

ChainBuilder ESB

Visual Enterprise Integration™

Version 1.0.1 – 2007/03/28

Reference Guide



©Copyright 2007
Bostech Corporation
2800 Corporate Exchange Drive
Suite 260
Columbus, OH 43231

Acknowledgements

This document contains proprietary information that is the property of Bostech Corporation. Any reproduction, disclosure, or transfer of this document or the information contained herein without the express written consent of Bostech Corporation is strictly prohibited.

The use of the information contained in this document and the implementation of any of its techniques are the sole responsibility of the client and depend on the client's ability to evaluate the information and implement it into the client's operational environment.

Except for any express written warranties made by it, Bostech Corporation makes no warranties or representations with respect to any information contained herein, whether express, implied, statutory, or otherwise, in fact or in law, including without limitation, any implied warranties of merchantability or fitness for a particular purpose; and in no event shall Bostech Corporation be liable for any special, consequential, indirect, punitive, or exemplary damages in connection with the use of the information contained herein. The information contained in this document is subject to change at any time without notice.

Trademarks

The following trademarks and acknowledgments apply to the information presented in this manual:

- ChainBuilder is a registered trademark of Bostech Corporation.
- Adobe and Acrobat Reader are registered trademarks of Adobe, Inc.
- Java is a registered trademark of Sun Microsystems, Inc.
- Windows (NT, 2000, XP, and Server 2003), .NET Framework, Internet Information Services (IIS) are registered trademarks of Microsoft Corporation.

Credits

The following third-party products are used within the ChainBuilder product, and acknowledgments apply to the information presented in this manual:

- Acrobat Reader is created and licensed by Adobe, Inc.
- This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)
- This product includes software developed by Eclipse (<http://www.eclipse.org/>)

Table of Contents

1. Introduction.....	1
1.1. Introduction to ESB.....	1
1.2. Introduction to JBI.....	7
1.3. ChainBuilder ESB.....	9
1.3.1. Features Overview.....	9
2. Installation.....	13
2.1. Prerequisites.....	13
2.1.1. Hardware Recommendation.....	13
2.1.2. Software Requirements.....	13
2.2. Obtaining the Software.....	14
2.3. Installation Procedure for Windows.....	14
2.4. Installation Procedure for Linux.....	22
3. Running ChainBuilder ESB Server.....	28
3.1. Command Line Interface.....	28
3.1.1. Installing Components.....	28
3.1.2. Deploying Service Assemblies.....	28
3.1.3. Starting and Stopping the Server.....	30
3.1.4. Monitoring Activity.....	30
3.2. Running as a Service.....	33
3.3. Admin Console Web Interface.....	34
4. Configuration.....	34
4.1. Directory Structure for ChainBuilder ESB.....	34
4.2. Directory Structure for ChainBuilder ESB Project and Service Assembly Project.....	38
4.3. Configuration Files.....	41
4.4. Logging.....	42
4.5. Optional Jar Files.....	43
5. CCSL Reference.....	44
5.1. Introduction.....	44
5.2. Introduction to the CCSL Control Files.....	45
5.3. ChainBuilder ESB Data Envelope Format.....	47
5.4. ChainBuilder ESB sendMessage Envelope.....	48
5.5. Endpoint Attributes.....	62
5.6. UPOC Attributes.....	64
5.7. User Scripting Points.....	64
5.8. The Groovy Script Interface.....	64
5.9. Applying the Script.....	64
5.10. Deployment and Running.....	64
5.11. The Error Database.....	64
5.12. Converting a Component to use CCSL.....	64
6. Transformation Reference.....	64
6.1. Transformation Source and Target.....	64
6.2. Transformation Operations.....	64
6.2.1. Operation Types.....	64
6.2.2. Parameters in Transformation Operation.....	64

6.2.3. Expressions	64
6.2.4. Lookup Operation.....	64
6.2.5. JDBC Operation.....	64
6.3. Transformation User Defined Classes	64
6.3.1. User Defined Operation.....	64
6.3.2. User Defined Filter	64
6.3.3. Steps to Create and Use User Defined Classes	64
7. Web Services Support	64
7.1. WSDL Import Wizard.....	64
7.2. WSDL Export Wizard.....	64
7.3. The sendMessage Interface.....	64
8. Component Reference	64
8.1. File Binding Component.....	64
8.1.1. Overview.....	64
8.1.2. Description	64
8.1.3. Configuration Settings	64
8.1.4. Example	64
8.2. Http Binding Component.....	64
8.2.1. Overview.....	64
8.2.2. Description.....	64
8.2.3. Example	64
8.3. JMS Binding Component.....	64
8.3.1. Overview.....	64
8.3.2. Description.....	64
8.3.3. Example	64
8.4. FTP Binding Component	64
8.4.1. Overview.....	64
8.4.2. Description.....	64
8.4.3. Configuration Settings	64
8.4.4. Example	64
8.5. Scripting Support in FTP Binding Component.....	64
8.5.1. connect.....	64
8.5.2. disconnect.....	64
8.5.3. login	64
8.5.4. logout.....	64
8.5.5. siteCommand	64
8.5.6. setConnectionMode.....	64
8.5.7. setTransferMode.....	64
8.5.8. changeWorkingDir	64
8.5.9. changeToParentDir.....	64
8.5.10. get	64
8.5.11. put.....	64
8.5.12. deleteFile.....	64
8.5.13. rename.....	64
8.5.14. createDirectory	64
8.5.15. removeDirectory.....	64
8.5.16. mget.....	64
8.5.17. mput	64

8.5.18. mDeleteFiles	64
8.5.19. changeLocalWorkingDir	64
8.5.20. deleteLocalFile	64
8.5.21. renameLocal	64
8.5.22. createLocalDirectory	64
8.5.23. removeLocalDirectory	64
8.5.24. mDeleteLocalFiles	64
8.6. Sequencing Service Engine	64
8.6.1. Overview	64
8.6.2. Description	64
8.6.3. Example	64
8.7. Content Based Router (CBR) Service Engine	64
8.7.1. Overview	64
8.7.2. CBR Message Identification	64
8.7.3. CBR Routing Rules	64
8.7.4. Example	64
8.8. Parser Service Engine	64
8.8.1. Overview	64
8.8.2. Description	64
8.8.3. Example	64
8.9. Transformation Service Engine	64
8.9.1. Overview	64
8.9.2. Description	64
8.9.3. Example	64
8.10. XSLT Service Engine	64
8.10.1. Overview	64
8.10.2. Description	64
8.10.3. Example	64
8.11. Script Service Engine	64
8.11.1. Overview	64
8.11.2. Description	64
8.11.3. Deployment Descriptor Example	64
8.11.4. IScriptObject Class	64
8.11.5. Example	64
8.12. JDBC Service Engine	64
8.12.1. Overview	64
8.12.2. Description	64
8.12.3. Deployment Descriptor Example	64
8.12.4. Message Formats	64
8.12.5. Message Definition Schema	64
8.12.6. Handler Classes	64
9. ChainBuilder ESB Community	64
Appendix A HTTP UPOC Groovy Source Code	64
Appendix B. Log output from HTTP UPOC	64
Appendix C. Error Database Schema	64

1. Introduction

ChainBuilder ESB is a Java Business Integration (JBI) compliant messaging platform that provides the tools necessary to implement Service Oriented Architecture (SOA) solutions as well as more classic Enterprise Application Integration (EAI) solutions.

1.1. Introduction to ESB

The Enterprise Service Bus is a new type of middleware software that has emerged within the last few years. The industry is still debating about its true definition. Each analyst and influencer is coming out with their own definition of an ESB.

Gartner's definition of ESB:

"An Enterprise Service Bus (ESB) is a new architecture that exploits Web Services, messaging middleware, intelligent routing, and transformation. ESBs act as a lightweight, ubiquitous, backbone through which software services and application components flow."

"Enterprise Service Buses (ESBs) are a new kind of middleware that combine features from several previous types of middleware into one package. ESBs support web services by implementing Simple Object Access Protocol (SOAP) and leveraging Web Services Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI). Many ESBs also support other communication styles that involve guaranteed delivery and publish-and-subscribe; those that don't soon will."

"All ESBs provide some value-added services beyond those found in basic communication middleware, such as message validation, transformation, content-based routing, security, service discovery for a service-oriented architecture (SOA), load balancing, failover and logging,"

"Some services are built into the ESB core, while others run in "plug in" modules. ESBs have a distributed architecture wherein some services are executed near the application programs, rather than in a central hub. ESBs support Extensible Markup Language (XML) and often also support other message formats."

The Wikipedia community has accumulated the following characteristics for ESB:

- It is not an implementation of service-oriented architecture.
- The Enterprise Service Bus (ESB) is to SOA as SOA is to e-business on demand.
- The Enterprise Service Bus is emerging as a [service-oriented infrastructure](#) component that makes large-scale implementation of the SOA principles manageable in a heterogeneous world.
- It is usually operating-system and programming-language agnostic; for example, it should enable interoperability between [Java](#) and [.NET](#) applications.

- It uses XML ([eXtensible Markup Language](#)) as the standard communication language.
- It supports web-services standards.
- It supports messaging (synchronous, asynchronous, point-to-point, publish-subscribe).
- It includes standards-based adapters (such as [J2C/JCA](#)) for supporting integration with legacy systems.
- It includes support for service orchestration and choreography.
- It includes intelligent content-based routing services (itinerary routing).
- It includes a standardized security model to authorize, authenticate and audit use of the ESB.
- To facilitate the transformation of data formats and values, it includes transformation services (often via [XSLT](#)) between the format of the sending application and the receiving application.
- It includes validation against schemas for sending and receiving messages.
- It can uniformly apply business rules, enriching messages from other sources, the splitting and combining of multiple messages and the handling of exceptions.
- It can route or transform messages conditionally, based on a non-centralized policy (i.e. no central rules-engine needs to be present).
- It is monitored for various [SLA](#) (Service Level Agreement) threshold message latencies and other SLA characteristics.
- It (often) facilitates "service classes," responding appropriately to higher and lower priority users.
- It supports queuing, holding messages if applications are temporarily unavailable.
- It is comprised of selectively deployed application adapters in a (geographically) distributed environment.

In his great presentation of [“The Role of the Enterprise Service Bus”](#), Mark Richards attempts to define ESB by exploring its role and core capabilities of the ESB in 10 areas:

1. **Service Mapping:** The ability to translate a business service into the corresponding service implementation and provide binding and location information
2. **Routing:** The ability to send a request to a particular service provider based on deterministic or variable routing criteria
3. **Messaging Process:** The ability to guarantee the delivery of the message without being lost
4. **Message Transformation:** The ability to convert the structure and format of the incoming business service request to the structure and format expected by the service provider
5. **Message Enhancement:** The ability to add or modify the information contained in the message as required by the service provider
6. **Protocol Transformation:** The ability to accept one type of protocol from the consumer as input (i.e. SOAP/JMS) and communicate to the service provider through a different protocol (i.e. IIOP)

7. **Process Choreography:** The ability to manage complex business processes that require the coordination of multiple business services to fulfill a single business service request
8. **Service Orchestration:** The ability to manage the coordination of multiple implementation services
9. **Transaction Management:** The ability to provide a single unit of work for a business service request by providing a framework for the coordination of multiple resources across multiple disparate services
10. **Security:** The ability to protect enterprise services from unauthorized access

A [2006 study by Network Computing](#) had survey respondents grade a set of statements about ESB technology using a scale from "Strongly agree" to "Strongly disagree". The top four statements in which respondents most strongly agreed to were:

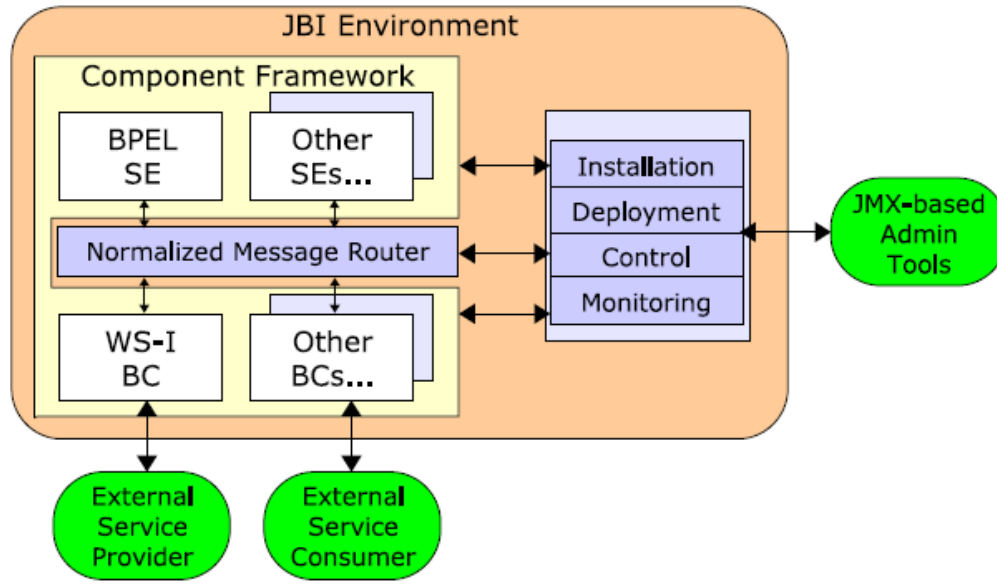
- ESBs must provide adapters to enterprise data sources (SAP, Peoplesoft, Oracle, SQL Server)
- ESBs must support at least rudimentary business process management
- Open standards (JMS, Web services) support is/was a requirement for our ESB implementation
- An ESB must integrate smoothly with existing enterprise application integration (EAI) and message-oriented products.

1.2. Introduction to JBI

ChainBuilder ESB is based on the standard-based component architecture called Java Business Integration (JBI). JBI is a specification developed under the Java Community Process (JCP) for an approach to implementing a Service-Oriented Architecture (SOA). The JCP reference is JSR208.

JBI defines a component framework where components can provide services and consume services within a solution. Individual service units are deployable to components; groups of components are gathered together into a service assembly. The complete service assembly includes metadata for "wiring" the service units together (that is, to associate service providers and consumers) as well as wiring service units to external services.

The following high-level diagram taken directly from the JBI Specification documentation illustrates the JBI environment of service consumers/providers requesting information from a Binding Component (BC) that talks to various Service Engines (SE) to complete the request over the Normalized Message Router (sometimes called the "container"). Installation, Deployment, Monitoring and Control are managed through JMX management tools.



1.3. ChainBuilder ESB

ChainBuilder ESB allows a developer to construct an Enterprise Service Bus style solution. It includes an open source JBI container, ServiceMix, as the ESB backbone. The supporting Chain Builder ESB components are written in Java and easily configured via a graphical user interface plugged into the popular Eclipse development platform. The ChainBuilder Common Service Layer (CCSL) is a robust functionality layer that provides common services to components, like improved error handling. Your ChainBuilder ESB components make use of this enhanced CCSL functionality layer. ChainBuilder also provides a set of pre-built integration components, like X12 mapping, to enable an enterprise's disparate software systems to plug into the ESB.

1.3.1. Features Overview

The features in ChainBuilder ESB v1.0 include:

Operating Systems and Server Environment

- ChainBuilder ESB server and Admin Console server runs as Windows and Linux services
- Supports both Windows and Linux operating systems

Binding Components

- Communication protocol support for FTP, HTTP/SOAP, File, and Java Messaging Service (JMS) for JMS-compliant servers including IBM Websphere MQ

Service Engines (SE)

- Parser SE to parse fixed, delimited, and tagged message formats based on the ChainBuilder ESB XML-based Message Definition Language (MDL).
- Parser SE to parse standard EDI X12 message formats.
- Transformer SE to support XSLT.
- Transformer SE to support the transformation between proprietary message (MDL) formats as well as XML messages defined by an XSD schema. This SE uses the ChainBuilder ESB XML-based Translation (TRN) Language.
- Message enrichment capability to support lookup functions to a relational database using JDBC, or an XML-format of Java property file.
- Content-Based Router SE component that examines the message content to route the message to the proper destination endpoint.
- Sequencer SE component to chain together any number of components to accomplish specific business requirement.
- JDBC SE component that accepts an XML-based message containing standard SQL statement to query or update a relational database.
- Script SE component that allows a developer to write custom functionality or business logic using Groovy script or Java.

Eclipse Plug-ins

- Project wizards to create ChainBuilder ESB project and Service Assembly project
- Eclipse perspectives and editors that allow the creation and editing of:
 - Component Flows (Service Assemblies)
 - Message Definitions (MDL custom formats)
 - X12 variants
 - Message Mappings
- Testers for Message Definitions (MDL) and Message Mappings
- Enhanced Web Services support to import WSDL file to generate XSD schema and export WSDL file for HTTP server component to expose as Web Services

Common Services, Pre-packaged components and Interfaces

- User Point of Control (UPoC) framework to allow developers to perform callouts to alter the pre-defined flow of ChainBuilder ESB.
- Metadata support in all Binding Components and Service Engines and in Message Mappings
- Pre-packaged format definitions for EDI X12 standard versions of 003030, 003040, 003050, 003060, 003070 and 004010
- Command-line interfaces to deploy and run ChainBuilder ESB service assembly project
- AJAX based web interface to administrate, monitor and control the JBI components, Service Units, Service Assemblies and ChainBuilder ESB server.

