report page will appear:



This page will tell you whether the project deployed normally, or an error was encountered. In case of error, there will usually be a complete stack-trace listing information that can be helpful in troubleshooting the cause of the problem. (Note that a delay of more than a few minutes in navigating between the browser screens described above can cause connections to time out. In that case, repeat the deployment procedure as needed.)

Deployment from exteNd Director

You can package your project into a deployable EAR archive using exteNd Director, if you do not want (or are unable) to use Composer's native deployment facilities.

NOTE: For users of the Professional Edition exteNd suite (which does not expose the Deployment xObject features discussed earlier), the deployment procedures outlined here represent the *only* way to deploy Composer applications built in the Professional Edition suite. If you are a user of the Enterprise Edition suite, you may choose to deploy *either* from Director (as outlined here) *or* directly from Composer (as outlined above).

The basic procedure is straightforward:

- Launch Director
- Create or open an EAR-based project Director project
- Add your Composer **.spf** file as a Subproject to the already-open EAR project.
- In Director, use **File > New > Web Services > Composer Web Service** to initiate the wizard that will lead you through the creation of "service trigger" artifacts for a given Web Service in your project. (This step will need to be repeated for each service in your project.)
- Deploy the EAR using Director's regular deployment wizard.

Director also has code-generation features to help you create a variety of different kinds of skeleton files containing Java code to trigger Composer services directly. (This is discussed further below.) By editing the generated source code, you can quickly embellish, override, or extend Composer's "plain vanilla" service triggers to suite your needs.

NOTE: In addition to the discussion here, be sure to see the *Novell exteNd Director* documentation for additional information on the many J2EE deployment capabilities of Director.

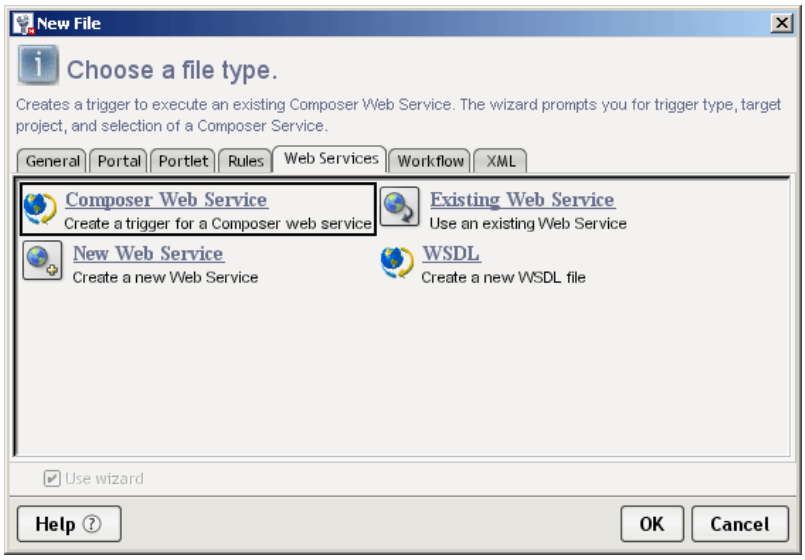
Composer Web Service Wizard: SOAP Service Deployment

If your Director project contains a Composer (sub)project, you can use Director wizards to specify deployment parameters for a service that is to be triggered by a SOAP servlet. The procedure is as follows.

➢ **To prepare a Composer service for deployment as a SOAP service:**

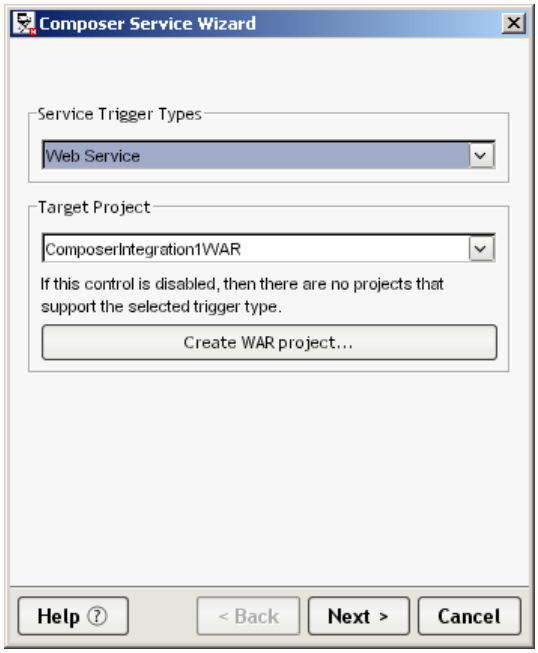
IMPORTANT: In order for the following steps to succeed, you must have an EAR, WAR, or EJB-JAR project open in Director. WSDL must also exist for the service in question. (See "To generate a WSDL Resource from an existing service or create one in the XML editor:" in the Resources chapter of this guide for information on how to generate WSDL automatically.)

1 In Director, choose Control-N or **File** > **New** > **File**. The new-file dialog opens.



Choose the **Web Services** tab.

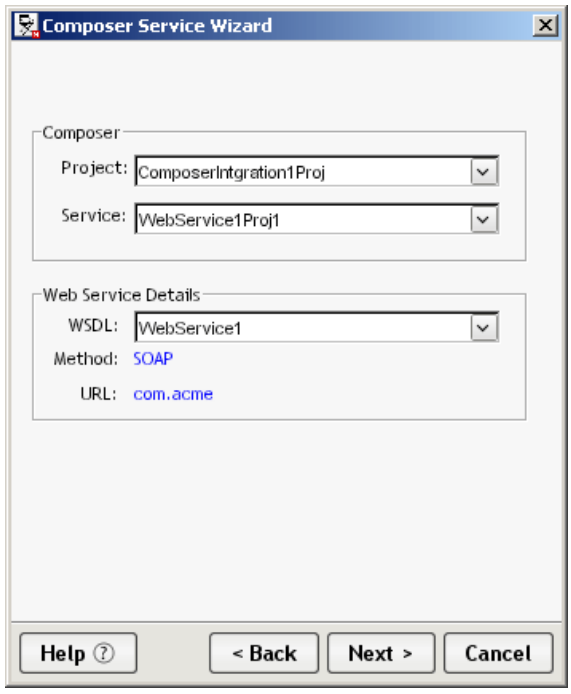
2 In the Web Services tab, select **Composer Web Service** and click **OK**. The first panel of the Composer Web Service wizard appears.



3 Use the pulldown menu control to select a **Service Trigger Type** of “Web Service.”

4 Use the pulldown menu control to select a **Target Project**.

5 Click the **Next** button. A new wizard screen appears.



6 Use the pulldown menu controls to select a Composer **Project** and a **Service** within that project.

7 Using the pulldown menu, choose a **WSDL** resource that describes this service.

8 Click the **Next** button. A new wizard screen appears.

9 Enter a path next to **URL Path** describing the location of the service. (This may or may not be prepopulated.)

10 Next to **Service Name**, select the appropriate service name if it is not already showing.

11 Next to **Port**, select the appropriate port description if it is not already showing.

12 Choose an **Operation** from among those listed in the WSDL. (This list will be prepopulated.)

13 Optionally choose a **Certificate Resource**.

14 Optionally check the **Validate Request XML Signature** checkbox if you want to require that every consumer of this web service provide a digital signature as part of the SOAP request (using XML Signature standard techniques).