- AJAX based web interface to view server logs, server statistics and error data base.

Each of above features will be covered in this Reference Guide document or other ChainBuilder ESB documents.

2. Installation

2.1. Prerequisites

2.1.1. Hardware Recommendation

Performance needs dictate the hardware requirements. For best performance, a Pentium class processor is recommended with 1 or more GB of RAM. ChainBuilder ESB can take advantage of multiple CPUs as well as hyper-threaded and multi-core CPUs.

2.1.2. Software Requirements

ChainBuilder ESB requires Java Development Kit (JDK) 5.0. JRE alone is not sufficient; a full JDK must be installed. The JDK can be downloaded from <http://java.sun.com>.

The JAVA_HOME environment variable must be set to point to the location of the JDK. For example:

```
JAVA_HOME=C:\Program Files\Java\jdk1.5.0_09
```

2.2. Obtaining the Software

An open source version of ChainBuilder ESB can be downloaded from:
<http://download.chainforge.net>

This download is licensed under the common open source General Public License (GPL). The formal terms of the GPL license can be found in the license text file included with the software or on the GNU GPL site at: <http://www.gnu.org/copyleft/gpl.html>.

Alternately, a flexible and affordable ChainBuilder ESB commercial license is also available. For more information about alternative licensing for ChainBuilder ESB, contact Bostech Corporation at info@bostechcorp.com.

2.3. Installation Procedure for Windows

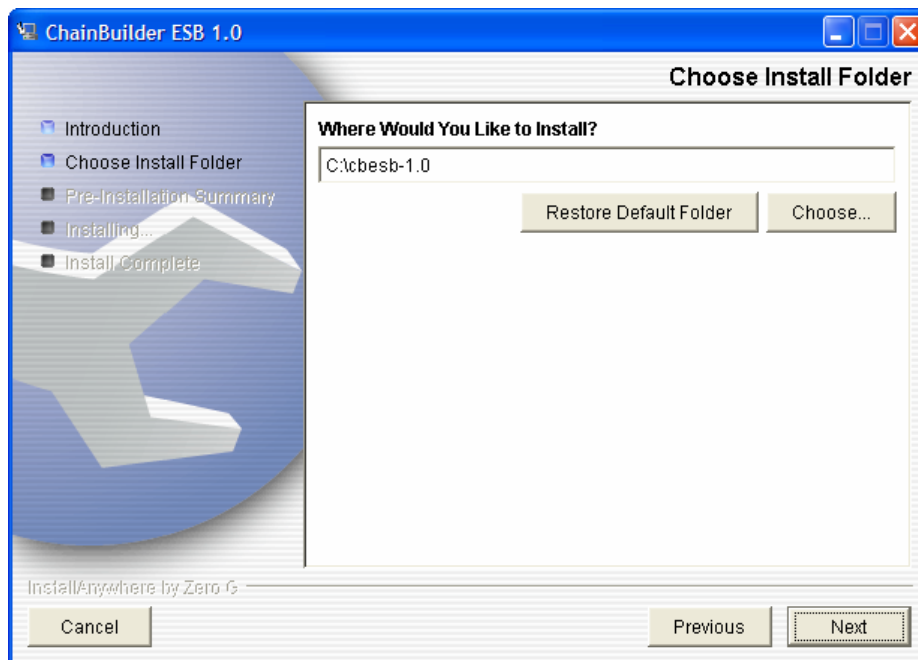
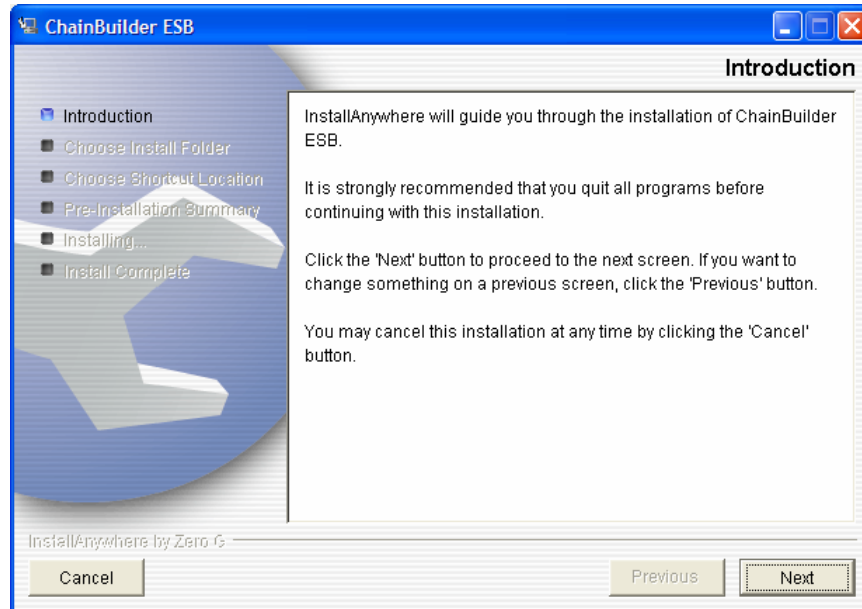
The download package is a zip file which contains the installer executable. The first step is to unzip the installer into a temporary location using a tool like WinZip or the built in compression utility in Windows XP or 2003.

The installer executable has the following naming convention, which includes the product release number.

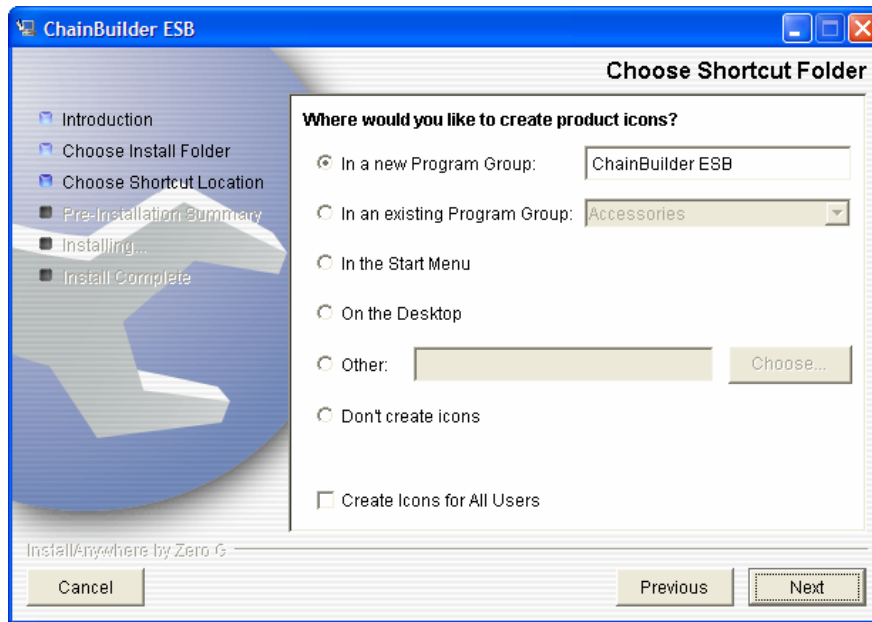
cbesb-n.n_xxxxxx_install.exe

(where n.n is the release number and xxxxxx indicates a build date)

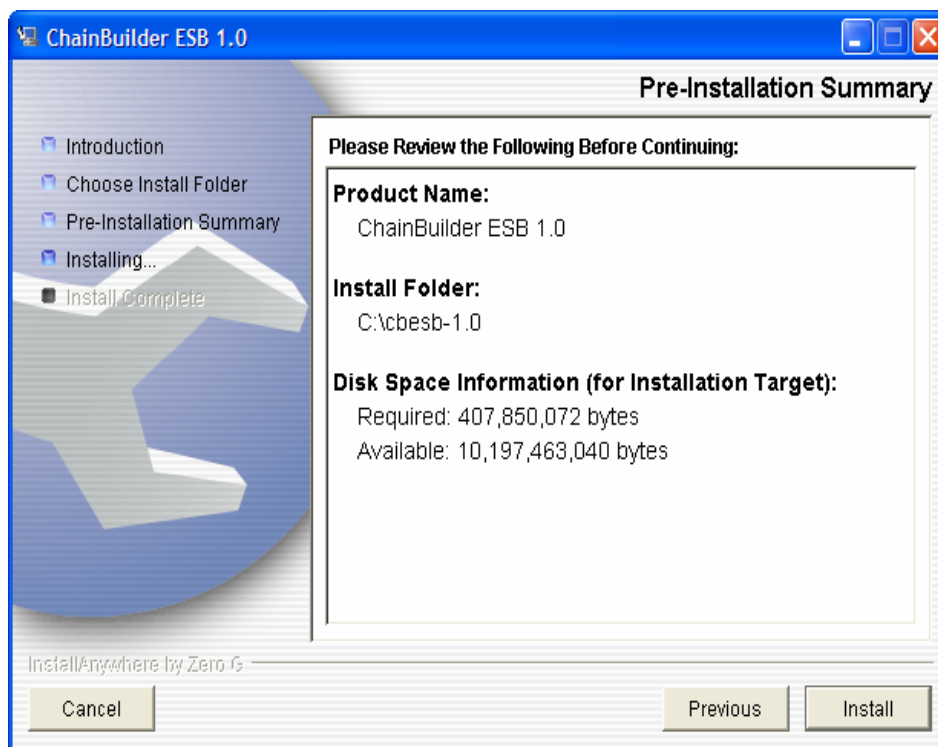
Close all other applications before installing. Double click on the executable to start the installation, there may be a slight delay as the installer decompresses to its full size.



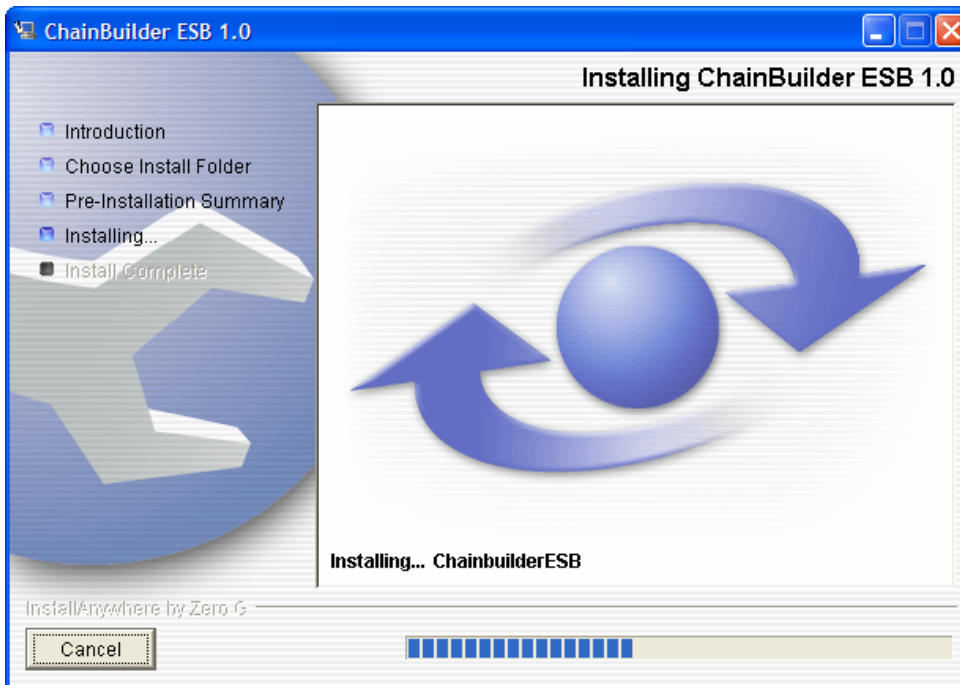
Choose the directory that you would like to install ChainBuilder ESB. The recommended location is `C:\cbesb-1.0`.



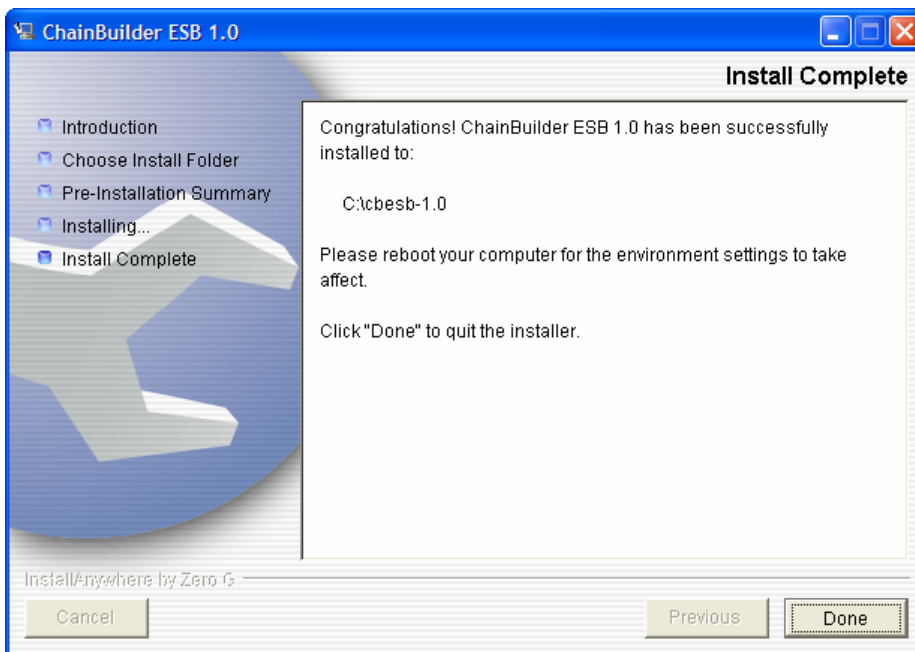
You may choose where the shortcuts will be placed.



Review the summary. If everything is correct, click Install.



The installer will take a few minutes to install all of the files.



When finished, click Done to exit the installer. Reboot your system to enable the changes to the environment to take affect.

There are five environment variables that are created or modified by the installer:

- **CBESB_HOME** – The user specified installation directory.
- **CBESB_CLASSPATH** – A Java classpath used by ChainBuilder ESB. This will reference directories inside CBESB_HOME needed by both the IDE and runtime.
- **ANT_HOME** – Provides the path to Apache Ant installed with ChainBuilder ESB.
- **SERVICEMIX_HOME** – Provides the path to ServiceMix, which is the JBI container used by ChainBuilder ESB.
- **PATH** – ChainBuilder ESB program directories are added to the system path.

2.4. Installation Procedure for Linux

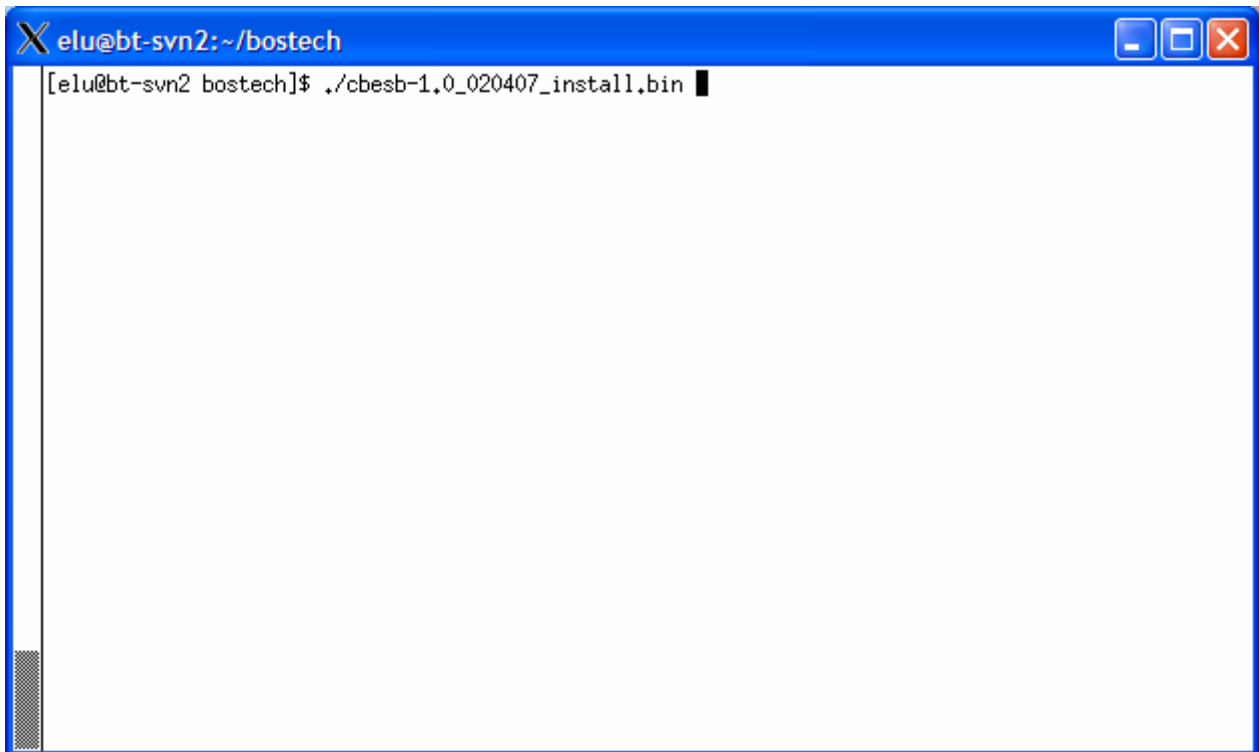
The download package is a bin file which is a self-extracted executable file which can run directly on Linux.