15 Click the **Finish** button. A summary dialog will appear, explaining what was done.

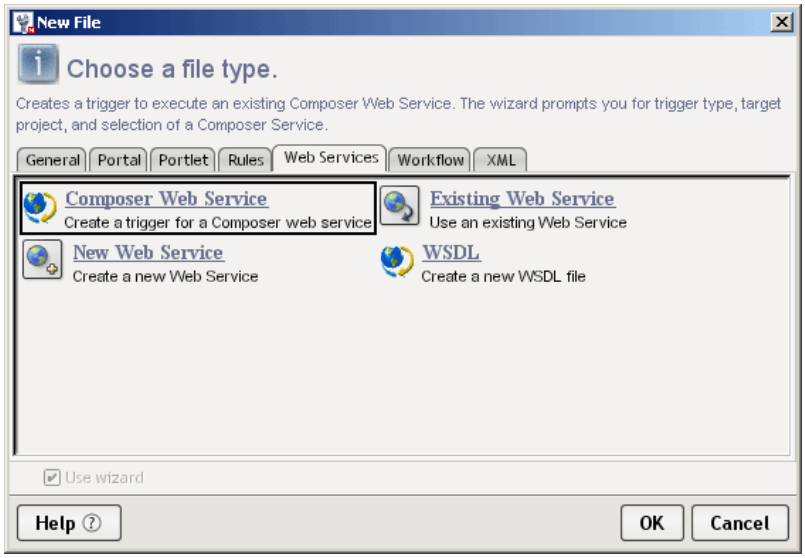
Composer Web Service Wizard: JSP and Servlet Triggers

If your Director project contains a Composer (sub)project, you can use Director wizards to specify deployment parameters for a service that is to be triggered by a servlet or JSP. The procedure is as follows.

➢ **To create a servlet or JSP to trigger a Composer Web Service:**

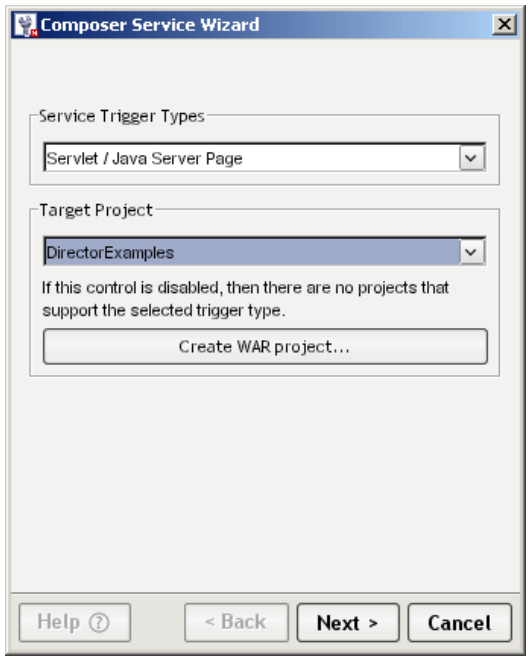
IMPORTANT: In order for the following steps to succeed, you must have an EAR, WAR, or EJB-JAR project open in Director.

1 In Director, choose Control-N or **File** > **New** > **File**. The new-file dialog opens.

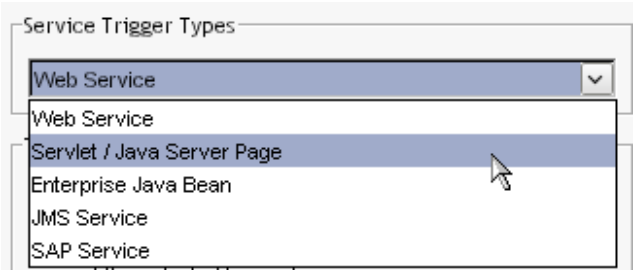


Choose the **Web Services** tab.

2 In the Web Services tab, select **Composer Web Service** and click **OK**. The first panel of the Composer Web Service wizard appears.



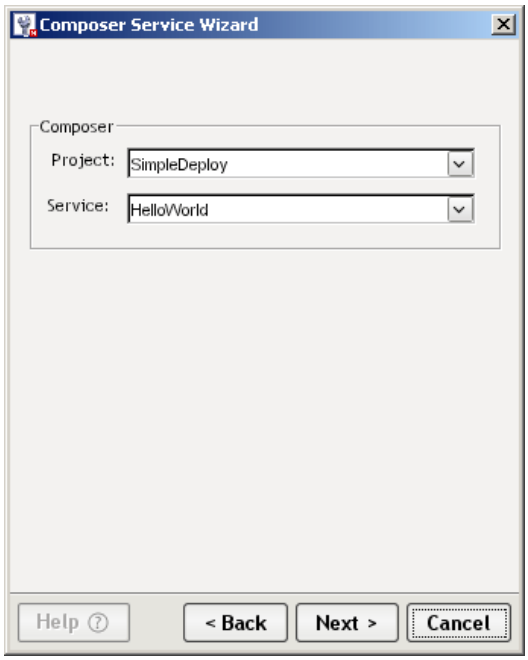
3 Use the pulldown menu control to select a **Service Trigger Type.** Depending on the type of project you have open and what it contains, and whether you are using Professional Edition or Enterprise Edition, you may see any or all of the following choices:



The following discussion assumes that you will be choosing “Servlet/Java Server Page.”

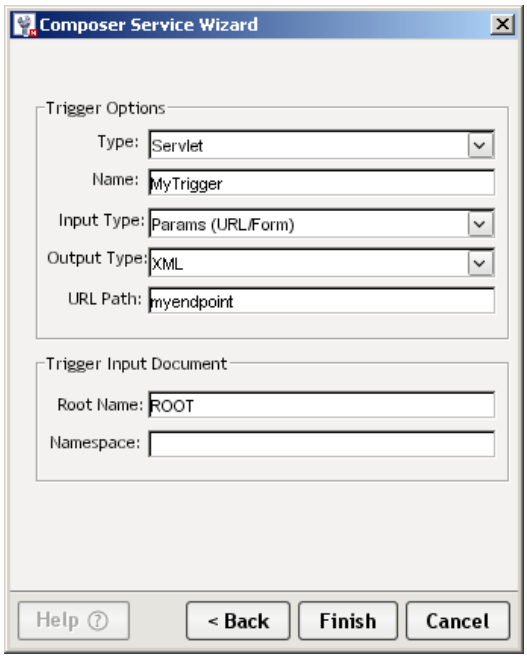
4 Use the pulldown menu control to select a **Target Project**.

5 Click the **Next** button. A new wizard screen appears.



6 Use the pulldown menu controls to select a Composer **Project** and a **Service** within that project.

7 Click the **Next** button. A new wizard screen appears.



8 Under Trigger Options, use the **Type** pulldown menu to choose from the available trigger types.Enter a **Name** for the trigger.

9 Next to **Input Type**, select the option that corresponds to the manner in which your service will receive its XML input document(s):

- Params (URL form)
- XML (HTTP/POST)
- XML (MIME/multipart)
- XML (HTML form field)

10 Use the **Output Type** selector to choose the appropriate output format for your servlet: XML or HTML.

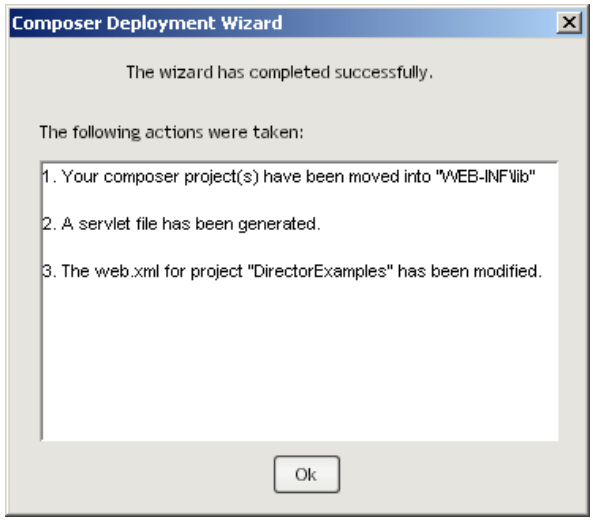
11 In the URL Path field, enter an arbitrary string representing the URL name of the servlet. For example, if your servlet were to handle requests on the following URL, you would enter “abcdef”:

http://myserver:80/mydatabase/myear/abcdef

12 If you selected Params “(URL Form)” as the Input Type, above, you will see two fields under Trigger Input Document.

- Enter an arbitrary string representing the **Root Name** (name of the document root node) for the input document that will be created on-the-fly as part of the marshalling of HTTP-GET params into XML.
- Next to **Namespace**, optionally enter a unique prefix to use for uniqueness in the input document.

13 Click the **Finish** button. A summary dialog will appear, explaining what was done:



14 Dismiss this summary dialog by clicking the **OK** button.

Deploying EARs from Novell exteNd Director

Once you have configured all of your project’s Composer services in terms of deployment parameters, you can deploy the archive using **Project > Deploy Archive** (Control-F5) on Director’s main menubar.

The following topics are discussed in more detail in Director’s documentation:

- Deploying archives
- Building archives
- Validating archives

Director Wizards for Composer Code Generation

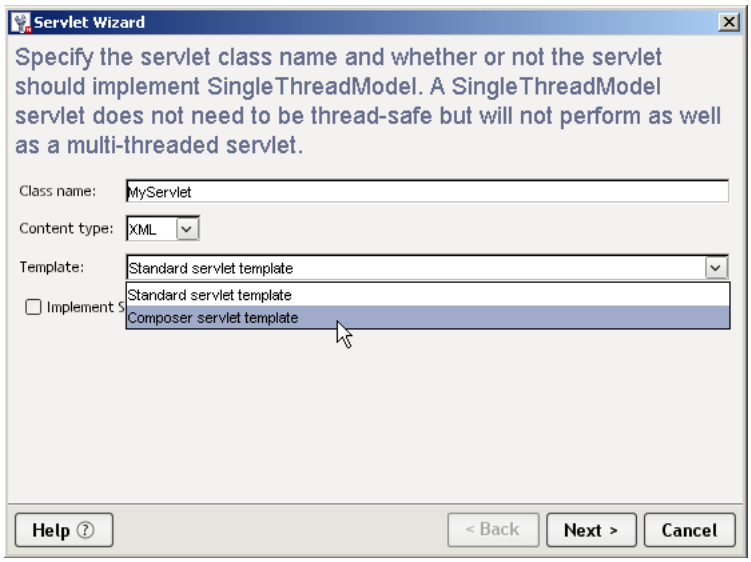
Director has wizards that can help you with the creation of custom service-trigger code of various kinds. Code generation exists for JSPs, servlet code, and custom Java classes that implement direct execution of Composer services.

Director Servlet Wizard

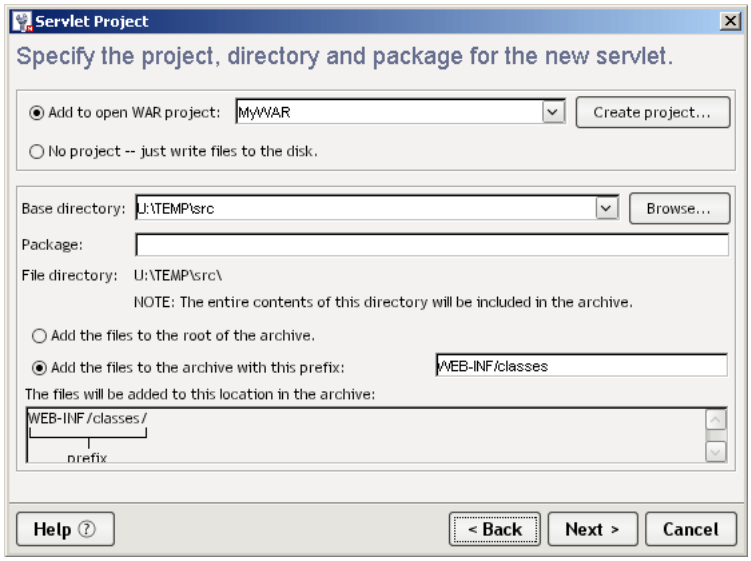
You can generate servlet source code for a Composer service trigger using Director's **File > New** servlet wizard. This wizard differs slightly from the one previously described (see above) in that you do not need to have a Director project open in order to use this wizard. So for example, if you merely want to generate the source code file for later use, you can do so with the following procedure, without first having to open an EAR or WAR project.

➢ **To create servlet code for triggering a Composer service:**

1. In Director, choose **Control-N** or **File > New > File**. The new-file dialog opens.
2. Select the **General** tab.
3. Select the **Servlet** option, and click **OK**. A dialog appears.



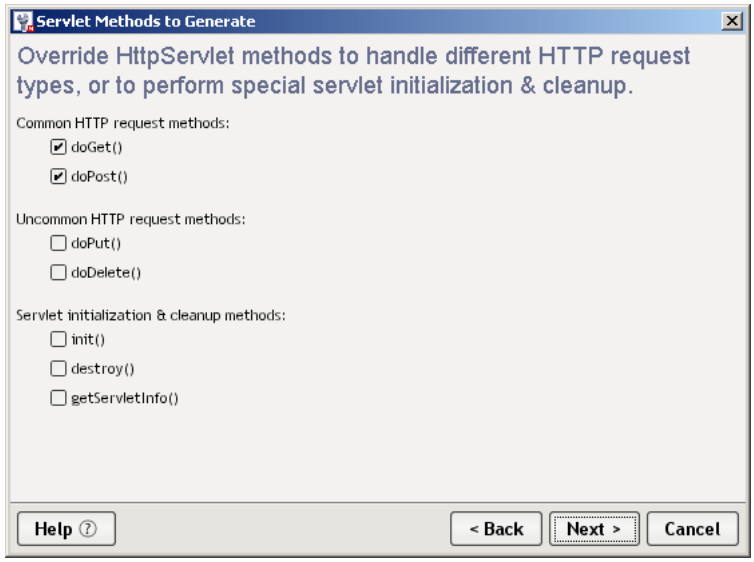
4. Enter a name next to **Class name**.
5. Choose the output-content type of the servlet by using the pulldown menu next to **Content type**.
6. Using the pulldown menu next to **Template**, select **Composer servlet template**.
7. Click **Next**. A new screen appears.



8 If you are adding this servlet to an open project, choose the **Add to open WAR project** radio button as shown in the illustration. (This button will be greyed out if you do not have a project open.) Otherwise, use the second radio button if you merely want to create a file and write it to disk.

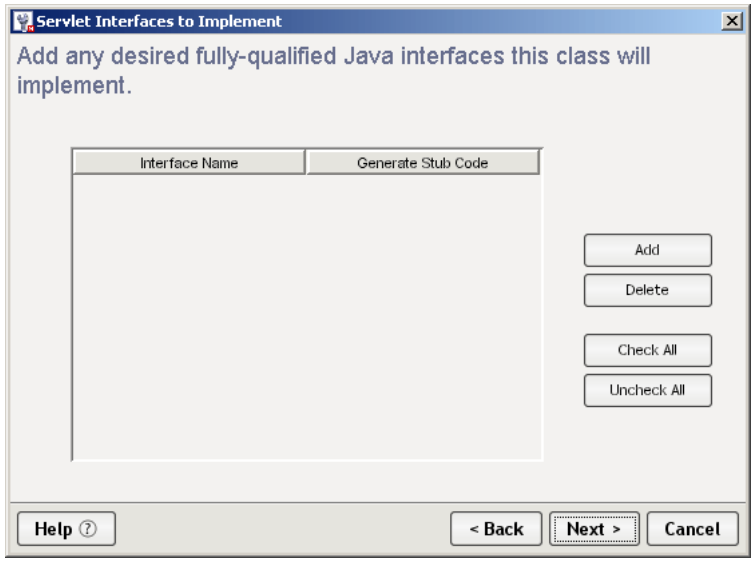
9 Specify **Base directory**, **Package**, and **File directory** information in the lower portion of the dialog, as needed.

10 Click **Next**. A new wizard screen appears.



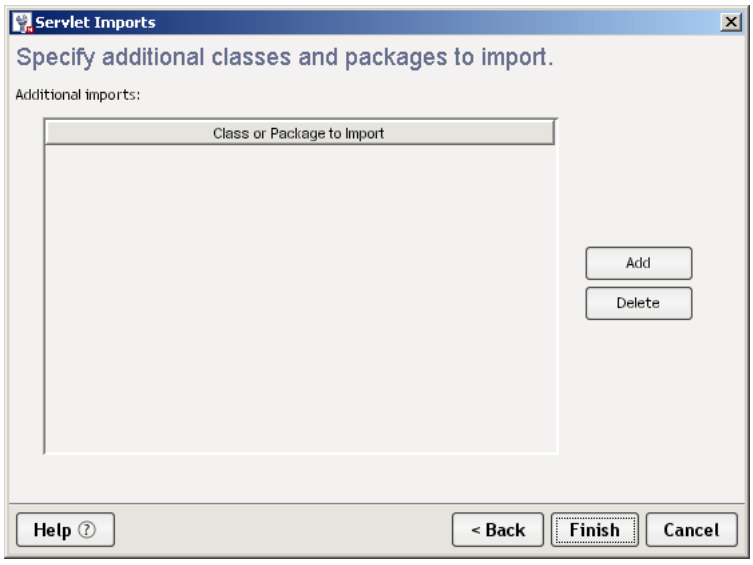
11 Select the servlet methods, if any, that you wish to override, using the checkboxes provided.