The installer executable has the following naming convention, which includes the product release number.

cbesb-n.n_xxxxxx_install.bin

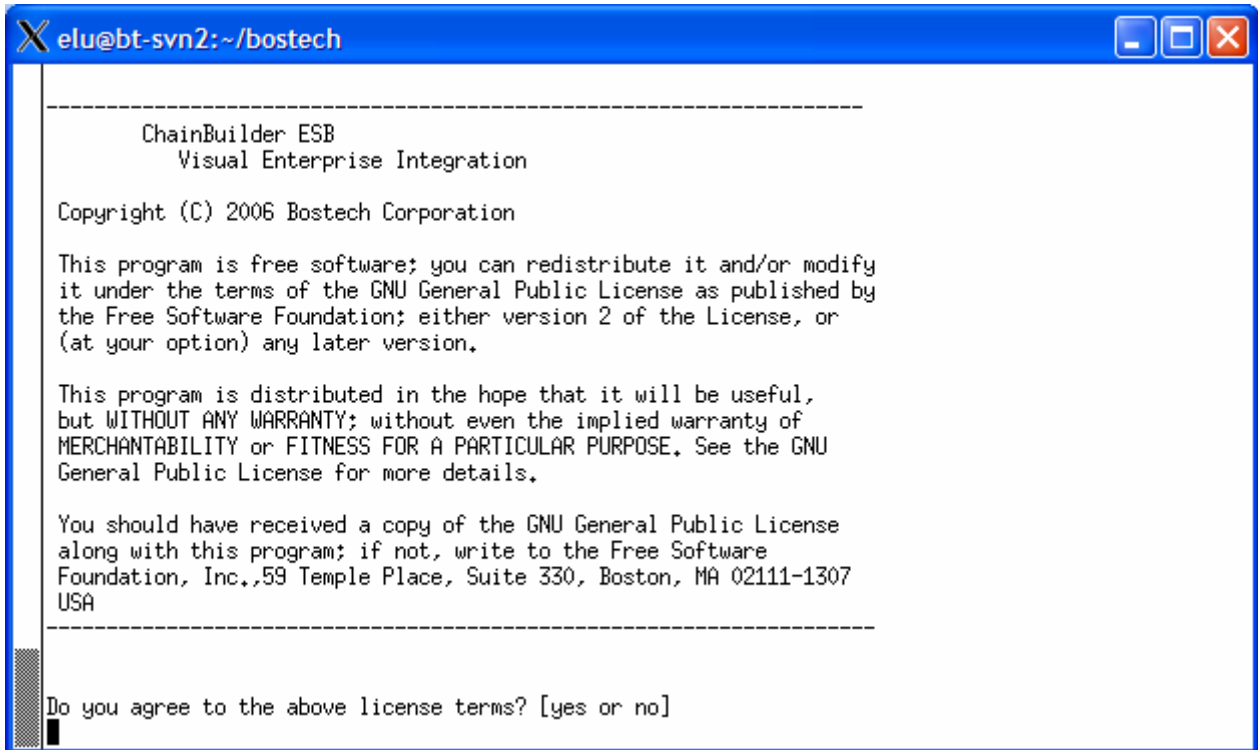
(where n.n is the release number and xxxxxx indicates a build date.)

Copy the above installer into the directory you want to have ChainBuilder ESB installed. (You need to make sure that you have write permission in this directory). Run the file from the Linux shell by typing its name at the command prompt.



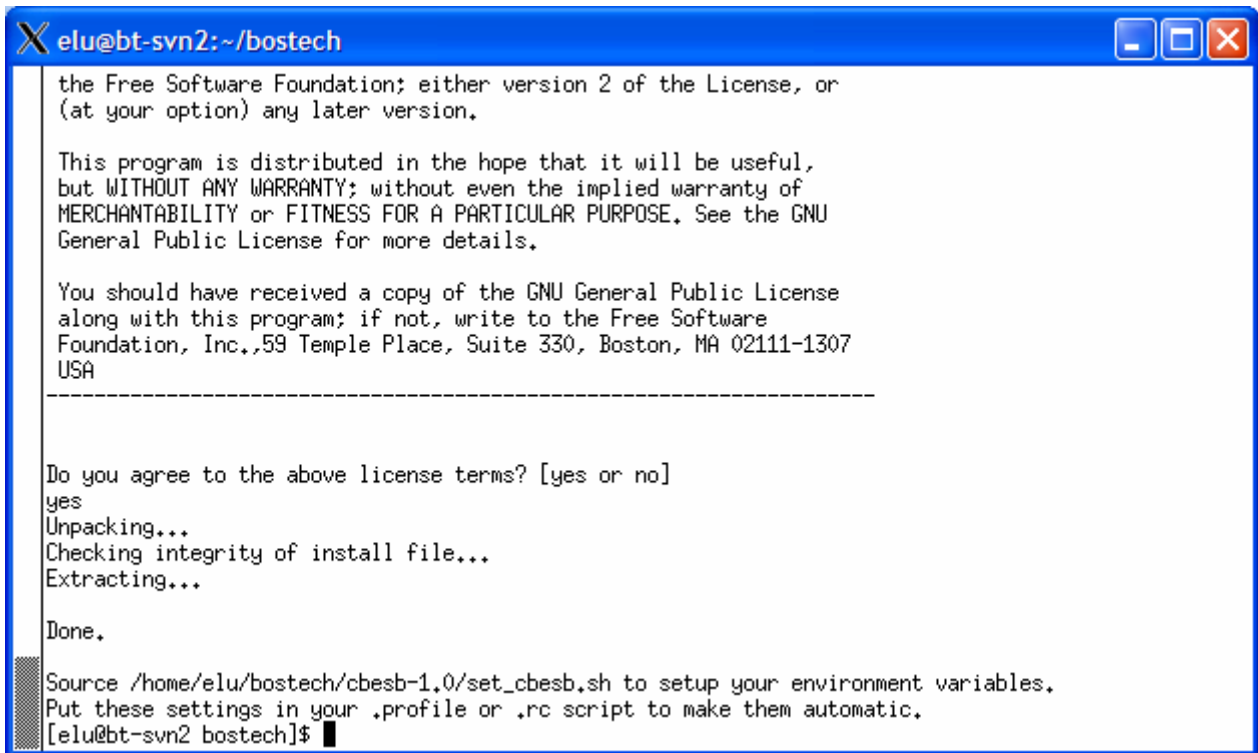
```
elu@bt-svn2:~/bostech
[elu@bt-svn2 bostech]$ ./cbesb-1.0_020407_install.bin
```


It then prompts for Copyright information and you acknowledge it by typing “yes”.

A terminal window with a blue title bar containing the text 'X elu@bt-svn2:~/bostech' and standard window control buttons. The terminal content is as follows:

```
-----  
ChainBuilder ESB  
Visual Enterprise Integration  
  
Copyright (C) 2006 Bostech Corporation  
  
This program is free software; you can redistribute it and/or modify  
it under the terms of the GNU General Public License as published by  
the Free Software Foundation; either version 2 of the License, or  
(at your option) any later version.  
  
This program is distributed in the hope that it will be useful,  
but WITHOUT ANY WARRANTY; without even the implied warranty of  
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU  
General Public License for more details.  
  
You should have received a copy of the GNU General Public License  
along with this program; if not, write to the Free Software  
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307  
USA  
-----  
  
Do you agree to the above license terms? [yes or no]  
█
```

It will then extract all files and install the product. The next screen shows the result when installation is done:



```
X elu@bt-svn2:~/bostech
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

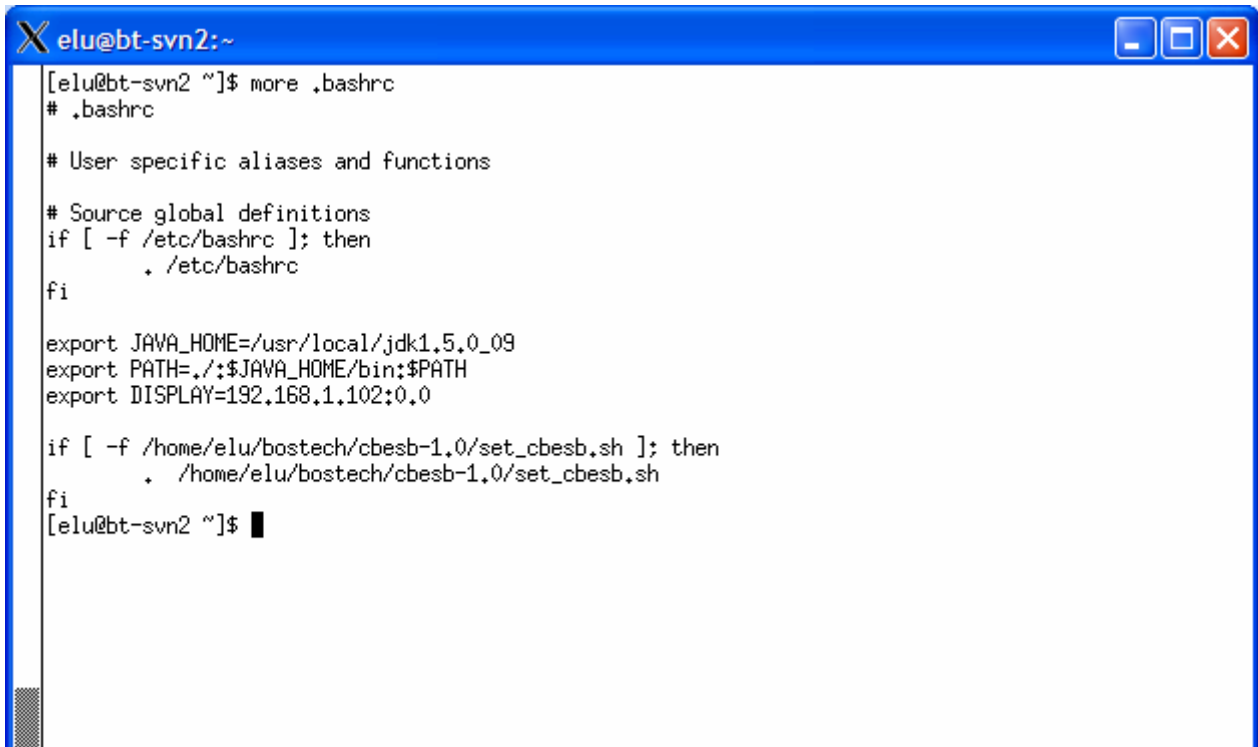
You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc.,59 Temple Place, Suite 330, Boston, MA 02111-1307
USA
-----

Do you agree to the above license terms? [yes or no]
yes
Unpacking...
Checking integrity of install file...
Extracting...

Done.

Source /home/elu/bostech/cbesb-1,0/set_cbesb.sh to setup your environment variables.
Put these settings in your .profile or .rc script to make them automatic.
[elu@bt-svn2 bostech]$
```

Follow the instruction as displayed on the screen and modify your Linux login profile (e.g., the `.bashrc` file if you use BASH shell). The following is an example of `.bashrc` file:



```
X elu@bt-svn2:~
[elu@bt-svn2 ~]$ more .bashrc
# .bashrc

# User specific aliases and functions

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

export JAVA_HOME=/usr/local/jdk1,5,0_09
export PATH=./:$JAVA_HOME/bin:$PATH
export DISPLAY=192.168.1.102:0.0

if [ -f /home/elu/bostech/cbesb-1,0/set_cbesb.sh ]; then
    . /home/elu/bostech/cbesb-1,0/set_cbesb.sh
fi
[elu@bt-svn2 ~]$
```

The `set_cbesb.sh` script will set up all of the necessary ChainBuilder ESB environment variables:

- **CBESB_HOME** – The user specified installation directory.
- **CBESB_CLASSPATH** – A Java classpath used by ChainBuilder ESB. This will reference directories inside `CBESB_HOME` needed by both the IDE and runtime.
- **ANT_HOME** – Provides the path to Apache Ant installed with ChainBuilder ESB.
- **SERVICEMIX_HOME** – Provides the path to ServiceMix, which is the JBI container used by ChainBuilder ESB.
- **PATH** – ChainBuilder ESB program directories are added to the system path.

You are now ready to use the Chainbuilder ESB by typing “`$CBESB_HOME/eclipse/eclipse`” from the Linux shell.

3. Running ChainBuilder ESB Server

ChainBuilder ESB supports basic command line tools for running the server and deploying artifacts. These provide a quick way for developers to test their applications. In production, the ESB server and an administration console server run as services in the background.

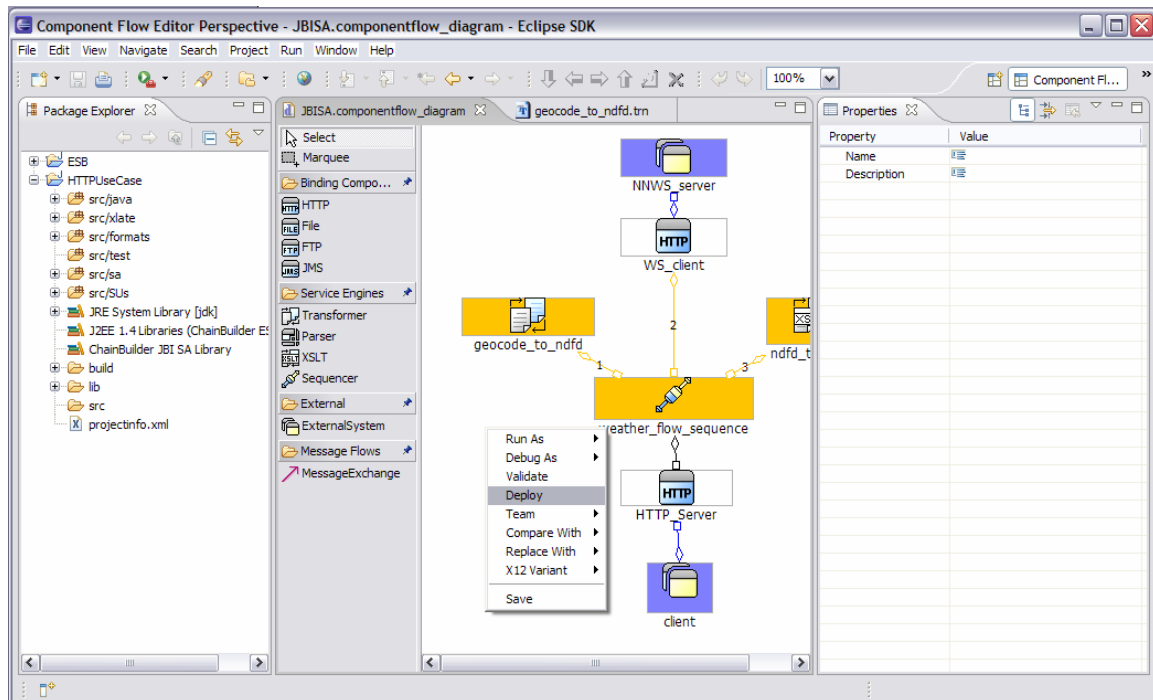
3.1. Command Line Interface

3.1.1. Installing Components

Components are automatically installed as required by the service assembly deployment.

3.1.2. Deploying Service Assemblies

First, construct the service assembly and service artifacts. This is done from the Flow Editor by right-clicking in the drawing canvas and selecting “Deploy”. The service assembly and service unit artifacts will be created in `src\sa` and `src\SUs` respectively in the project directory. The required files are also copied to the `CBESB_HOME\runtimes\test\SA_Proj_Name` directory.



The deployment can also be done from the command line. The command line interface for deploying a service assembly is:

```
➤ cbesb_deploy SA_Proj_Name
```

(where `SA_Proj_Name` is the name of ChainBuilder ESB Service Assembly project created using the ChainBuilder ESB IDE)

3.1.3. Starting and Stopping the Server

The `cbesb_run SA_Name` batch file runs a service assembly. It continues to run until terminated by control-C or closing the command window.

The command line interface for starting ChainBuilder ESB server is:

➤ `cbesb_run SA_Proj_Name`
(where `SA_Proj_Name` is the name of ChainBuilder ESB Service Assembly project created using the ChainBuilder ESB IDE)

When the server is run from the command line, its working directory is `CBESB_HOME\runtimes\test\SA_Proj_Name`. Relative file names are based here.

3.1.4. Monitoring Activity

Later releases of ChainBuilder ESB will have more robust monitoring tools. However, the server does listen on port 1099 for JMX clients. There are several JMX console tools, like JConsole, that can provide some level of monitoring.