12 Click **Next**. A new wizard screen appears.



13 (Optional) Use the **Add** button to specify interfaces that your servlet class will implement, if any.

14 Click **Next**.



15 Click **Finish**. Director generates the servlet source code and displays it in the code editor.

Note that when you generate servlet code this way with no project open, it is up to you to add appropriate entries to the **web.xml** file for any web application in which you wish to use the servlet. (Director will do this for you automatically if you have a WAR project open already, before starting the wizard.)

Director JSP Wizard

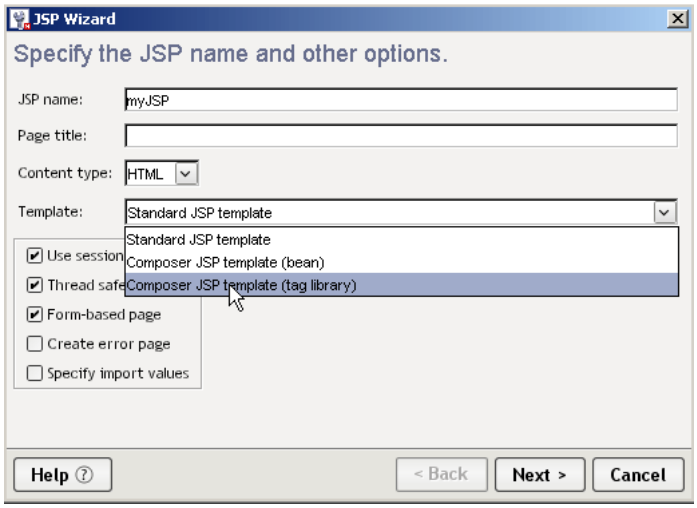
You can use Director's JSP wizard to create a Java Server Page that contains tag-library calls designed to invoke Composer services. The procedure is outlined below.

➢ **To create a JSP for triggering a Composer service:**

1 In Director, choose **Control-N** or **File > New > File**. The new-file dialog opens.

2 Select the **General** tab.

3 Select the **JSP** option, and click **OK**. A dialog appears.



4 Specify a name for the page, in the field labeled **JSP name**.

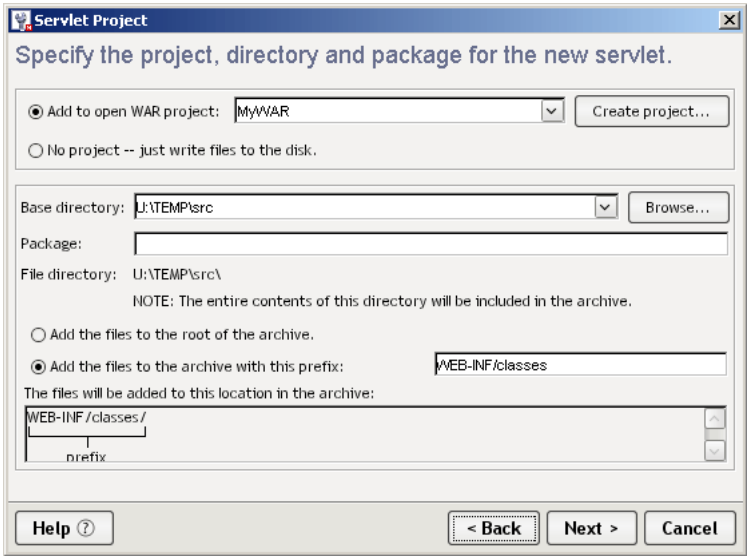
5 Optionally specify a **Page title**.

6 Select the intended output **Content type**.

7 Use the **Template** pulldown menu control to select the template upon which the generated JSP will be based:

- **Standard JSP Template**—Creates a non-Composer JSP
- **Composer JSP template (bean)**—Creates a JSP that delegates Composer service execution to a bean.
- **Composer JSP template (tab library)**—Creates a JSP in which service execution occurs through custom tags defined in the Composer tag library.

8 Click **Next**. A new screen appears.



9 Specify project, directory, and package info for the servlet.

10 Click **Next** (or **Finish**, as applicable) to exit the wizard. Director will create the JSP code and open the new JSP in the code editor.

11 Hand-edit the generated JSP as desired.

Java Class Wizard

You may find it necessary or convenient, in some situations, to invoke a Composer service directly from a Java class rather than via traditional HTTP-based servlet or JSP sessions. Novell exteNd Director contains wizards that can autogenerate a Java stub class that contains methods for calling a Composer service directly.

The following procedure explains how to create a Java class that programmatically executes a Composer service, using the built-in Director wizard.

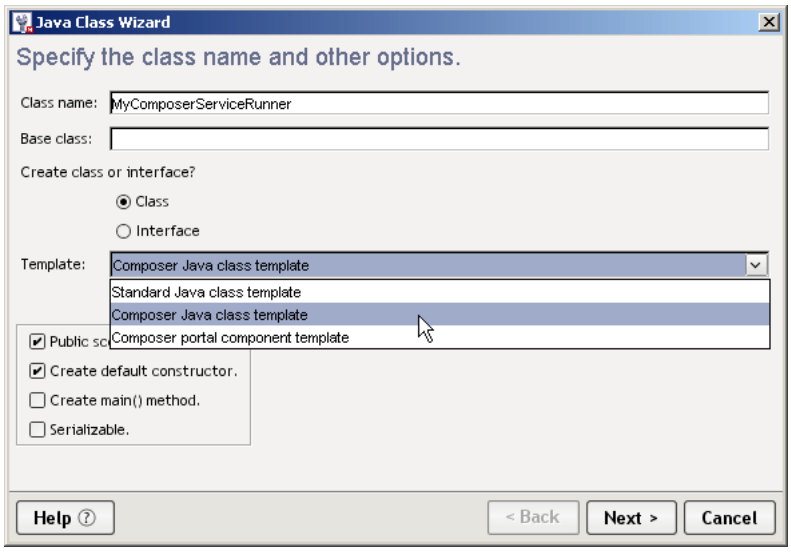
➢ **To create a Java class that invokes a Composer service:**

1 If you have not already done so, open a Composer project or a Director project that contains a Composer subproject.

2 In Director, choose **Control-N** or **File > New > File**. The new-file dialog opens.

3 Select the **General** tab.

4 Select the **Java** option, and click **OK**. A dialog appears.



5 Enter an arbitrary **Class name** for the generated class.

6 If the new class should inherit from another class, enter the appropriate **Base class** name; otherwise, leave the **Base class** field blank.

7 Accept the default (enabled) state of the **Class** radio button.

8 Next to Template, select **Composer Java class template** as shown above.

9 Use the checkboxes provided, as required, to customize the generation of the Java class in terms of visibility, default constructor, creation of a main() method, etc.

10 Click **Next**. A new wizard panel appears, prompting you for the names of any interfaces you want your class to implement. Add interface names as desired.

11 Click **Next**. A new wizard panel appears, prompting you for the names of any additional classes or packages to import. Add names as desired.

12 Click **Next**. A wizard panel prompts you for information about the project, directory, and package(s) with which to associate the new class.

13 Click **Finish**. Director generates source code and displays it in the editor.

The code generated by Director is straightforward. Commented areas show where to insert information pertinent to your service (such as the service's name). The generated code instantiates a **GXSServiceComponentBean** and uses it to invoke your service.

NOTE: For further information about **GXSServiceComponentBean** and other Composer runtime framework classes, consult the relevant Javadoc and source code archives located in your **AppServer/Composer/api_xs** directory.