The Java-2 Platform, Standard Edition (J2SE) 5.0 release includes a JMX monitoring tool, JConsole. JConsole monitors applications running on the Java platform and provides information on their performance and resource consumption. Please see [Sun's](#) documentation for more information on using this tool.

The following sections are instructions on configuring and using JConsole with ChainBuilder ESB.

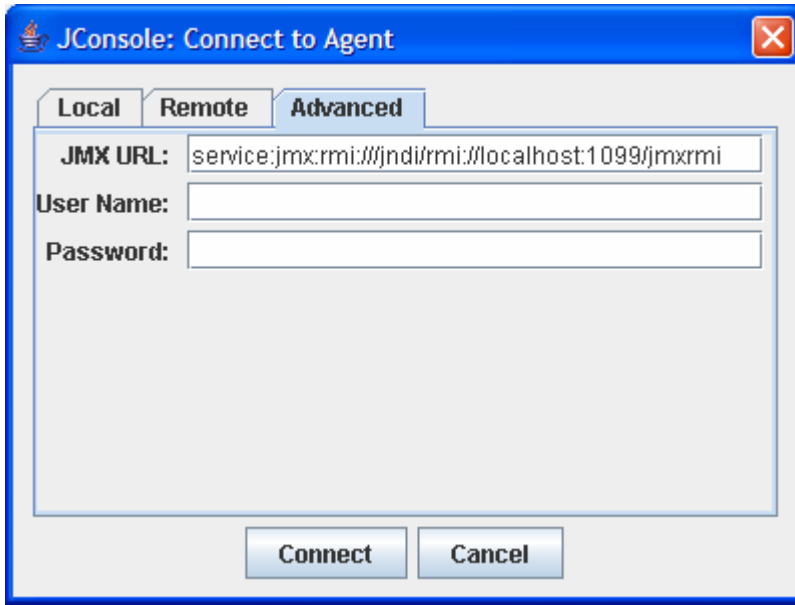
- Start Chainbuilder ESB server using the `cbesb_run` command.
- Start JConsole from a command shell. If you have JDK 1.5's bin directory in your PATH, you can invoke it with the tool name:

```
jconsole
```

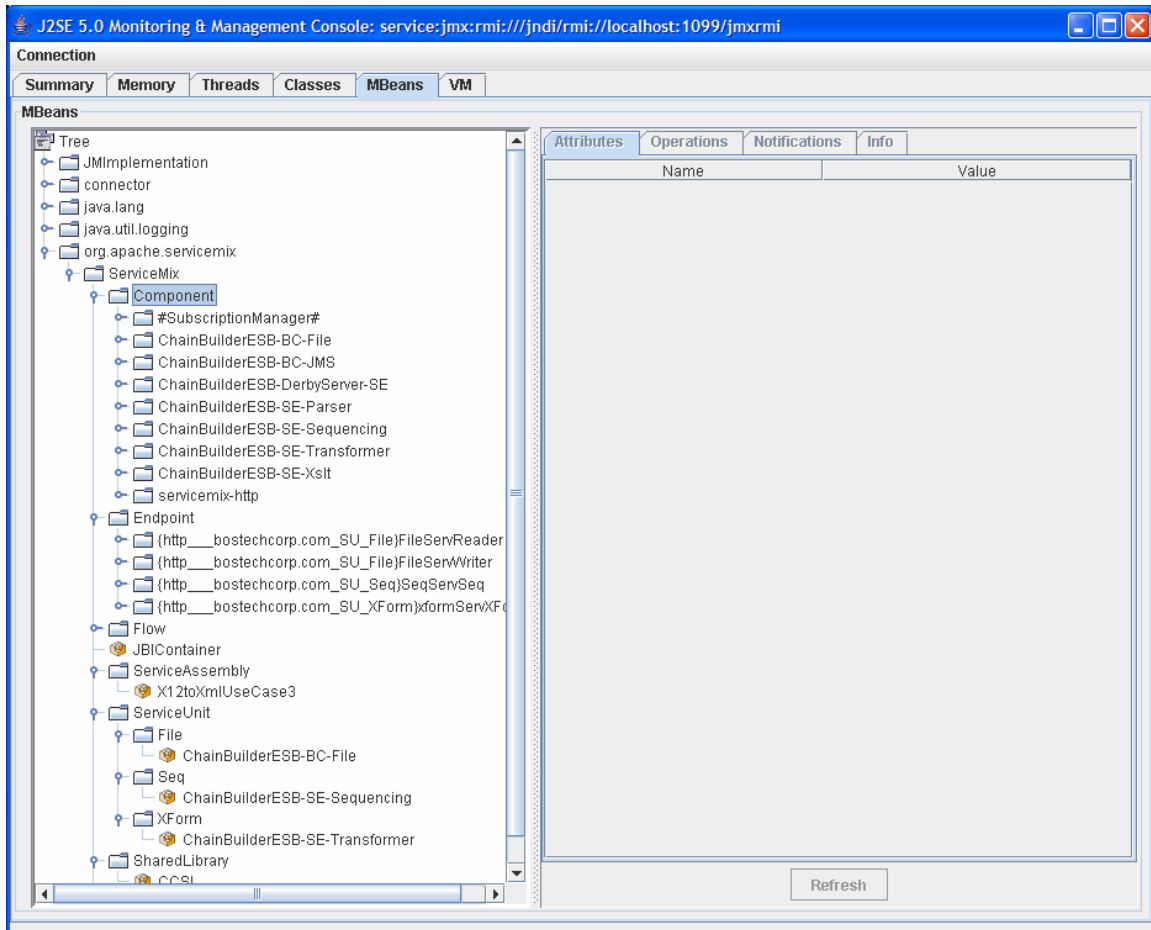
The JConsole window appears.

- Click on the "Advanced" tab. Enter the following URL in the "JMX URL" box

```
service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
```



- From the above screen, click the **Connect** box to connect to the ServiceMix container. The screen below shows the ChainBuilder ESB components including endpoint, service units and service assembly.



3.2. Running as a Service

The ChainBuilder ESB installer creates two services, one for the ESB server and one for the administrative console. These are set to automatically start when the system is rebooted. If you do not want the ESB server or admin console running as a service you should alter their properties in the control panel. In many cases it is easier to test a project using the command line.

3.3. Admin Console Web Interface

You can also use the ChainBuilder ESB Admin Console web interface to deploy, start and stop and monitor ChainBuilder ESB Service Assembly project. Please refer to the *Admin Console Guide* for additional information.

4. Configuration

4.1. Directory Structure for ChainBuilder ESB

All of the files installed with ChainBuilder ESB are placed in the directory specified at the time of installation. This directory is referred to by the environment variable that holds its location, %CBESB_HOME%. Inside the %CBESB_HOME% directory are the following subdirectories:

- **apache-servicemix** - The supported version of Apache ServiceMix used by ChainBuilder ESB.
- **bin** - Contains the executable files including batch files and shell scripts. This includes utilities that start/stop the server, deploy projects and display the contents of the error database.
- **brand** - Contains the version information as well as brand files and icons/images.
- **components** - Contains the JBI compliant components (Binding Components, Service Engines and Shared Libraries) provided as part of ChainBuilder ESB.
- **config** – Contains configuration files for ChainBuilder ESB.
- **db** - Contains an Apache Derby database that stores error information at runtime.
- **docs** - Contains all ChainBuilder ESB documentation.
- **eclipse** - The supported version of Eclipse used as the ChainBuilder ESB IDE. ChainBuilder ESB specific plug-ins are located in the eclipse/plugins directory.

- **formats** - Contains the format definitions included with ChainBuilder ESB, including EDI X12.
- **ideworkspace** - The default Eclipse workspace used by the ChainBuilder ESB IDE.
- **javacc** - The supported version of javacc used by ChainBuilder ESB.
- **lib and lib\ext** - These two directories contain java libraries in JAR format that ChainBuilder ESB uses. The lib directory contains JARs that need to be in the ChainBuilder ESB system class path. The JARs in lib\ext are made available to the ChainBuilder ESB component classloader. If there is a need to add some Java libraries to ChainBuilder ESB such as JMS jar files, X12 Finite State Machine (FSM) jar files, the compiled TRN jar files, they should be placed in the lib\ext directory to be picked up by ChainBuilder ESB automatically.
- **licenses** - Contains the license file of ChainBuilder ESB, as well as license files for each third party product used in ChainBuilder ESB.
- **log** - The ChainBuilder ESB log files are located in this directory. File-based logging is enabled by default. This can be changed in the log4j.properties file in the conf directory.
- **runtimes** – The ChainBuilder ESB server runtime environments. After a ChainBuilder ESB Service Assembly project is deployed, a directory structure will be created in this directory.
- **runtimes\console** – Stores the ChainBuilder ESB custom components and service assembly projects to be installed or deployed by the Admin Console.
- **runtime\server** – The root directory for ChainBuilder ESB runtime server.
- **runtimes\test** – The root directory for the ChainBuilder ESB command-line deployment and runtime interface. For each ChainBuilder ESB service assembly project you deploy using “cbesb_deploy” command, a sub-directory named after the Service Assembly project is created under this directory.
- **samples** - Contains example use case configurations.
- **tmp** - Temporary location for ChainBuilder ESB to store files and directories.
- **UninstallerData** - Contains uninstaller information to remove ChainBuilder ESB. It is only for Windows.
- **version** - Contains version information of ChainBuilder ESB.
- **wrapper** - Contains Windows and Linux executables to run ChainBuilder ESB as services.

4.2. Directory Structure for ChainBuilder ESB Project and Service Assembly Project

When a new ChainBuilder ESB project or Service Assembly project is created using ChainBuilder ESB IDE, the following directory structure is created for each project in %CBESB_HOME%\ideworkspace:

- **bin** - Contains the generated files by the ChainBuilder ESB IDE. You can ignore this directory
- **build** – The ANT script files for the project to build and deploy the project. This directory also contains the generated Java files for TRN files and the Finite State Machine (FSM) for tagged MDL files.
- **lib** – The library directory for UCM jar file, MAP jar file and FSM jar file.
- **lib\generated** – Not used.
- **lib\optional** - Contains optional jar files which can be used in Component Flow Editor and Map Editor for script component's POJO class, CCSL's UPoC POJO class, Map's User Defined class and Filter class. Bostech may develop some common user classes and distribute to users by putting them into this directory. Users may develop their own user classes and use them in many projects.
- **src** – The root directory for all source files.
- **src\formats** – The directory to store all MDL files and XSD schema files.
- **src\formats\x12** – The directory to store X12 variant files.
- **src\java** – Contains all Java files to be used for script component, Upoc, or User Defined class in Map. The files will be compiled and packaged into the CBESB_ucm_{ProjName}.jar file.
- **src\sa** – The directory for files used by the Component Flow Editor. You can click the {ProjName}.componentflow_diagram file to start the Component Flow Editor. The {ProjName}.componentflow file stores the information about all components. The “Deploy” function will create many zip files in this directory include the Service Assembly zip file {ProjName}.zip which will be deployed via the Admin Console.
- **src\scripts** – The directory to store all Groovy scripts used in Script component or CCSL UpoC.
- **src\SUs** – The directory to store all Service Unit artifacts created by Component Flow Editor.

- **src\tables** – The lookup file to be used by the lookup operation in Map Editor. The file is in Java XML property file format.
- **src\test** – Contains all test data files for the Format Tester and Map Tester. The result files of Map Tester are also saved into this directory by default.
- **src\wsdl** - Contains the WSDL files to be imported to generate XSD schema files in src\formats directory.
- **src\xlate** – Contains all the TRN file created by the Map Editor and the XSL files.

4.3. Configuration Files

You can modify the config files in %CBESB_HOME%\apache-servicemix\conf directory to change the ServiceMix runtime server. The debug_log4j.xml is the debug version of log4j.xml. The normal_log4j.xml is the normal version of the log4j.xml. You can copy debug_log4j.xml to log4j.xml if you want to have all debug information shown out in the cbesb_server.log file in %CBESB_HOME%\log directory.

Please refer to ServiceMix web site for additional information on how to modify other configuration files.

4.4. Logging

There are three log files in %CBESB_HOME%\log directory:

- **cbesb_server.log** – The log file generated by the ChainBuilder ESB server runtime. Writing to the log is handled by the Apache log4j package and the detail level is controlled by settings in %CBESB_HOME%\apache-servicemix\conf\log4j.xml.
- **sm_wrapper.log** – The log file created by the ChainBuilder ESB server Windows or Linux service. See the configuration file in %CBESB_HOME%\wrapper\apache-servicemix on how to control the logging level.
- **tc_wrapper.log** – The log file created by the ChainBuilder ESB Admin Console server Windows or Linux service. See the configuration file in %CBESB_HOME%\wrapper\tomcat on how to control the logging level.

4.5. Optional Jar Files

It is often necessary to have additional Jar files to be loaded by the ChainBuilder ESB server runtime in order to run certain component. For example, in order to use JMS component to connect to IBM Websphere MQSeries, you need to have ChainBuilder ESB to load the IBM

MQSeries jar files. For similar reason, in order to use the JDBC component to access Microsoft SQL Server, you need to have ChainBuilder ESB to load the Microsoft SQL JDBC driver jar file.

You can put such optional jar files into %CBESB_HOME%\apache-serviemix\lib\optional directory for ChainBuilder ESB server runtime.

5. CCSL Reference

5.1. Introduction

The ChainBuilder Common Services Layer (CCSL) is a module that plugs in between JBI components and the container. It provides a set of general services that can be useful for any component. Since CCSL is separate from and invisible to both the component and the container, CCSL can work with any JBI components and containers. CCSL currently provides three functions.

1. Allows for user scripting at various points of the exchange flow.
2. Provides an error handling mechanism that saves exchanges to a database in the event of processing exceptions.
3. Provides conversion of the message content between raw XML and an enveloped, multi-record format used by other ChainBuilder ESB components.

The following sections contain details regarding CCSL configuration and use.

5.2. Introduction to the CCSL Control Files

CCSL is controlled by two types of xml files.

1. The first type is located in the component and it contains component level settings. The component level file is part of the component package and it will not normally be changed. Section 5.11 shows an example of repackaging a component for CCSL.
2. The second type is located in the service unit and it contains endpoint level settings. The service unit level file is located in the service unit **META-INF** directory and is called **ccsl.xml**. This file is not required and if it is not found, the component level settings act as defaults. This shows an example of a **ccsl.xml** file.

```
<ccslSuConfig>
  <endpoints>
    <endpoint serviceNS="http://bostechcorp.com/SU/HTTP_Server"
      serviceLocal="weather_service"
      name="HTTPServer-8192"
      addRecord="false"
      stripRecord="true"
      saveErrors="true">
      <upoc context="presend" type="groovy" class="httpUPOCs.groovy"
        method="HTTTPresend"/>
    </endpoint>
  </endpoints>
</ccslSuConfig>
```

```

    </endpoint>

    <endpoint
        serviceNS="http://www.weather.gov/forecasts/xml/DWMLgen/
        wsdl/ndfdXML.wsdl"
        serviceLocal="ndfdXML"
        name="ndfdXMLPort"
        addRecord="true"
        stripRecord="true"
        saveErrors="false">
    </endpoint>
</endpoints>
</ccslSuConfig>

```

Each endpoint tag corresponds to an endpoint in the service unit. Each upoc tag represents a user script. There are several points where user scripts can occur so there can be multiple upoc tags for one endpoint.

5.3. ChainBuilder ESB Data Envelope Format

To understand CCSL attributes, we start with the ChainBuilder ESB Data Envelope Format. Most ChainBuilder ESB components use a data envelope format for the message content. This allows a single message to contain multiple records with mixed content types. The message content consists of a data envelope which may contain one or more data records. Each data record has a type: XML, String or Byte. String and Byte records have their content stored as attachments and XML records have their content stored inline.

Non-ChainBuilder ESB Components do not use this envelope format. To compensate we added a feature to CCSL that converts between our envelope format and raw XML. The following shows a basic data envelope format message.

```

<?xml version="1.0" encoding="UTF-8"?>
<DataEnvelope>
<XMLRecord>
<ResultSet xmlns="urn:yahoo:maps"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:yahoo:maps
http://api.local.yahoo.com/MapsService/V1/GeocodeResponse.xsd">
<Result precision="zip">
<DateToday>2006-11-11</DateToday>
<Latitude>40.2483</Latitude>
<Longitude>-83.3671</Longitude>
<Address/>
<City>MARYSVILLE</City>
<State>OH</State>
<Zip>43040</Zip>
<Country>US</Country>
</Result>
</ResultSet>
</XMLRecord>
</DataEnvelope>

```

5.4. ChainBuilder ESB sendMessage Envelope

The sendMessage envelope is introduced to support the Web Services in ChainBuilder ESB. It is designed so it can work with any Binding Component, but for now will only be used with the HTTP component. It works in conjunction with the DataEnvelope format described above. The CCSL API intercepts the in message in JBI MessageExchange as it goes from the component to the NMR to convert from the "sendMessage" xml format into the DataEnvelope structure. If a response message is sent back to the component, the out message should also be intercepted and converted from the DataEnvelope format into the "sendMessageResponse" XML format.

The following shows an example of the sendMessage formatted data within a SOAP wrapper:

```
<?xml version="1.0" encoding="UTF-8"?>
<SoapRequest xmlns="http://cbesb.bostechcorp.com/soap/1.0">
<SoapBody>
<sendMessage xmlns="http://cbesb.bostechcorp.com/soap/sendmessage/1.0">
<messageType>String</messageType>
<message>ST*270*D10000054~S2S*JE*BLOCK***Q~BHT*AB12*AB~HL*ABCDE**11~TRN
*AB*ABCDE12345~NM1*VN*2*HEALTHCARE DATA
EXCHANGE*****ZZ*00000000609~REF*F1*3.0~N4*HOPEWELL*VA*23860~PER*PZ**WP*
(610)219-1385~PRV*SB*ZZ*541779911~DMG*D8*19740529*M~INS*Y*
18~DTP*150*D8*20000322~DTP*151*D8*20000322~EQ*1~AMT*11*5000.00~REF*REF*
F1*3.0~SE*56*D10000054~
</message>
</sendMessage>
</SoapBody>
</SoapRequest>
```

The following shows an example of the sendMessageResponse formatted data within a SOAP wrapper:

```
<?xml version="1.0" encoding="UTF-8"?>
<SoapResponse xmlns="http://cbesb.bostechcorp.com/soap/1.0">
<SoapBody>
<sendMessageResponse
xmlns="http://cbesb.bostechcorp.com/soap/sendmessage/1.0"
xmlns:env="http://www.w3.org/2003/05/soap-envelope">
<messageType>XML</messageType>
<message>
<M270 xmlns="http://cbesb.bostechcorp.com/x12/004010">
```

```
<ST>
<ST00>270</ST00>
<ST01>D10000054</ST01>
<ST02/>
</ST>
<S2S>
<S2S00>JE</S2S00>
<S2S01>BLOCK</S2S01>
<S2S02/>
<S2S03/>
<S2S04>Q</S2S04>
<S2S05/>
<S2S06>
<C00/>
<C01/>
<C02/>
<C03/>
</S2S06>
<S2S07>
<C00/>
<C01/>
<C02/>
<C03/>
<C04/>
<C05/>
<C06/>
</S2S07>
<S2S08/>
<S2S09/>
</S2S>
<BHT>
<BHT00>AB12</BHT00>
<BHT01>AB</BHT01>
<BHT02/>
<BHT03/>
<BHT04/>
<BHT05/>
```

```
<BHT06/>
</BHT>
<loop2000>
<HL>
<HL00>ABCDE</HL00>
<HL01/>
<HL02>11</HL02>
<HL03/>
<HL04/>
</HL>
<TRN>
<TRN00>AB</TRN00>
<TRN01>ABCDE12345</TRN01>
<TRN02/>
<TRN03/>
<TRN04/>
</TRN>
<loop2100>
<NM1>
<NM100>VN</NM100>
<NM101>2</NM101>
<NM102>HEALTHCARE DATA EXCHANGE</NM102>
<NM103/>
<NM104/>
<NM105/>
<NM106/>
<NM107>ZZ</NM107>
<NM108>00000000609</NM108>
<NM109/>
<NM110/>
<NM111/>
</NM1>
<REF>
<REF00>F1</REF00>
<REF01>3.0</REF01>
<REF02/>
<REF03/>
```

```
<REF04>
<C00/>
<C01/>
<C02/>
<C03/>
<C04/>
<C05/>
</REF04>
</REF>
<N4>
<N400>HOPEWELL</N400>
<N401>VA</N401>
<N402>23860</N402>
<N403/>
<N404/>
<N405/>
<N406/>
</N4>
<PER>
<PER00>PZ</PER00>
<PER01/>
<PER02>WP</PER02>
<PER03>(610)219-1385</PER03>
<PER04/>
<PER05/>
<PER06/>
<PER07/>
<PER08/>
<PER09/>
</PER>
<PRV>
<PRV00>SB</PRV00>
<PRV01>ZZ</PRV01>
<PRV02>541779911</PRV02>
<PRV03/>
<PRV04/>
<PRV05>
```



```
<C00/>
<C01/>
<C02/>
</PRV05>
<PRV06/>
</PRV>
<DMG>
<DMG00>D8</DMG00>
<DMG01>19740529</DMG01>
<DMG02>M</DMG02>
<DMG03/>
<DMG04/>
<DMG05/>
<DMG06/>
<DMG07/>
<DMG08/>
<DMG09/>
</DMG>
<INS>
<INS00>Y</INS00>
<INS01>18</INS01>
<INS02/>
<INS03/>
<INS04/>
<INS05/>
<INS06/>
<INS07/>
<INS08/>
<INS09/>
<INS10/>
<INS11/>
<INS12/>
<INS13/>
<INS14/>
<INS15/>
<INS16/>
<INS17/>
```

```
</INS>
<DTP>
<DTP00>150</DTP00>
<DTP01>D8</DTP01>
<DTP02>20000322</DTP02>
<DTP03/>
</DTP>
<DTP>
<DTP00>151</DTP00>
<DTP01>D8</DTP01>
<DTP02>20000322</DTP02>
<DTP03/>
</DTP>
<loop2110>
<EQ>
<EQ00>1</EQ00>
<EQ01/>
<EQ02>
<C00/>
<C01/>
<C02/>
<C03/>
<C04/>
<C05/>
<C06/>
</EQ02>
<EQ03/>
<EQ04/>
</EQ>
<AMT>
<AMT00>11</AMT00>
<AMT01>5000.00</AMT01>
<AMT02/>
<AMT03/>
</AMT>
<REF>
<REF00>REF</REF00>
```

```

<REF01>F1</REF01>
<REF02>3.0</REF02>
<REF03/>
<REF04>
<C00/>
<C01/>
<C02/>
<C03/>
<C04/>
<C05/>
</REF04>
</REF>
</loop2110>
</loop2100>
</loop2000>
<SE>
<SE00>56</SE00>
<SE01>D10000054</SE01>
<SE02/>
</SE>
</M270>
</message>
</sendMessageResponse>
</SoapBody>
</SoapResponse>

```

More of the sendMessage interface is discussed in the Web Service section.

5.5. Endpoint Attributes

This table shows the attributes for the endpoint tag in ccs1.xml.

Attribute	Description	Default
serviceNS	Service namespace associated with the endpoint.	None
serviceLocal	Service local name associated with the endpoint.	None
name	Endpoint name.	None
addRecord	Boolean, indicates if messages sent from this endpoint must be wrapped in a Data Envelope.	value from component level ccs1.xml
stripRecord	Boolean, indicates if messages received at this	value from

	endpoint must have the Data Envelope stripped.	component level ccsl.xml
useSendMessage	Boolean, indicates if the message received at this endpoint must have the sendMessage wrapper. The response message sent out from this endpoint will be wrapped by a sendMessageResponse. The sendMessage wrapper is explained in section of Web Services Support. Currently, only the HTTP component should use this.	value from component level ccsl.xml
saveErrors	Boolean, indicates if the message exchange should be sent to the error database when an exception occurs. The error database is explained in section 5.10.	value from component level ccsl.xml

Note: addRecord and stripRecord are considered component level attributes so they can be omitted for the endpoints. In general, ChainBuilder ESB components use the Data Envelope natively and the attributes will be false. Non-ChainBuilder Components will use raw XML and the attributes will be true.

5.6. UPOC Attributes

This table shows the attributes for the upoc tag in ccsl.xml.

Attribute	Description	Default
context	Indicates the scripting point (described in section 6).	none
type	Script type. The only type currently supported is “groovy”.	none
class	For groovy scripts, this is the file name.	none
method	Method to run.	none

5.7. User Scripting Points

User scripts can be inserted at specific points in the message exchange flow. Each of these points is identified by a name such as “presend”. This name is referred to as the context. The following table shows the valid contexts and explains them.

Context	Description
presend	The user’s script is run immediately before a call to send() or sendSync()
postsend	The user’s script is run immediately after a call to sendSync() returns. When sendSync() returns, the out message is available in the exchange so this script allows you to process the response.
postaccept	The user’s script is run immediately after an exchange is received from a call to accept().
start	The user’s script is run immediately after the service unit starts. The exchange is null for the start context.
stop	The user’s script is run immediately before the service unit stops. The exchange is null for the stop context.

5.8. The Groovy Script Interface

This section describes the interface between CCSL and the user's groovy script. Groovy is quite similar to Java and it is compatible with Java objects. Inputs to the user's script are passed as Java object arguments. The user's script must return a (possibly empty) Java LinkedList object.

This shows a fragment of a typical User script:

```
def HTTPPresend(log, context, componentContext, channel, exchange) {
    LinkedList sendList = new LinkedList()

    // user's code

    sendList.add(exchange)
    return sendList
}
```

Arguments:

[log] Logger log – The log object for the component associated with this endpoint.

[context] String context – The context as described in section 6.

[ComponentContext] Component Context – This is the jbi container's component context object for the component associated with this endpoint.

[channel] Delivery Channel– The jbi container's delivery channel object.

[exchange] Message Exchange– The message exchange that is being operated on.

You will notice that actual JBI objects are passed to the script. While the script writer must be familiar with JBI it allows for high flexibility in coding. Also, the method receives one exchange but it returns a list. It is permissible for the user code to return any number of exchanges and the script does not need to return the original exchange. This is useful for operations like splitting up envelopes where one message may turn into several.

Since the script has the ComponentContext and DeliveryChannel, it is possible for the script to make JBI calls on its own. For example, the script could do the send() call and return an empty exchange list. This also allows complex flows to be implemented completely inside of a user script.

Appendix A contains source code for a script and Appendix B shows the log output that it produces. The script is from samples\UseCase4. It takes in HTML from data and converts it to valid xml suitable for further processing in JBI, showing the power of user scripts. All JBI message content must be XML. However, the user script runs early enough that HTML form data can be handled by converting it to XML.

5.9. Applying the Script

This section describes modifying the service unit so that it will execute the presend script. Go to your service assembly project directory which will be located in %CBESB_HOME%\ideworkspace*your_project_name*. We will be modifying the HTTP_Server service unit. All of its files are in the src\scripts directory. Create a ccsl.xml file in the META-INF directory and make it look similar to this:

```
<ccslSuConfig>
  <endpoints>
    <endpoint serviceNS="http://bostechcorp.com/SU/HTTP_Server"
      serviceLocal="weather_service"
      name="HTTPServer-8192"
      addRecord="false"
      stripRecord="true"
      saveErrors="true">
      <upoc context="presend" type="groovy" class="httpUPOCs.groovy"
        method="HTTTPresend" />
    </endpoint>
  </endpoints>
</ccslSuConfig>
```

Next, create the httpUPOCs.groovy file in the current directory. CCSL will look for script files in the service unit main directory. You have added a script.

5.10. Deployment and Running

Deployment and running service assemblies that include scripts is no different than deployment and running service assemblies without scripts.

5.11. The Error Database

Message exchanges that throw an exception during processing are lost in JBI. With CCSL, exceptions in send(), sendSync() and in user scripts can be caught. Through the saveErrors attribute, CCSL can be instructed to save these exchanges to an error database.

We use Derby for the database manager and the error database is located at %CBESB_HOME%\db\errordb.

Until the database viewer is available in the GA release you will need some familiarity with SQL to view the data. First, you will need to create a batch file to run an interactive SQL session. Create something like this.

```
set CP=%CBESB_HOME%\eclipse\configuration\org.eclipse.osgi\bundles\8\1\cp\derby.jar;
%CBESB_HOME%\samples\UseCase4\wdir\sharedlibs\CCSL\version_1\derbyclient.jar;%CBESB_HOME%
\samples\UseCase4\wdir\components\ChainBuilderESB-DerbyServer-SE\version_1\derbytools.jar

java -cp %CP% org.apache.derby.tools.ij
```



```

    <path-element>lib/commons-collections-3.1.jar</path-element>
    <path-element>lib/jencks-1.3.jar</path-element>
    <path-element>lib/commons-logging-1.2.jar</path-element>
  </component-class-path>
  <bootstrap-class-name>org.apache.servicemix.jms.JmsBootstrap</bootstrap-
class-name>
  <bootstrap-class-path>
    <path-element>lib/servicemix-jms-3.0-incubating.jar</path-element>
    <path-element>lib/concurrent-1.3.4.jar</path-element>
    <path-element>lib/commons-collections-3.1.jar</path-element>
    <path-element>lib/jencks-1.3.jar</path-element>
    <path-element>lib/commons-logging-1.2.jar</path-element>
  </bootstrap-class-path>
  <shared-library version="3.0-incubating">servicemix-shared</shared-library>
  <shared-library>CCSL</shared-library>
</component>
</jbi>

```

1. Remember the existing component class name. It is used later.
2. Change the component class name as shown. Every CCSL-enabled component uses the CcsComponent class. The CcsComponent class then loads the real component during initialization. CCSL then acts as a proxy between the container and the real component.
3. Add the class path and shared library entries as shown.

Next create a META-INF\ccsl.xml file. This file specifies the real component class and the component level CCSL default settings.

```

<?xml version="1.0" encoding="UTF-8"?>
<ccslComponentConfig>

  <componentClassName>org.apache.servicemix.jms.jmscomponent</componentCl
assName>
    <defaultSettings saveErrors="true" addRecord="false"
stripRecord="true" useSendMessage="false"/>
</ccslComponentConfig>

```

Then copy ccsl-base.jar into the temporary directory and zip up the temporary directory as the new CCSL-enabled component. The CCSL shared library must be installed when you run the component. All ChainBuilder ESB deployments already have the CCSL shared library installed.

6. Transformation Reference

The transformation support is a key feature in ChainBuilder ESB. The ChainBuilder ESB IDE Map Editor is used to define a map which will be saved into an XML formatted Transformer Control File (TRN) file. Then, the control file is run through a compiler that converts it to java code. The resulting class is deployed to a transformer service engine for runtime operation.

The transformer control file corresponds closely with the transformation operations displayed in the Map Editor. In this section, we will describe some of the important concepts in the transformer control file.

6.1. Transformation Source and Target

A transformation is defined by the source and target formats and a set of operations. The source and target formats are stored in the control file in the “formats” tag.

```
<formats>
  <input format="xsd" name="informat.xsd" root="Message"/>
  <output format="xsd" name="outformat.xsd" root="o4oput10"/>
</formats>
```

The *format* attribute may be “mdl” for Message Definition Language, “xsd” for XML schema, or “x12” for EDI X12. Other formats may be added later. For DTD, you may use a utility to convert DTD to schema. The input and output elements for xsd formats contain a root attribute. This specifies the global element in the schema that all paths originate from. Paths do not contain this root element. It is assumed for all paths.

6.2. Transformation Operations

Here we describe the transformation operations. A list of operations is stored in the transformer control file in the *operationList* tag. Each individual operation is contained in an *operation* tag. Every operation has a type and a list of parameters (some operations, like “else”, do not have any parameters). Some operations also have a nested *operationList* tag. The following example shows some operations in the control file. Comment operations are a special case. They use XML comments as described below.

```
<operation type="built-in" name="iterate">
  <parameters>
    <source type="absolute" path=" InvoiceHeader/CartonHeader"/>
    <property name="context" value="CartonHeader"/>
  </parameters>
  <operationList>
    <!-- This defines a comment operation -->
    <operation type="built-in" name="iterate">
      <parameters>
```

```
<source type="context" name="CartonHeader"
path="CartonDetail"/>
```

6.2.1. Operation Types

There are three types of operations - “built-in”, “class” and “user”. The *type* attribute is required and it controls the internal behavior of the Transformation Engine.

The “built-in” operations are hard-coded into the Transformation Engine so there is a fixed set available. These have a name attribute which must come from the following list.

Built-In operation names	Description
Iterate	Defines a context that is associated with a repeating element in the source or target tree. The context is essentially a quick pointer into the data tree. An iterate can have both a source and target in which case the two are tied together. Iterate operations will contain a nested <i>operationList</i> .
Comment	Comment operations have special handling. XML comments that occur inside of an <i>operationList</i> tag but outside on any <i>operation</i> tags will be treated as comment operations. No escaping should be required for the comment text but double hyphens should not be permitted.
Begincomment	Start of comment section. Skip all following sibling operations until a Endcomment operation appears. This can be used to comment out a block of operations. It is equivalent to Java’s “/*” operation.
Endcomment	End of the comment section started by a Begincomment operation. It is equivalent to Java’s “*/” operation. A valid TRN file has to have matching Begincomment and Endcomment operation.
While	Performs the nested operations while an expression is true. Expressions are described in section 5. The expressions used in “while”, “if” and “elseif” must evaluate to a boolean type.
If	Performs the nested operations if an expression is true.
Elseif	Performs the nested operations if the preceding “if” or “elseif” was false and an

	expression is true. It must immediately follow an “if” or “elseif” operation.
Else	Performs the nested operations if the preceding condition was false. It must immediately follow an “if” or “elseif” operation.
Send	Generates an output message from the current output tree. It can either leave the tree as-is or clear it.
Suppress	Prevents an output message from being sent at the end of transformation.
Arithmetic	Performs arithmetic operations on multiple sources and saves the result to a target.