Compiling and Deploying Director-Generated Code

In Director, use **Project > Compile** (when a Java source file is open in the editor) and/or **Project > Build** to make "generated" items ready for deployment. See the discussion of **archive-related tasks** in Director's documentation.

A properly configured EAR can be deployed directly from Director using **Project > Deploy Archive**. See the discussion of **deployment-related tasks** in Director's documentation.

For More Information

Composer Enterprise Server Documentation

The deployment characteristics for the supported application server environments vary somewhat across product versions and vendors. Accordingly, server-specific instructions for deploying Composer projects are given in separate documents: the *Composer Enterprise Server User's Guide for the Novell exteNd Application Server* and corresponding titles for WebSphere and WebLogic servers. Consult those guides for detailed discussion of deployment issues, including:

- How to resolve CLASSPATH and security issues.
- How to administer running services.
- How to use the Composer Enterprise Server Framework API to create custom trigger objects or manipulate Composer objects programmatically.
- Additional information about the Composer Enterprise Server runtime architecture, and the runtime architecture of Composer services in general.

A The Composer JSP Tag Library

Novell exteNd Composer comes with a tag library file, **composer-taglib.tld**, designed to make it easy for you to call Composer services, and manipulate associated DOM data, within your own Java Server Pages. The tags defined in this file are especially suited to JSP-driven applications in which XML data must be collated, formatted, or post-processed immediately before being sent to the client's browser.

In many cases, you will not need to do any hand-coding in order to use Composer's custom taglib, because many of the wizard-generated JSPs created by Director and Composer contain custom-tag calls. (For details on how to generate JSPs using these wizards, see the discussions at "Director JSP Wizard" and/or "Creating a JSP-Based Service Trigger".) In other cases, you will find it desirable to tap the full power of the tag library, which *does* require hand-coding. The discussion below summarizes the features available in the custom tag library and the syntax for using the various tags.

Preparing to Use the Tag Library

When you use Composer's built-in wizards to generate JSPs containing custom tags, then deploy your project using Composer's deployment UI, Composer takes care of certain "packaging" issues for you. But when you create your own JSPs from scratch and deploy a WAR/EAR manually, you must carry out the following steps yourself in order to ensure that your web app is fully custom-tag-enabled.

NOTE: The following steps apply only if you are hand-creating your own JSP. When you use the Composer and Director JSP wizards, these steps are taken care of for you.

➢ **To tag-enable your JSP-based web app:**

1 Near the top of your JSP(s), insert the following line:

```
<%@ taglib uri="/composer" prefix="composer" %>
```

- The `uri` attribute corresponds to a `uri` definition in the `taglib` entry of your WAR file's *web.xml* file. (See next two steps.)
- The `prefix` attribute allows you to specify a namespace prefix that you can combine with other tags specified in the tag library file.

2 In your WAR file, be sure to include the **composer-taglib.tld** file. A minimal WAR file composition is shown below.

3 In your *web.xml* file, be sure there is a `taglib` element referencing the `uri` from Step 1 above as well as the location of the tag library. For example:

```
<web-app>
        . . .
     <taglib>
          <taglib-uri>/composer</taglib-uri>
          <taglib-location>/WEB-INF/composer-taglib.tld</taglib-location>
     </taglib>
     . . .
</web-app>
```

Custom Tags Defined in *composer-taglib.tld*

The **composer-taglib.tld** file defines the following custom tags:

TAG NAME	PURPOSE
execute	Executes a Composer service
fault	Provides the ability to handle Composer fault documents
forEach	Provides the ability to loop over DOM nodes in a nodeset contained in an output document
hasnopart	Provides conditional processing based on the nonexistence of a particular output message part
hasnovalue	Provides conditional processing based on a particular node being empty
haspart	Provides conditional processing based on the existence of a particular output message part
hasvalue	Provides conditional processing based on a particular node being non-empty
if	Evaluates children if specified XPath condition is true
value	Obtains the data value of a particular node

Each of these tags relies on attribute values that, in effect, pass argument values to the underlying Java methods. Those attributes and their usage are discussed further below, in the individual discussions of the tags.

Tag API

execute

ATTRIBUTE	PURPOSE
converter	Converter class to be used when marshalling input data (see text for discussion)
faultHandled	Flag to indicate whether Composer faults will be handled inside the page using <composer:fault> tags. The default value is “false.”
name	Arbitrary identifier for use within the JSP, to refer to the service in question.
root	Name of the root element of the input message part.
service	Name of the Composer service to invoke.
xmldoc	When the GXS*InputFromJavaObject* converter class is used, this attribute must be specified and must contain the name of the Java String (or String array) variable that points to the input document(s) to be passed to the service.

NOTE: Required attributes are shown in **boldface** type.

Inside your JSP, you will call your Composer service using the `execute` tag. The only required attribute is the `service` attribute, which specifies the fully qualified (full-context) service name. This attribute identifies the service for Composer’s benefit but does not provide any access to the service elsewhere in the JSP (in other tags). If you want to refer to the service in another tag, you can define an alias for it using the `name` attribute.

For example, suppose you’ve built a Composer service called *ListInventory* (in a deployment context of **com.inventory**) and you want to refer to that service in downstream tags as `inventory`. To execute the *ListInventory* service, you would do:

```
<composer:execute name="inventory" service="com.inventory.ListInventory" />
```

This line will execute your service, using the default “InputFromHttpParams” converter class that Composer uses when no other converter is specified. This is equivalent, in other words, to the **Params (URL/Form)** trigger mode.

You can specify the manner in which your service should obtain its input via the use of the `converter` attribute.

IMPORTANT: You should manually inspect all JSPs prior to deployment to verify that the deployment context matches the context used in the `service` attribute of the `execute` tag. (See example above.) To do this: In Composer Enterprise Edition, open your Deployment xObject, then use **File > Properties** to bring up the Properties dialog for the deployment. Inspect the *deployment context* (as shown under “Deployment Context in the Project JAR” on the Deploy tab of the Properties dialog). This context must match the context used in the `service` attribute of the `execute` tag, in any JSP that executes a Composer service.

Converter Classes

Composer Enterprise Server can marshall XML data in a number of ways, using helper classes that are part of the Composer installation. When calling a service from a JSP using the `<composer:execute>` tag, you can specify any of five different kinds of data marshalling.

Possible values for the `converter` attribute are:

```
com.sssw.b2b.xs.service.conversion.GXSInputFromHttpParams
com.sssw.b2b.xs.service.conversion.GXSInputFromHttpContent
com.sssw.b2b.xs.service.conversion.GXSInputFromHttpMultiPartRequest
com.sssw.b2b.xs.service.conversion.GXSInputFromHttpSpecificParam
com.sssw.b2b.xs.service.conversion.GXSInputFromJavaObject
```

These classes are part of the Composer Enterprise Server installation runtime. You do not have to install them, package them, or register them in any special way to use them from your Composer web app.

The mode of operation of each of these classes is described below (and also in the Composer Enterprise Server documentation):

com.sssw.b2b.xs.service.conversion.GXSInputFromHttpParams

This converts `HttpServletRequest` parameters (i.e. those supplied either as URI parameters or form fields submitted) into an XML document. The document will use the root name that the service was defined with, unless you have specified the root name using the `execute` tag's `root` attribute.

com.sssw.b2b.xs.service.conversion.GXSInputFromHttpContent

This opens an InputStream from the supplied `HttpServletRequest` and retrieves the content of the request buffer, which it expects to be in XML format.

com.sssw.b2b.xs.service.conversion.GXSInputFromHttpMultiPartRequest

This class expects an HTML form with content type `multipart/form-data`. It will look for a specific file parameter ("xmlfile") and use that as the XML input document. If the mime type for the file thus found is not `text/xml`, then this class will create an XML document and place the contents of the file in a CDATA section within the XML document.

com.sssw.b2b.xs.service.conversion.GXSInputFromHttpSpecificParam

This class takes the contents of a form field and uses it as the input XML document. By default, the form field containing the XML is expected to be named "xmlfile."

com.sssw.b2b.xs.service.conversion.GXSInputFromJavaObject

This class expects the input XML document to be passed as raw XML contained in a Java String.The class's methods are overloaded in such a way that if your service is designed to handle more than one input document, you can pass multiple XML strings in a String array, in which case each String will be passed to your service as a message part.

fault

ATTRIBUTE	PURPOSE
name	Scopes the behavior of the tag to faults thrown by a <composer:execute> tag whose 'name' attribute has an identical value
part	Can be used to specify a user-defined custom fault document by name. If not specified, this attribute will take on a default value of "_SystemFault."