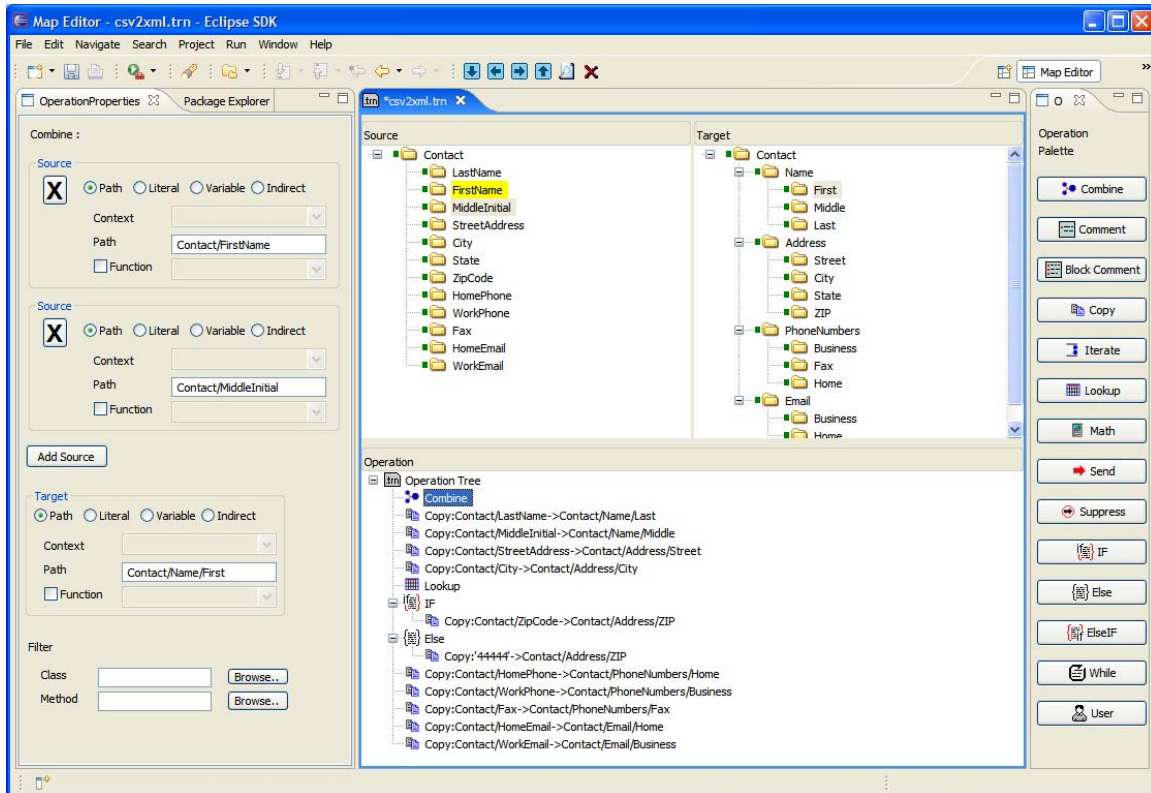
The “class” operations invoke methods in a java class. These have *name* and *class* attributes. These classes must implement the `ITransformationOperation` interface. The Transformer Engine creates an instance of the class for each occurrence of the operation and calls a method to perform the operation. The name is simply used for display purposes. We will initially support the class operations in this table.

Initial Class Operations	Description
Copy	Copy one source to one target.
Combine	Combine multiple sources into one target.
JDBC	Performs lookup operation using JDBC to a relational database on multiple sources and saves the result to targets.
Lookup	Uses the source to read a lookup table and saves the result to the target.

The “user” operations work exactly like class operations internally. The only difference is that user operations do not have a name, only a class. The main distinction is that “class” operations will appear to be an integral part of the product while “user” operations are user extensions.

6.2.2. Parameters in Transformation Operation

Every operation has parameters which control it. Currently, there are four types of parameters. These are “source”, “target”, “property” and “propertylist”.



Source and Target

Sources and targets are data addresses. They can reference the input and output trees, variables and literals. The source and target types are shown here.

Type	Description
Literal	A literal from the value attribute
Variable	A transformer variable from the name attribute. Variables do not need to be declared. They are automatically available. These are Java variables internally so the name must be a valid Java variable
Absolute	References data in the source and target trees
Context	References data in the source and target trees
Indirect	References data in the source and target trees

Here are some examples of sources and targets that reference the data trees.

1. Defines an absolute path into the source tree.

```
<source type="absolute"
path="Message/InvoiceHeader/CartronHeader" />
```

2. Defines a context based path into the source tree. The value refers to a context created by an iterate operation.

```
<source type="context" name="CartonHeader" path="CartonDetail" />
```

3. Defines an indirect address to a target. The path is retrieved from the variable var1.

```
<target type="indirect" name="var1" />
```

4. Defines an absolute path to the third element of CartonHeader.

```
<source type="absolute" path=" InvoiceHeader/CartonHeader[3]" />
```

5. Get the number of CartonHeader elements present in the tree,

```
<source type="absolute"
path="count( InvoiceHeader/CartonHeader )" />
```

The paths in sources and targets use an XPath-like syntax. Slashes separate elements, @ means attribute for XML and [xx] refers to a repetition value. The *count(path)* function evaluates to the number of repetitions of *path*. The *length(path)* function evaluates to the length of the data addressed by *path*. Other functions may be added later and they will use the same syntax with a name followed by the path in parenthesis.

6. Get value of the user variable *myvar*. Variables do not need to be declared. They are automatically created as needed.

```
<source type="variable" name="myvar" />
```

7. Get the length of the data stored in the user variable *myvar*.

```
<source type="variable" name="length(myvar)" />
```

Property

A property is a name, value pair. Properties are used to pass general settings to an operation. Here is an example property.

```
<property name="variable" value="var1" />
<ns:operation class="OpLookup" name="lookup" type="class">
  <ns:parameters>
    <ns:source type="absolute" path="Contact/State" />
    <ns:target type="absolute" path="Contact/Address/State" />
    <ns:property name="table"
      value="MiscTesting::src/tables/testLookUp.tbl" />
    <ns:property name="default" value="Unknown" />
  </ns:parameters>
</ns:operation>
```

Propertylist

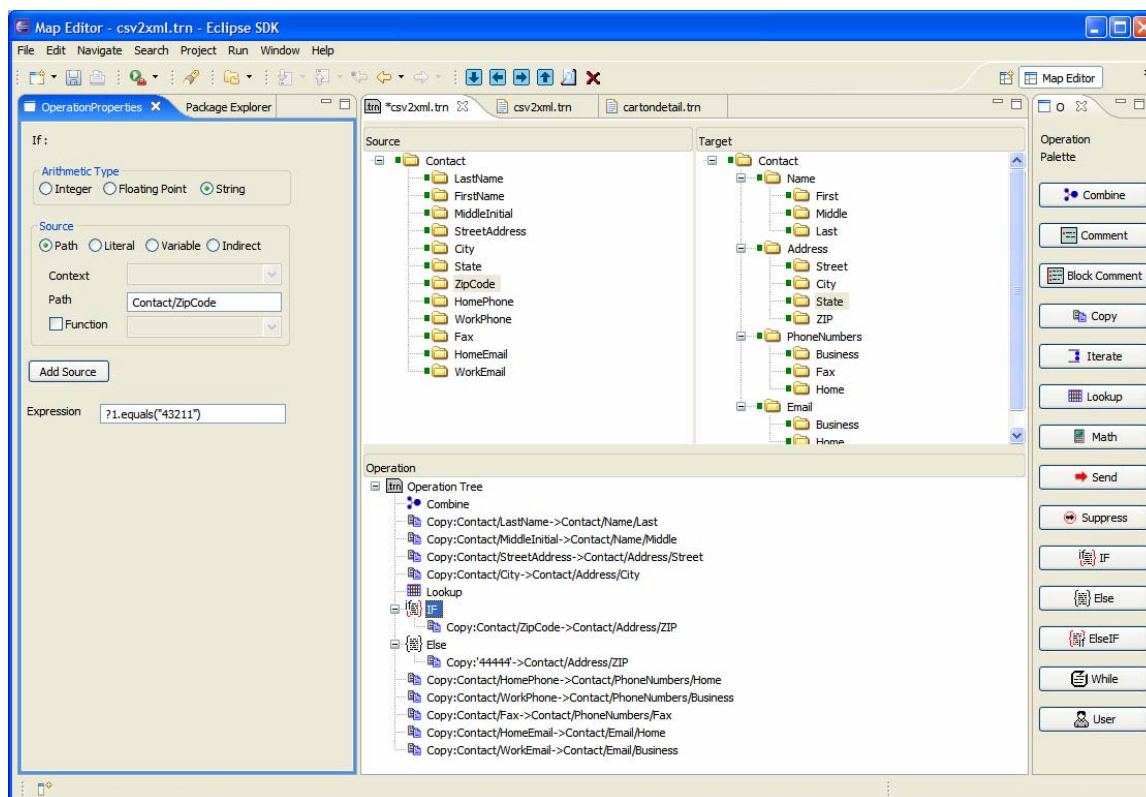
A propertylist is a named set of properties. Propertylists are used to pass a related set of properties to an operation. Here, a property list is used to define the class and method for a filter.

```
<propertylist name="filter">
  <setting name="class" value="MyClass" />
  <setting name="method" value="someMethod" />
</propertylist>
```

6.2.3. Expressions

Expressions are used by the arithmetic operation as well as “if”, “elseif” and “while”. The parameters for expressions are somewhat complex due to the large number of cases. Expression parameters are described here.

1. Arithmetic type – The document data and user variables are stored internally as strings. Arithmetic expressions will need to convert this data into an appropriate type for evaluation. Arithmetic evaluation can be done either in integer or floating point mode. The *arithmeticType* property specifies which type to use. It is also necessary to support string comparisons in conditions. Therefore, the *arithmeticType* may be “integer”, “float” or string.
2. Sources – One or more sources as described above may be specified.
3. Expression – The expression will be represented as a string where the location of the sources is indicated by a “?n” placeholder. This syntax is similar to JDBC statements. The expression string is stored in an *expression* property.
4. Arithmetic expressions may contain arithmetic operators and comparison operators. The resulting type must be appropriate for the intended target. That is, expressions used in an “arithmetic” operation must evaluate to a numeric type. Expressions used as conditions for “while”, “if” and “elseif” must evaluate to a Boolean type. Note that the *arithmeticType* property refers to the operands, not the result of the expression. The result type will depend on the operators used.
5. String type expressions also use the “?n” notation to indicate the position of the operands. However, the “?n” is followed by a period and a method specifier that is valid for String types. The result type for string expressions will depend on the methods and other operators used.



The above screen shot shows an example of If statement in the Map Editor with expression show in the left properties panel.

Example 1

This is an example of an expression used in an arithmetic operation. The result evaluates to the value of a user variable plus 1. This is stored back to the original variable, incrementing its value.

```
<operation type="built-in" name="arithmetic">
  <parameters>
    <source type="variable" name="myvar" />
    <target type="variable" name="myvar" />
    <property name="arithmeticType" value="integer" />
    <property name="expression" value="?1 + 1" />
  </parameters>
</operation>
```

Example 2

This is an example of a string type expression used as the condition in an “if” operation. The nested operations will be executed if the string stored in *myvar* equals “foo”. Notice how the double quotes are escaped with the entity reference "

```
<operation type="built-in" name="if">
  <parameters>
```

```

        <source type="variable" name="myvar"/>
        <property name="arithmeticType" value="string"/>
        <property name="expression"
value="?1.equals(&quot;foo&quot;)" />
    </parameters>
    <operationList>
        . . . nested operations
    </operationList>
</operation>

```

This shows how the operation properties would appear on the screen for the increment example described above.