NOTE: Required attributes are shown in **boldface** type.

You can place a <composer:fault> element in the JSP to wrapper JSP code that should execute when a fault condition occurred in your service. The tag acts as a conditional expression: If a fault part is returned as the result of execution of a service using a <composer:execute> tag, the instructions between the start and end tag of the <composer:fault> element are executed. If no fault condition occurred during execution of the service, then the instructions bracketed by the <composer:fault> tags are ignored.

For example, to handle a standard Composer System Fault that might be returned from one Composer service *and* a custom fault returned from a *second* Composer service executed by a JSP, you might use JSP code similar to the following:

…

```
<composer:execute name="myServ1" service="com.context.myService" xmldoc="myInput"
/>
<composer:execute name="myServ2" service="com.context.myOtherService"/>

<HTML>
  <HEAD><TITLE>My Page<</TITLE></HEAD>
<BODY>
…
<composer:fault name="myServ1" part="_SystemFault">
<B>A Fault has occurred!</B><P/>
Component: <composer:value name="localName" xpath="FaultInfo/ComponentName" /><P/>
Date: <composer:value name="localName" xpath="FaultInfo/DateTime" /><P/>
MainCode: <composer:value name="localName" xpath="FaultInfo/MainCode" /><P/>
SubCode: <composer:value name="localName" xpath="FaultInfo/SubCode" /><P/>
Message: <composer:value name="localName" xpath="FaultInfo/Message" /><P/>
</composer:fault>
…
<composer:fault name="myServ2" part="myCustomFault">
  <B>Doh! A fault happened in my other service!</B>
</composer:fault>
…
</BODY>
</HTML>
```

Note that a JSP may have more than one set of <composer:fault> tags. All tag sets that satisfy a specified name and part will be executed if a fault condition occurs in the service at runtime.

forEach

ATTRIBUTE	PURPOSE
name	Scopes the behavior of the tag to the service whose name attribute (as defined within an execute tag elsewhere in the JSP) has an identical value.
part	Scopes the tag behavior to a particular message part associated with the service specified by name (above).
xpath	An XPath expression to be evaluated against the document associated with part and name attributes (or with the Output message part, if no part attribute is specified). This expression should be one that returns a nodelist (as defined by the Document Object Model).

NOTE: Required attributes are shown in **boldface** type.

The forEach tag provides a looping construct so that you can iterate over the nodes in a nodelist at tag-execution time. The only required attribute is the xpath attribute, which should be an XPath expression that resolves to a nodelist. The expression will be evaluated against the service specified by name and the message part (or DOM) specified in part. If no extra attributes are defined, the most recently executed service's Output part is used.

The following example shows how you could loop over a set of `ITEM` nodes in a message part called `Output1` produced by a service whose alias (defined by the `name` attribute in an `execute` tag) is `inventory`:

```
<composer:forEach name="inventory" part=”Output1” xpath="/MYROOT/INVENTORY/ITEM">
   <composer:value xpath="./ITEMNAME" /><br>
   <composer:value xpath="./SKU" /><br>
   <composer:value xpath="./QTY" /><br>
   <composer:value xpath="./PRICE" /><br>
</composer:forEach>
```

A node value (XML data corresponding to the elements under /ITEM) will be placed in the page where each `value` tag occurs.

hasnopart

ATTRIBUTE	PURPOSE
name	Scopes the behavior of the tag to the service whose `name` attribute (as defined within an `execute` tag elsewhere in the JSP) has an identical value.

NOTE: Required attributes are shown in **boldface** type.

The `hasnopart` tag can be used to enclose a block that should execute only when a particular service (identified by name) has no Output part.

See `haspart` (below) for additional information and a code example.

hasnovalue

ATTRIBUTE	PURPOSE
name	Scopes the behavior of the tag to the service whose `name` attribute (as defined within an `execute` tag elsewhere in the JSP) has an identical value.
part	Scopes the tag behavior to a particular message part associated with the service specified by `name` (above).
xpath	An XPath expression to be evaluated against the document associated with `part` and `name` attributes (or with the Output message part, if no `part` attribute is specified). This expression should be one that returns a DOM node that may contain data.

NOTE: Required attributes are shown in **boldface** type.

This tag can be used to enclose a block of markup that should only be evaluated if a particular node in a particular output message part contains no data. In other words, this tag can be used in cases where it is necessary to handle an empty node. It is complementary in function to the `hasvalue` tag (discussed further below).

haspart

ATTRIBUTE	PURPOSE
name	Scopes the behavior of the tag to the service whose `name` attribute (as defined within an `execute` tag elsewhere in the JSP) has an identical value.
part	Scopes the tag behavior to a particular message part associated with the service specified by `name` (above).

NOTE: Required attributes are shown in **boldface** type.

This tag enables conditional processing based on the existence of a particular output message part from a particular service. Its behavior is similar to that of the `hasvalue` tag in that a number of HTML statements, JSP markup blocks, and/or `<composer:>` tags can be embedded between the `<composer:haspart>` opening tag and `</composer:haspart>` closing tag, and the embedded tags will be processed *only* if the specified part exists.

For example, the following JSP code fragment will display an HTML table and XML data retrieved from "myService" *if and only if* the service actually produced a part named "myFirstPart".

```
<composer:haspart name="myServiceName" part="myFirstPart">
  <table>
    <tr>
      <td>some data</td>
      <td>
        <composer:value name="myServiceName"
         part="mySecondPart" xpath="root/element/element"/>
      </td>
    </tr>
  </table>
</composer:haspart>
```

hasvalue

ATTRIBUTE	PURPOSE
name	Scopes the behavior of the tag to the service whose `name` attribute (as defined within an `execute` tag elsewhere in the JSP) has an identical value.
part	Scopes the tag behavior to a particular message part associated with the service specified by `name` (above).
xpath	An XPath expression to be evaluated against the document associated with `part` and `name` attributes (or with the Output message part, if no `part` attribute is specified). This expression should be one that returns a DOM node that may contain data.

NOTE: Required attributes are shown in **boldface** type.

The hasvalue tag enables conditional processing based on a given XPath node (in a particular message part, from a given service's output) being non-empty. This makes it possible for you to "skip over" a particular node if it is empty, but process or display it if it contains data. For example, consider the following usage:

```
<composer:value xpath="./CUSTOMER/NAME" /><br>
<composer:hasvalue xpath="./CUSTOMER/ADDRESS">
       <composer:value xpath="./CUSTOMER/ADDRESS" /><br/>
</composer:hasvalue>
<composer:value xpath="./CUSTOMER/CITY" /><br/>
<composer:value xpath="./CUSTOMER/STATE" /><br/>
<composer:value xpath="./CUSTOMER/ZIP" /><br/>
```

Data from the /ADDRESS node, in this example, will show up in the JSP's output only if that node is non-empty. Otherwise, it is skipped.

if

ATTRIBUTE	PURPOSE
name	Scopes the behavior of the tag to the service whose `name` attribute (as defined within an `execute` tag elsewhere in the JSP) has an identical value.
part	Scopes the tag behavior to a particular message part associated with the service specified by `name` (above).
xpath	An XPath expression to be evaluated against the document associated with `part` and `name` attributes (or with the Output message part, if no `part` attribute is specified). *This expression should be one that evaluates to a boolean value.*

NOTE: Required attributes are shown in **boldface** type.

The `<composer:if>` tag allows conditional evaluation of a block of markup based on the boolean value of an XPath expression. Consider the following example:

```
<composer:if xpath="string(./CUSTOMER/STATE)=&quot;CA&quot;">
    <b>
<composer:value xpath="./CUSTOMER/STATE" />
</b>
</composer:if>
```

In this example, if /STATE is "CA", the state value will be written out to output, wrapped in boldface tags.