6.2.4. Lookup Operation

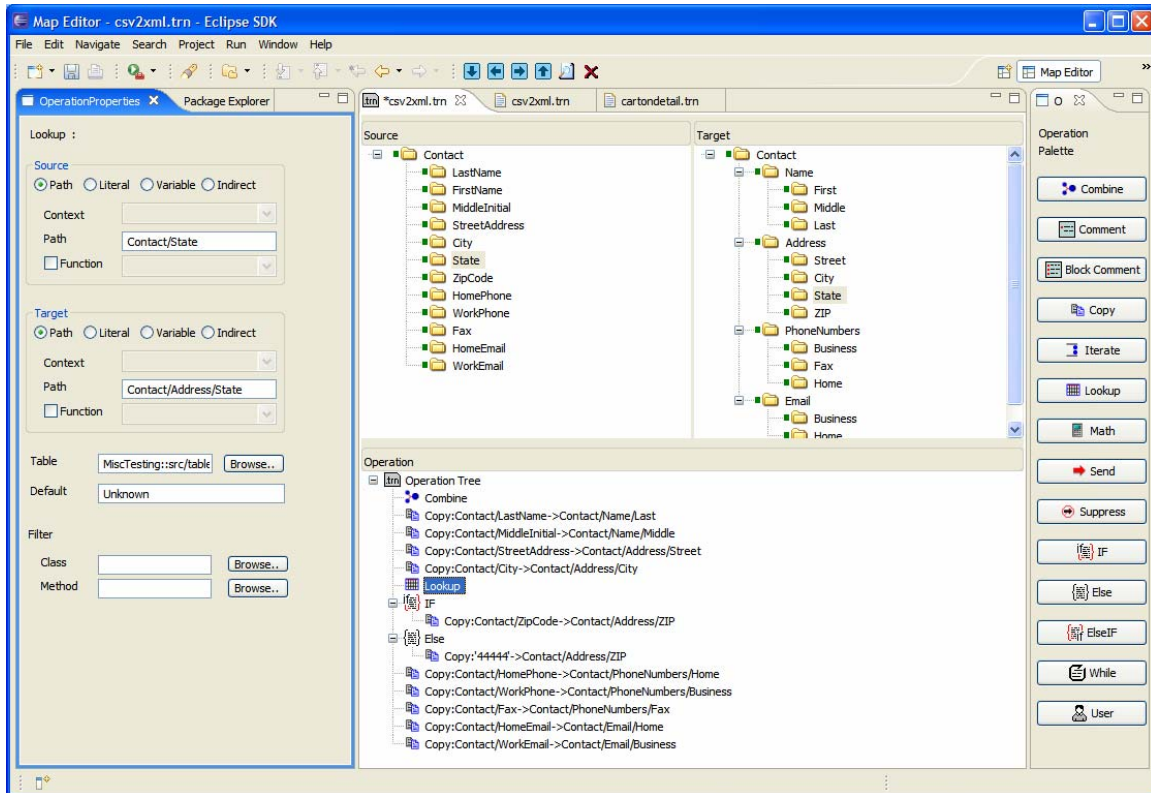
The lookup tables use the Java xml property file format. Here is an example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM
"http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>monthNameLookup</comment>
<entry key="Jan">January</entry>
<entry key="Feb">February</entry>
<entry key="Mar">March</entry>
<entry key="Apr">April</entry>
<entry key="May">May</entry>
<entry key="Jun">June</entry>
<entry key="Jul">July</entry>
<entry key="Aug">August</entry>
<entry key="Sep">September</entry>
<entry key="Oct">October</entry>
<entry key="Nov">November</entry>
<entry key="Dec">December</entry>
</properties>

```

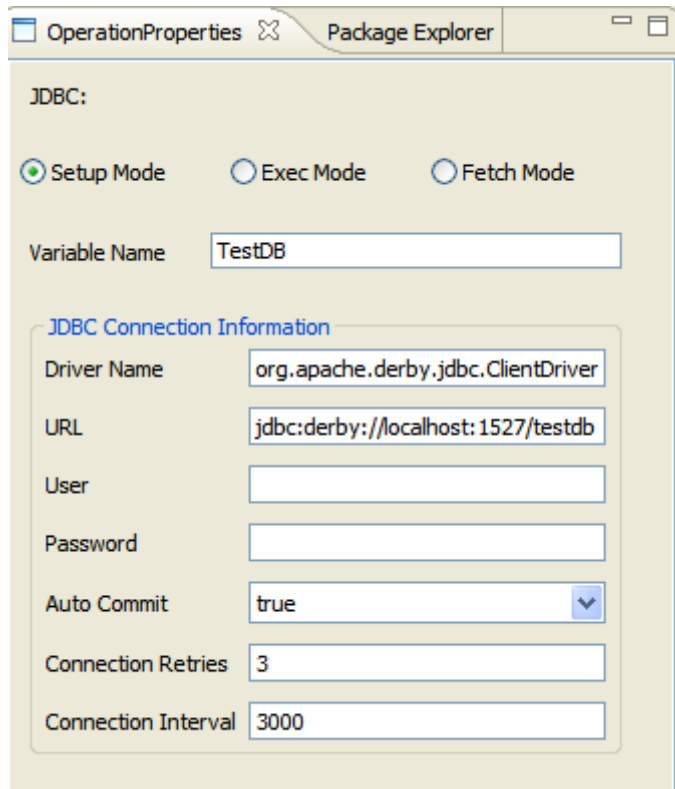
The lookup file would go into the `src/tables` directory of your project. When you specify the lookup table, use **`project_name::src/tables/file_name`**, similar to what is shown below.



6.2.5. JDBC Operation

The JDBC Operation is a special kind of User operation which allows users to interact with a relational database via standard JDBC SQL statements. There are three modes in the JDBC Operation. The first one is the “Setup” mode which allows users to define the setup information to a database resource. The setup information is saved into a map variable. The second mode is the “Exec” mode which will use a JDBC setup variable to execute the specified SQL statement. You typically setup a JDBC connection once and use it in the “Exec” mode as many times as you want. You can also setup multiple database connections and use them in the “Exec” mode simultaneously. The third mode is the “Fetch” mode which allows random access to the result set from a previously executed SQL statement. The database connections close automatically when the translation operations are complete.

When you drag a JDBC Operation from palette into the operation tree, the operation properties will have the options of “Setup”, “Exec” and “Fetch” at the top. The following screen shows an example of the “Setup” mode for the JDBC connection to Apache Derby.



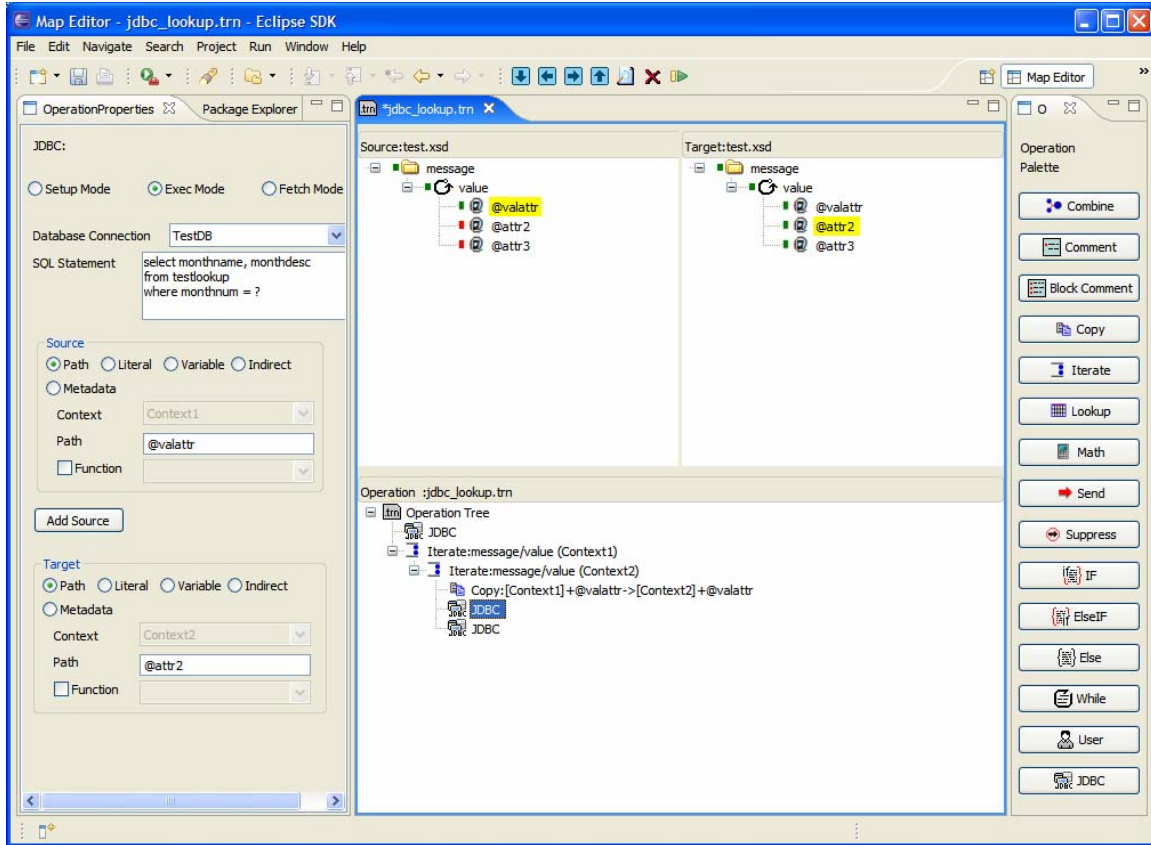
The screenshot shows the 'JDBC:' configuration window. At the top, there are three radio buttons: 'Setup Mode' (selected), 'Exec Mode', and 'Fetch Mode'. Below them is a text field for 'Variable Name' with the value 'TestDB'. A section titled 'JDBC Connection Information' contains the following fields:

- Driver Name: org.apache.derby.jdbc.ClientDriver
- URL: jdbc:derby://localhost:1527/testdb
- User: (empty)
- Password: (empty)
- Auto Commit: true
- Connection Retries: 3
- Connection Interval: 3000

In this example, the TestDB JDBC setup variable defined here will be appeared in the dropdown list in the “Exec” or “Fetch” mode in later JDBC Operation.

Note : As discussed in the section of “Optional Jar Files”, make sure to put the JDBC driver jar files into %CBESB_HOM%\apache-servicemix\lib\optional directory to use this feature.

In JDBC Operation’s “Exec” mode, users can select from a list of JDBC setup variables defined in the “Setup” mode. Users can enter a JDBC compliant SQL statement in a text box and also add one or more Source and one Target in the operation. The following screen shows an example of the “Exec” mode.

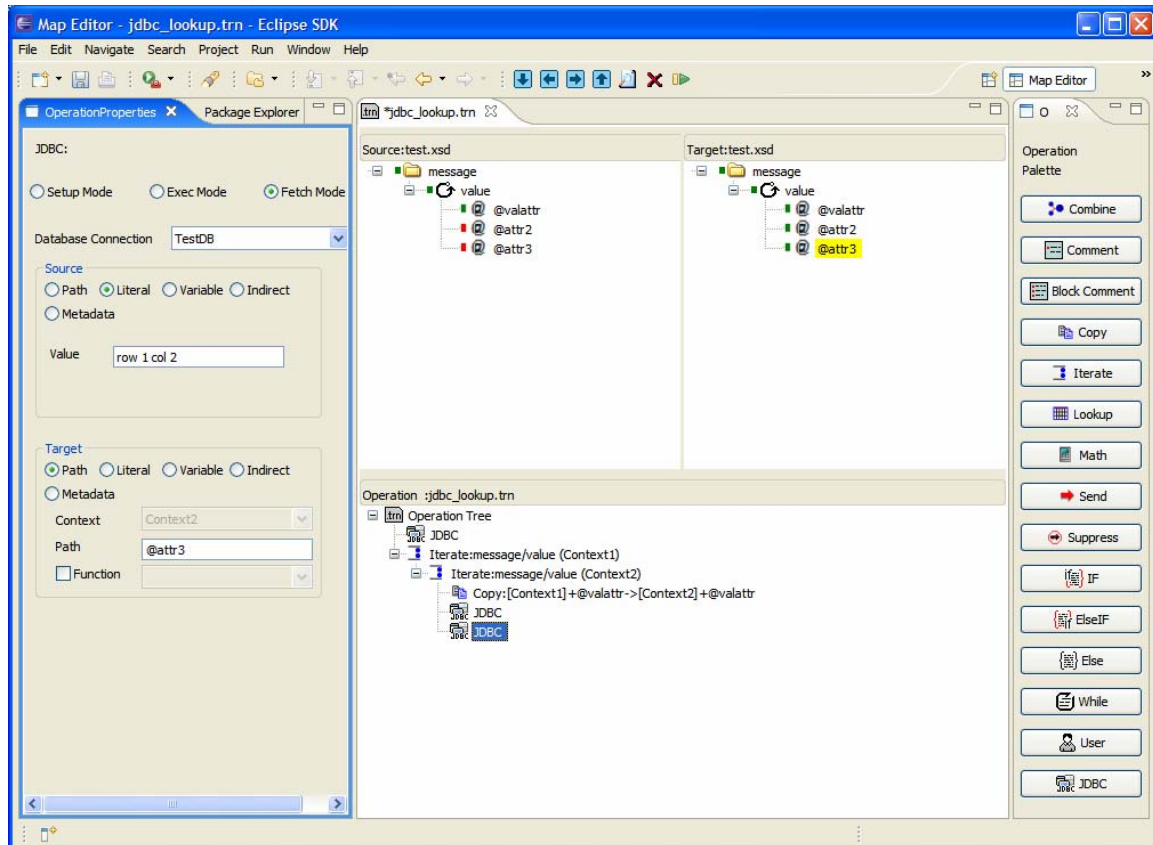


In this example, the source “@valattr” in “Context1” binds with the “?” specified in the SQL statement. The target “@attr2” in “Context2” binds with the first column “monthname” in the result columns in the Select statement.

As discussed above, the “Fetch” mode allows random access to the result set from a previously executed SQL statement. The “Fetch” mode must have exactly one source and one target. The source is a string that indicates the data to return from the result set. The requested data item will be returned in the target. The source may contain the literal string “count”. In this case, the number of rows in the result set is returned. The source may also contain a string of the form “row xxx column xxx”. In this case the specified row and column is returned from the result set. Row and column numbers start at 1.

The “Fetch” mode will be useful for efficiently processing select statements with multiple columns. It can also be used in conjunction with a while loop to process multiple rows.

The next screen shows an example of the “Fetch” mode continuing from the previous example.



In this example, the “Fetch” mode essentially binds the Target “@atr3” in “Context2” with the second column “monthdesc” in the result columns in the Select statement.

6.3. Transformation User Defined Classes

There are two types of user defined code supported by the transformer.

1. A user defined operation which acts just like a standard transformer operation.
2. A user defined filter which alters a single value as it flows through an operation.

6.3.1. User Defined Operation

The user defined operation needs to implement the `ITransformationOperation` interface. Here is the source code for `ITransformationOperation` interface:

```
package com.bostechcorp.cbesb.runtime.transformer.engine;

/*
 * This is the interface to implement for transformer "class" and "user"
 * operations
 */
public interface ITransformationOperation {
```

```
/*
 * NOTE: implementing classes must have a default constructor.
 */

public void addProperty( String name, String value );
/*
 * This is called once for each property immediately after the class is
 * instantiated. The class should save these settings to member variables.
 */

// implement propertylist later
// public void addPropertyList( PropertyList );
/*
 * This is called once for each propertylist immediately after the class
 * is instantiated. The class should save these settings to member
 * variables.
 */

public void initialize();
/*
 * This is called once before each message is transformed, the class
 * should initialize variables, etc.
 */

public void cleanup();
/*
 * This is called once after each message is transformed. Clean up any
 * resources
 */

public boolean process(String[] sources, String[] targets);
/*
 * This is called to perform the operation. Return false to skip
 * target processing after the operation completes.
 */
}
```

The process method returns a Boolean. If the return value is false then no processing of the targets is performed. This is essentially a way for internal logic to abort the operation. If the process method returns true then target Nodes associated with user variables are copied back to the appropriate variables. Target Nodes associated with the target data tree require no copying since they refer directly to the destination data and the process() method will have directly written to them.

There is an initialize method that will be called once before each transformation begins and a cleanup method that will be called once when a transformation is done. The operation instance may need to initialize or clean up since many messages may run through the same instance.

The Combine operation is implemented as a user defined operation behind the scenes.

Here is the code for the Combine operation:

```
package com.bostechcorp.cbesb.runtime.transformer.engine;

public class OpCombine implements ITransformationOperation {
public OpCombine() {}

public void addProperty(String name, String value) {
// no relevant properties for copy
}

public void addPropertyList() {
// TODO need to support filter propertyList
}

public void cleanup() {
// no cleanup
}

public void initialize() {
// no initialization
}

public boolean process( String[] source, String[] target) {
if(target[0]==null) target[0]="";
for(int i=0;i<source.length;i++){
target[0]=target[0].concat(source[i]);
}
return true;
}
}
```

6.3.2. User Defined Filter

A filter method is a simple string-to-string method call that can be associated with any data storing operation. The filter method is run on the target data immediately before it is stored. If a filter method returns null then nothing is stored. Filters must implement the `IFilterInterface` interface. Many filter methods may be placed into one class for convenience.

The following is a simple example of a Filter class:

```
public class Filter1 implements IFilterInterface {

    public String convert(String source) {

        if (source.equals("Male"))
            return "M";
        else if (source.equals("Female"))
            return "F";