A corresponding `if` block based on modification of the XPath to use *inequality* (instead of equality) can be used inline with the above block to achieve an "else" branch, so that states that are *not* "CA" are not boldfaced.

value

ATTRIBUTE	PURPOSE
name	Scopes the behavior of the tag to the service whose `name` attribute (as defined within an `execute` tag elsewhere in the JSP) has an identical value.
part	Scopes the tag behavior to a particular message part associated with the service specified by `name` (above).
xpath	An XPath expression to be evaluated against the document associated with `part` and `name` attributes (or with the Output message part, if no `part` attribute is specified). This expression should be one that returns a DOM node containing data.

NOTE: Required attributes are shown in **boldface** type.

The value tag allows you to retrieve data from a DOM node at a specified location in a specified message part from a named service. The node location must be specified via an XPath expression. For example:

```
<composer:value xpath="/MYROOT/INVENTORY/DATE" />
```

The data at `/MYROOT/INVENTORY/DATE` will be placed in the JSP output. This example presumes that `/MYROOT/INVENTORY/DATE` is a single, discrete node representing a single piece of XML data. When a *nodeset* is returned, you should iterate over the individual nodes by means of the `forEach` tag, then apply the value tag to each individual node:

```
<composer:forEach name="inventory" xpath="/MYROOT/INVENTORY/ITEM">
   <composer:value xpath="./ITEMNAME" /><br>
   <composer:value xpath="./SKU" /><br>
   <composer:value xpath="./QTY" /><br>
   <composer:value xpath="./PRICE" /><br>
</composer:forEach>
```

In this example, it is assumed that `/MYROOT/INVENTORY/ITEM` returns a set of nodes, each one in turn containing `ITEMNAME`, `SKU`, `QTY`, and `PRICE` child nodes. The above JSP fragment outputs a listing of the relevant data for each `ITEM`.

Notice that the opening `forEach` tag specifies a `name` attribute with a value of `inventory`. This is the local (within this JSP) name of the service whose Output DOM we are accessing at this point. You might very well execute more than one service from a single JSP page, then access data from *each* resulting Output DOM. The `name` attribute allows you to operate on different Output DOMs within the same JSP session.

For More Information

The **xcs-src.jar** file in the **\xc_api** folder of your Composer installation (under **\Common**) contains the actual Java source code for the Composer tag library. It also contains source code for various trigger classes, converter classes, and deployment-related interfaces and support classes that Composer relies on at runtime. Javadoc API documentation for these classes is available in the same folder.

B Reserved Words

The following terms are reserved words in exteNd Composer and should be avoided in any user created terms or objects.

- Input, Temp, Output
- Input1, Temp1
- Input(n), Temp(n)
- ERROR
- Math, Date, String, Array, etc. (ECMAScript reserved words)
- theComponent
- SQLCODE, SQLSTATE, UPDATECOUNT, LASTSQL
- USERID PASSWORD

In addition to the above-listed terms, it is good practice not to use *Java-language keywords* in deployment context strings, user variable names, etc. Reserved words in Java include the following:

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

C Glossary

Action An action is similar to a programming statement: it takes input in the form of parameters and performs specific tasks.

Action Model An Action model is a visual representation of a sequence of actions. An Action model is located in a component editor.

Alias A name given to an element identified by an XPath expression for use in Repeat actions. An alias ensures that the next repeating element matching the XPath expression is processed separately with each iteration of a Repeat loop.

Animate The process of visually executing a component in Composer, step-by-step, for debugging problems or testing new inputs.

Attribute An Attribute is the part of an XML document that is associated with an element and provides descriptive information about the element. An Attribute is also an Object type in the DOM specification.

CDATA A declaration inside an XML document that prevents any character data inside the CDATA section from being interpreted as XML markup language. This allows characters such as the angle brackets (< >) to be used inside an XML document without being interpreted as part of a start or end tag.

Character Data The data contained within an XML document. Character data is any non-markup data. Character data in XML documents are composed of characters from the Unicode character set. See also CDATA.

Code Table A code table stores commonly used business codes and their associated descriptions. Two Code Tables work in conjunction with a Code Table Map to produce a translation from one set of values to another set of codes.

Component A component is an object that accepts one or more XML documents as inputs, uses a collection of actions to operate on the inputs, and returns an XML document as output. Components of various types (see Connect products) can also interface with external non-XML data sources such as relational databases, 3270 / CICS transactions, etc.

Composer Composer is a visual productivity tool used to create exteNd Services and Components that perform XML transformations and external data source connections. Composer creates applications that enable non-XML information sources and environments to inter-operate through the exchange of XML encoded data.

Connect An enterprise connector is an installable Composer component editor (and related resources). It allows you to integrate XML data with an enterprise data source or legacy platform that does not support XML, by providing the user with a visual representation of the environment. An example of an enterprise connector is the JDBC Component editor.

Connection A Connection is a resource used to establish communications with an external data source or with a server that uses HTTP authentication.

Content Editor A dialog box available in the Map Action designed to perform XML element level transformations of data. The Content Editor can splice and re-splice data by character or character position, insert constants, and apply functions.

Custom Script A collection of user-defined ECMAScript functions in a Composer project.

Deployment The process of packaging and installing an exteNd Composer project into an application server/exteNd Server environment for production use.

Document An XML document is typically referred to as a document. The document is also an object type in the DOM specification. Document is often used synonymously with DOM.

Document Handle The name assigned to an XML document's DOM. Default document handles are Input, Input1, Input(n), for input XML Templates; Temp, Temp1, Temp(n) for Temp documents; and Output for all Service and Component results. Custom Document Handle names can be created via Component actions (the Returned ID field), Temp documents (Identifier field), and the XML Interchange action.

Document Type Definition (DTD) A DTD specifies how elements inside an XML document relate to each other. It defines semantic rules about the document, as well as elements to which an XML document must conform in order to be considered a valid document of that type.

DOM A document object model (DOM) is an XML document constructed as an object in a software program's memory. It provides standard methods for manipulating the object. In Composer, DOM is often synonymous with XML Document. DOMs are represented as hierarchical trees with a single root node.

ECMAScript ECMAScript (based on JavaScript) is an object-oriented scripting language for manipulating objects in a host environment. As a host environment, Composer provides ECMAScript access to various objects (primarily XML documents) for processing. ECMAScript in turn provides a Java-like language that can operate on these objects.

Element An Element is a fundamental part of an XML document containing the majority of the document's data. The Element is also an object type in the DOM specification.

Entity An entity in an XML document is a specially formatted placeholder that represents something else. (That is, references to entities are replaced with their entity content.) In XML and HTML, there are certain predefined entities, such as `>` for '>' and `<` for '<'. User-defined entities are also allowed.

ERROR A global variable accessible by any component running in the context of a Service. The value in ERROR is set by the Error Expression in the RAISE ERROR action.

Expression Builder The Expression Builder is an interface in Composer that helps you construct valid ECMAScript and XPath syntax.

GET An HTTP Request Method used in the XML Interchange action. The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI.

Group A Group represents the list of unique values across an element that occurs multiple times in an XML document. Groups are used to control Group Repeat loops by determining how many times the Repeat Loop will iterate (i.e., once for each unique value). Map actions inside a Group Repeat based on a group executes only once for each unique value. A Group is created with a Declare Group action. Groups are referenced by an alias name that is associated with the XPath of an element that repeats.

Group(Detail) A Group(Detail) represents the list of all values across an element that occurs multiple times in an XML document. Group(detail)s are used to control Group Repeat loops by determining how many times the Repeat Loop will iterate (i.e., once for every value). Map actions inside a Group Repeat based on Group(Detail) executes once for every value in the group. A Group(Detail) is created automatically with a Declare Group action. Group(Detail)s are referenced by an alias name that is the same as the group appended with the text "(Detail)." Group(Detail) Repeat Loops must always be used inside a Group Repeat loop.

Group Name An alias for an XPath expression used to define a group for a Repeat action.

Input Input is the term used to describe how a component accepts data. You specify the format for inputs for a component by selecting one or more XML templates when you create the component. All components accept one or more XML documents as inputs.

In Value The data in a code table map you wish to translate to a different value in another code table.

Input DOM The Input DOM(s) of a component is the XML document containing XML encoded information that you wish to map into an external data source and/or transform into another XML document type. The Input DOM(s) is passed into a component by the service (or component) that calls it. Components can accept one or more Input DOMs. Services can accept only one Input DOM from a Service Trigger, but can accept more than one from another component.

JMS Java Messaging Service: A Sun-defined Java API for implementing a standard set of messaging operations and constructs. Most popular message-oriented middleware (MOM) products are either JMS-aware or pure JMS implementations.

JDBC Java Database Connectivity. The Sun-designed Java API for accessing relational database data.

Map A generic term used to indicate the association of a source of data with a target of data for the purpose of copying data from the source to the target. For instance, an XML Map component associates and copies data between source and target XML documents. A Map action associates and copies data between source and target elements or attributes.

Mapping Panes Mapping panes represent sources of data that can exchange information via the Map action in a component. Mapping Panes display the DOMs associated with the current component's sample input and output documents and display representations of external data sources such as relational databases or 3270 screen transactions.

Markup Markup in XML documents consists of reserved metadata symbols and constructs such as start-tags, end-tags, empty-element tags, comments, processing instructions, etc. XML relies on certain tokens, such as angle brackets, to have special meaning. When these symbols appear in XML data, they generally cause the XML document to be invalid. Converting them to entity form (see Entity, above) is one way to pass these "reserved characters" through as XML data. Another way is to wrap markup in a CDATA section (see CDATA, above).

MessageListener An object that is created when a JMS Service is deployed. The MessageListener object registers with a (preexisting) message queue or topic so that a message arriving at the queue automatically "fires" the JMS Service. This provides a messaging-based trigger mechanism for Composer services.

Namespace A mechanism to ensure that names used in an XML DTD are unique so that names from different DTDs can be combined in the same document.

Node A node is the basic object used to build a DOM. DOMs consist of a collection of connected nodes, some of which are XML elements, some of which are attributes, some of which are comments, etc. The node is also an object type in the DOM specification. (Note: attributes, documents and elements are all nodes.)

NodeList An object returned by an explicit XPath expression (e.g., Input.XPath ("INVOICEBATCH/INVOICE/INVOICEDATE")) that contains one or more nodes. Nodelists are usually used in ECMAScript expressions. Only nodelist methods and properties may be applied to a nodelist. To apply any node or element methods to a nodelist, you must first select a single node using the nodelist method item().

Output Output is the term used to describe how a component returns data. You specify the format for output for a component by selecting an XML template when you create the component. All components return a single XML documents as outputs.

Output DOM The Output DOM of a component (or service) is the XML document containing the results of any transformations performed in the component. The Output DOM is the XML document that is returned to the service (or component) that called the component. Components and services can only return one Output DOM.

Out Value The data in a code table map that will be the new value for an associated In Value (see In Value).

POST An HTTP Request Method used in the XML Interchange action. The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. The actual function performed by the POST method is determined by the server and is usually dependent on the Request-URI.

POST with Response An HTTP Request Method used in the XML Interchange action. Same as POST above except that the XML Interchange action is expecting a response XML object back from the origin server.

Project A project is a collection of exteNd objects designed to perform XML integration services. A project holds all the objects for the application you're building.

Project File When you create a project, Composer creates a project file which is stored as *<project name>.spf* in the folder you choose. The project file contains start-up information for your project.

Project JAR When you deploy an exteNd Composer project, all the objects in the project are stored in a single JAR (Java Archive) file, which is installed in the application server for production use.

Project Variable A name–value pair created in Composer for adding replaceable parameters at the project level for Composer services. Project variables are maintained in a separate file (project.xml) which when replaced in a deployed application can change its behavior without re-deploying the entire project. Project variables are accessed via a system DOM called PROJECT/USERCONFIG.

Public An attribute of a custom script function. Public means the function is directly accessible from any action that takes an expression as a parameter and makes the function name appear in the Expression Builder pick-lists. Non-public functions can only be called by other functions.

PUBLIC A variant specification for the XML DOCTYPE instruction. PUBLIC is used to specify a DTD intended for widespread use and accessibility.

PUT An HTTP Request Method used in the XML Interchange action. The PUT method requests that the enclosed entity be stored under the supplied Request-URI.

Resource A resource is an xObject that performs a specialized operation to help services and components carry out tasks. Resource types include Code Tables, Code Table Maps, Connections, and Custom Scripts. At deployment, resources also refer to XML DTDs/Schema files and XSL stylesheets that may be deployed separately from the Project JAR.

ROW TARGET The XPath location that serves as the parent element for rows returned by a SQL statement. For each row returned, a ROW TARGET element is created and each column in the row becomes a child element of the ROW TARGET.

Schema A schema is similar to an XML document definition in that it helps to validate data. But, unlike a document definition, a schema is created in a language that is extensible (i.e. XML). Using a schema, you can define precisely which element names are permitted in a document and, within each element, which sub-elements, attributes, and relations are allowed.

Service A service is used to combine the various components you build to create a logical unit of work within the application server environment. It is initiated with a request XML document and requires a response XML document. The work that is performed and the responsibility of each component depends on the design of your application. A service typically executes various components in a sequential and/or conditional manner and can even execute other services. Other service-level tasks may include general error handling and execution logging operations.

Service Trigger 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 URL and converts inbound data into XML documents as input to the Service it triggers.

System A reserved XML template category for immutable XML templates such as {ANY}.

SYSTEM A variant specification for the XML DOCTYPE instruction. SYSTEM is used to specify a DTD intended for private use by an XML document.

UDDI Universal Description, Discovery and Integration—a public-registry standard that gives businesses a way to describe their services and discover other companies' services online.

Unicode A double-byte character set including thousands of useful characters from around the world. See also UTF-8.

URI Uniform Resource Identifier. An extension to URLs that allows more detailed access to information within an URL.

URL An URL is a Uniform Resource Locator that specifies the syntax and semantics of text strings used to locate and access resources via the Internet. The basic URL is constructed of a scheme identifying the communications protocol and a scheme-specific part identifying the resource; <scheme>:<scheme-specific-part>.

Userfunc A keyword used in XPath expressions indicating that the following term is a Custom Script function that should not be evaluated as part of the XPath.

UTF-8 A character encoding scheme of the Unicode character set whose first 128 characters are compatible with 7-bit ASCII characters allowing many common text editors to create XML.

W3C The World Wide Web Consortium at **http://www.w3.org**. A standards body organized to lead the World Wide Web to its full potential by developing common protocols that promote its evolution and ensure its interoperability.

WSDL Web Services Definition Language—a standard for describing business services in XML. WSDL gives Web Service providers a way to understand the methods necessary to conduct e-business online, in an automated or semi-automated way, with remote partners.

XML Category An XML category contains XML templates. You create XML categories to organize XML templates used in a project.

XML Document Definition An XML document definition is the standard validation method created by the W3C. It defines the rules of the document, such as which elements are present and what structural relationship exists between the elements. A document definition helps to validate the data when the receiving application does not have a built-in description of the incoming data.

XML Meta Data xObjects created in exteNd Composer are stored on disk as XML files. These files are often referred to as a project's meta data.

XML Sample Document An XML sample document is a representative model of the data your application will process in a production environment. Sample documents are used to help build accurate Action Models.

XML Template An XML template contains sample documents, a document definition, and an XML stylesheet associated with a particular document type. You create XML templates in Composer, then use them to describe the inputs and outputs of the components you build.

xObject An xObject is a building block of all exteNd data integration services. xObjects include services, components, resources, and XML templates.

XPath XPath is a language for addressing parts of an XML document. It is a W3C-recommended standard and is used as the primary XML addressing language in exteNd.

XPointer XPointer is a language to be used to identify fragments within any URI-reference that locates a resource of Internet media type.

XSL Stylesheet An XSL stylesheet defines the display properties of an XML document. You create or obtain the stylesheet external to Composer. A stylesheet is useful for a component that is creating a page to be displayed in a Web browser.

XSL XSL is a stylesheet language for transforming XML documents into other XML documents.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X