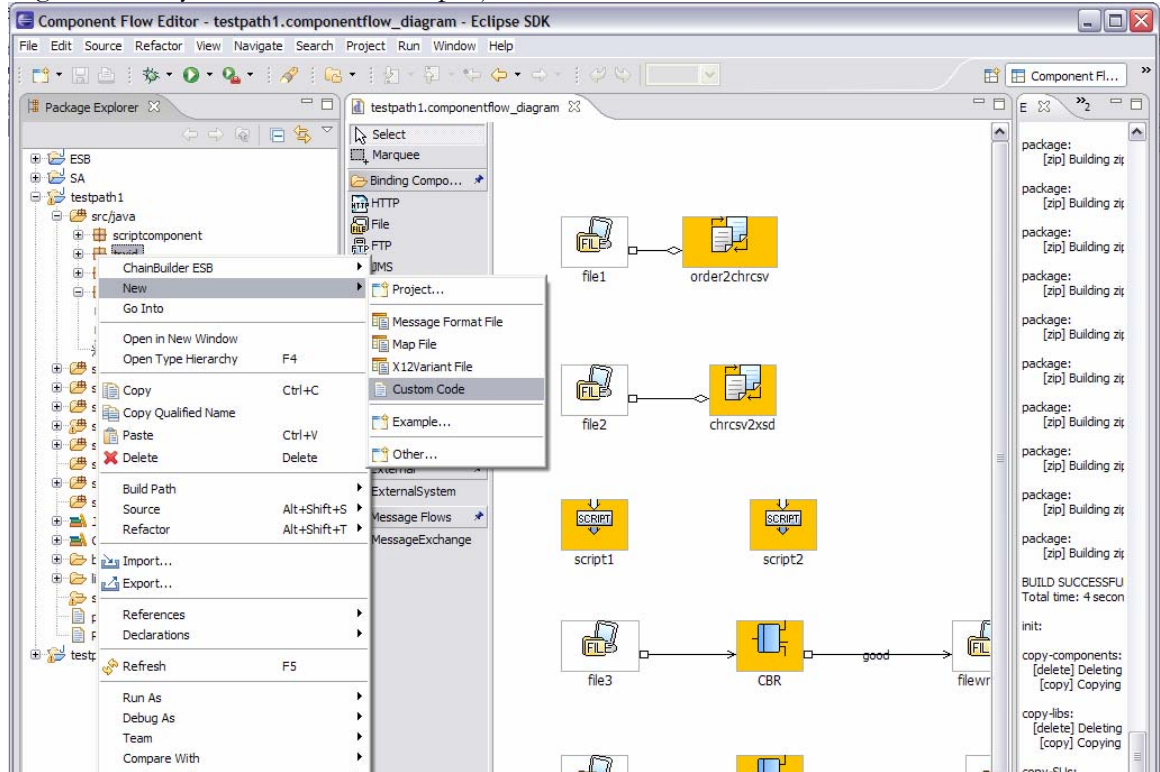
        return "U";
    }
}
```

```
}  
}
```

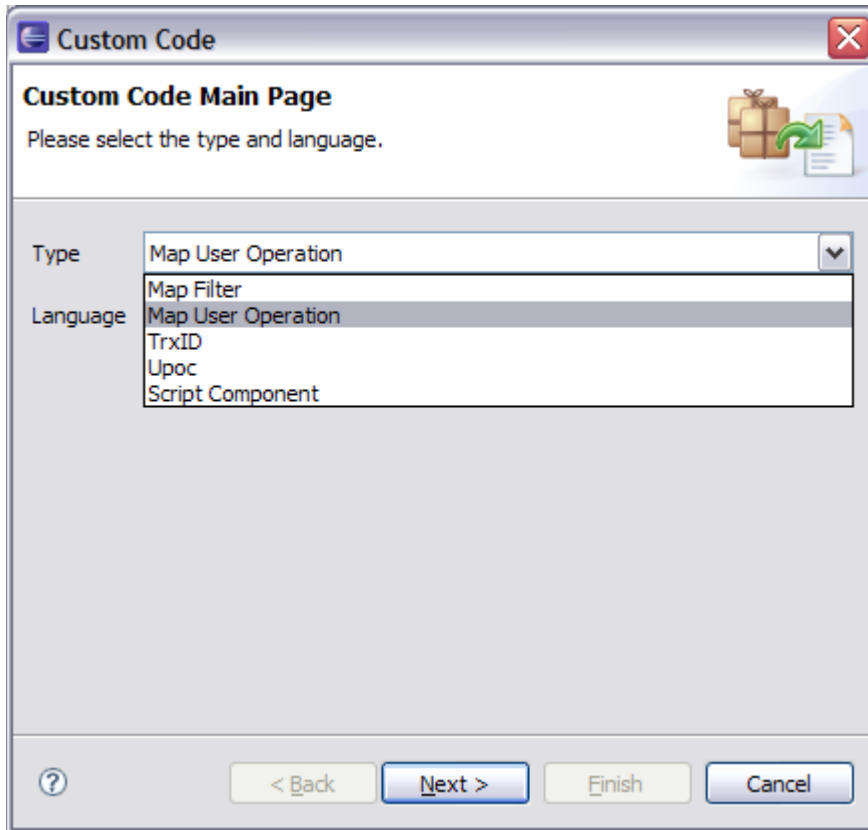
6.3.3. Steps to Create and Use User Defined Classes

1. Create a package for your user operation classes.
Right-click on src/java in the package explorer and select “New→Other→Java→Package”. Enter a name.

2. Create a Java class for your operation.
Right-click anywhere inside of the project and select “New→Custom Code”



Then select “Map User Operation” for the type and press “Next”



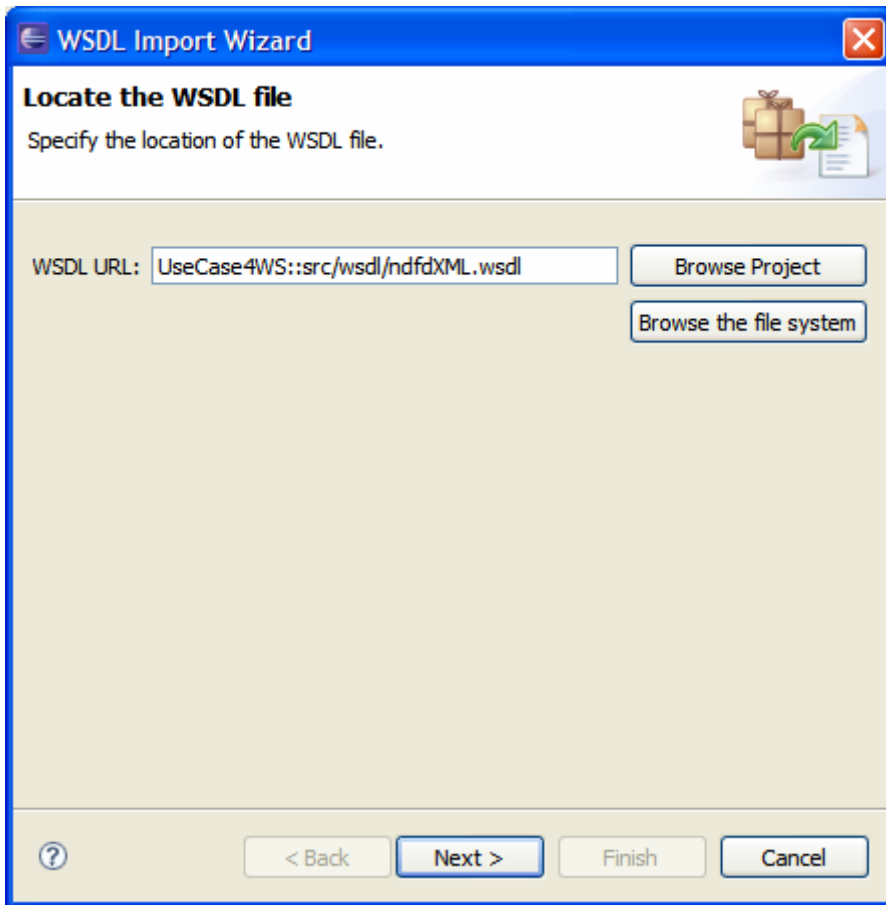
Next, select a package and enter a class name. The system creates a .java file with the basic skeleton code. You can edit this file to create your user operation.

7. Web Services Support

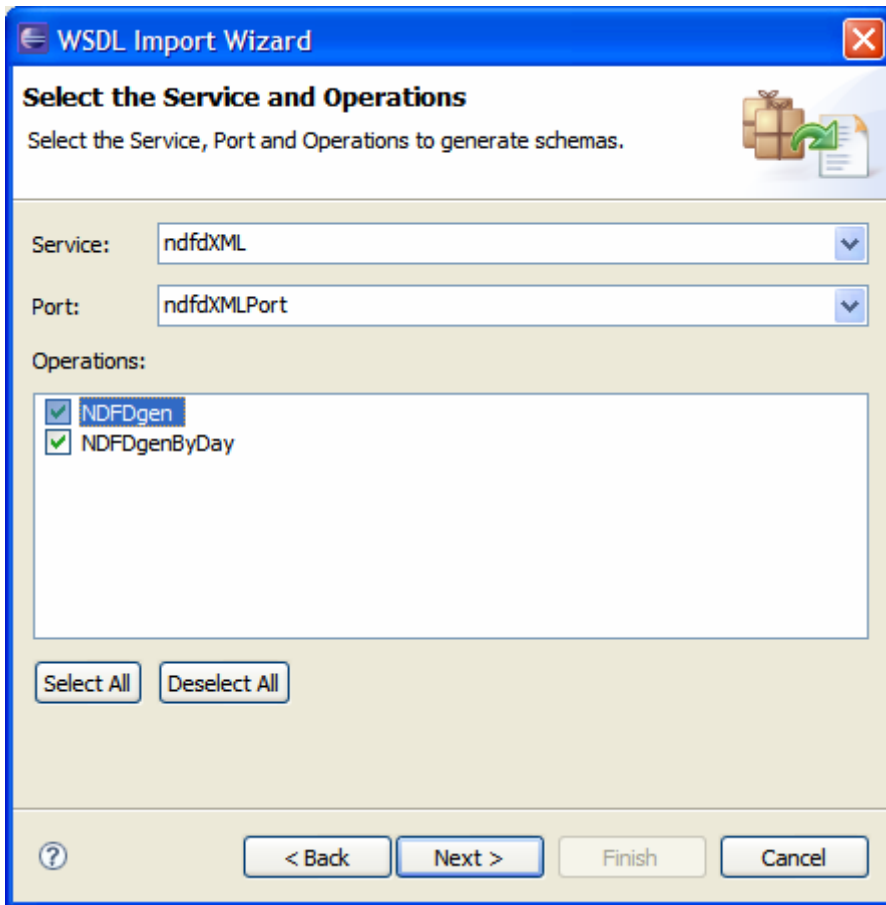
7.1. WSDL Import Wizard

To act as a web service client, the WSDL file of the web service must be provided. Using this WSDL file, the IDE can create schema definitions of the messages supported by the service. This function is provided by the WSDL Import wizard. To launch the wizard, you right click on a service assembly project context sensitive menu called “ChainBuilder ESB” and select the sub-menu called "WSDL Import Wizard".

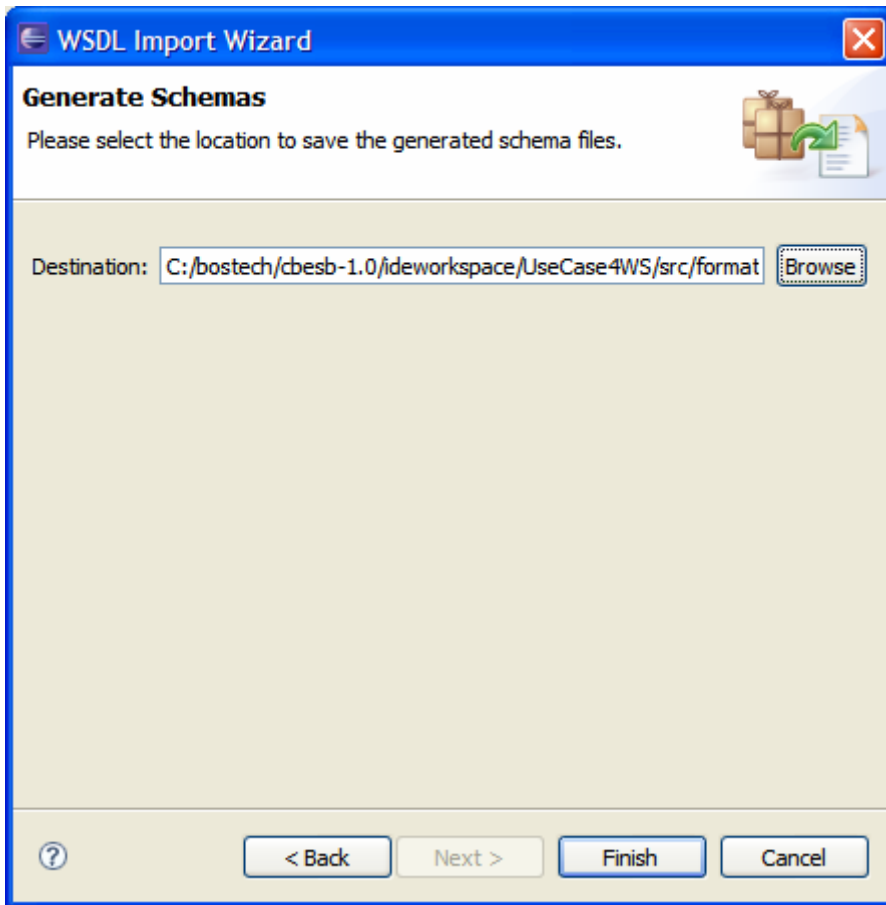
The next screen shows the first step to select a WSDL file. You can select from the project under “src\wsdl” or from local file system.



The next step shown in the screen below is to select Service, Port and Operations defined in the WSDL to be exported.

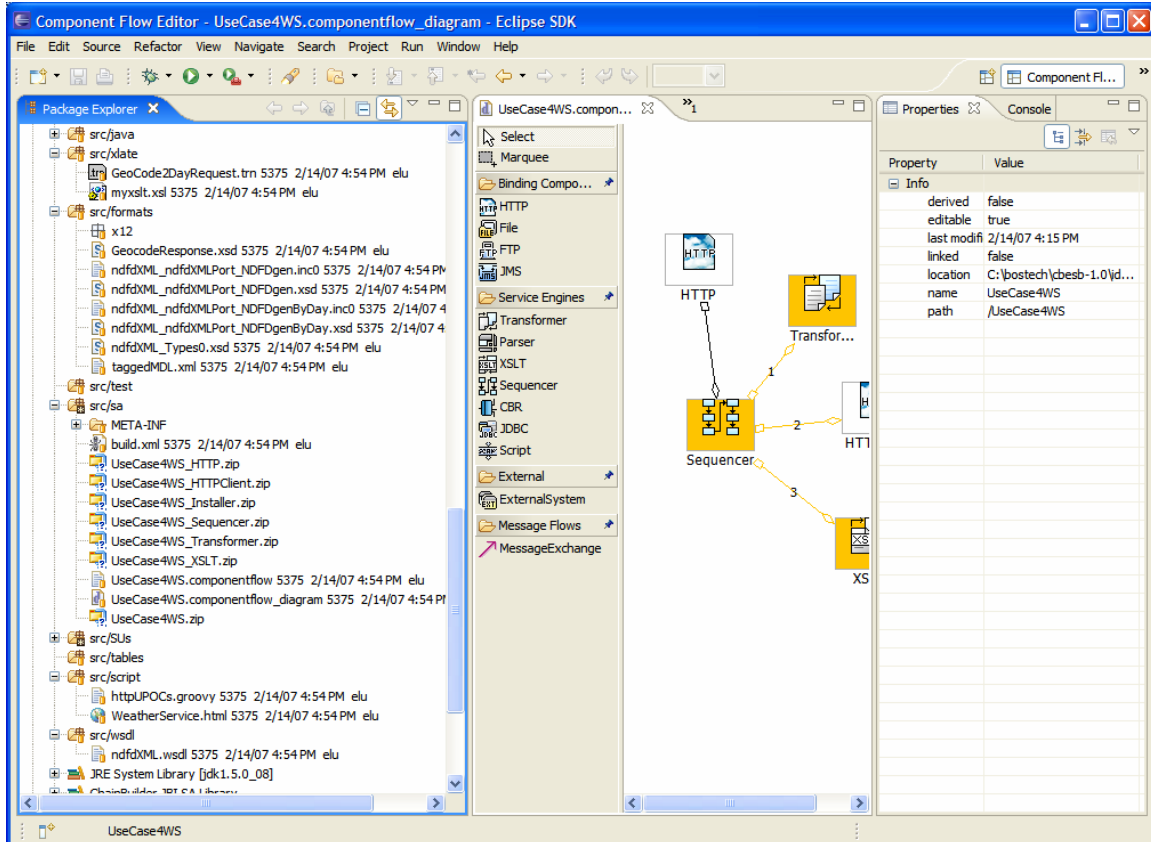


The third step shown in the screen below is to select the location for the generated XSD schema files. You can create a sub-folder under `src\formats` and place the XSD files under the sub-folder.



The screen below the result of the generated XSD files. . The name of each schema should follow the naming convention of:
ServiceName_PortName_OperationName.xsd

A root element should be created for each message defined for the operation, for example there might be an input message, output message and fault message.

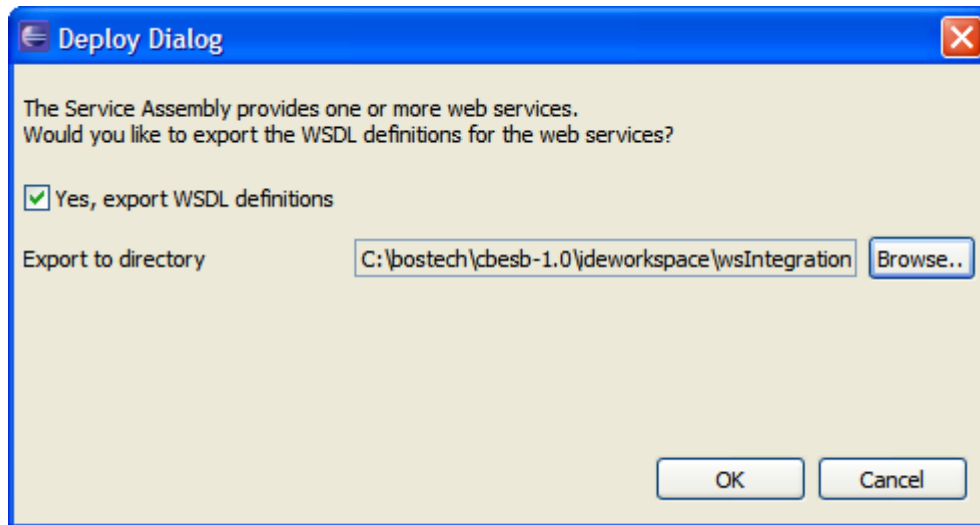


7.2. WSDL Export Wizard

The WSDL Export Wizard is invoked when the "Deploy" function is selected from the Component Flow Editor only when the flow is detected whether there are any service units that are exposing a web service. This is done by checking for HTTP components where the SOAP setting is enabled in the server section. For each exposed web service that is defined in the service assembly, a WSDL file that defines that web service should be generated.

If at least one exposed web service is detected, a new dialog box should be presented to allow the user to export the generated WSDL files. The dialog box has a check box labeled "Yes, export WSDL definitions." and a text box labeled "Export to directory" and include a browse button. This will supply the directory to export the WSDL files to.

If the user selects yes, then they need to supply a valid path to save the files to. The default directory is src\wsdl. The generated WSDL file has the naming convention of {ProjName}_ {ComponentName}.wsdl where the ComponentName is the name of HTTP server component which has SOAP enabled. The following screen shows the dialog box.



The following lists an example of the exported WSDL file:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name='wsIntegrationTest_WebServiceServer_Server'

targetNamespace='http://cbesb.bostechcorp.com/wsIntegrationTest/wsInteg
rationTest_WebServiceServer'

xmlns:tns='http://cbesb.bostechcorp.com/wsIntegrationTest/wsIntegration
Test_WebServiceServer'
  xmlns:sm='http://cbesb.bostechcorp.com/soap/sendmessage/1.0'
  xmlns:http='http://schemas.xmlsoap.org/wsdl/http/'
  xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
  xmlns='http://schemas.xmlsoap.org/wsdl/'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
  <types>
    <xsd:schema elementFormDefault="qualified"

targetNamespace="http://cbesb.bostechcorp.com/soap/sendmessage/1.0">
      <xsd:element name="sendMessage">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="messageType"
type="sm:MessageType" />
            <xsd:element name="message" minOccurs="0"
type="xsd:anyType" />
            <xsd:element name="properties" minOccurs="0"
              type="sm:Properties" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="sendMessageResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="messageType"
type="sm:MessageType" />
            <xsd:element name="message" minOccurs="0"
type="xsd:anyType" />
```

```

        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:simpleType name="MessageType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="XML" />
        <xsd:enumeration value="STRING" />
        <xsd:enumeration value="BASE64" />
    </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="Properties">
    <xsd:sequence>
        <xsd:element name="property" minOccurs="0"
maxOccurs="unbounded"
                type="sm:Property" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Property">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string" />
        <xsd:element name="value" type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
</xsd:schema>
</types>
<message name="sendMessageSoapIn">
    <part name="parameters" element="sm:sendMessage" />
</message>
<message name="sendMessageSoapOut">
    <part name="parameters" element="sm:sendMessageResponse" />
</message>
<portType name='sendMessageInterface'>
    <operation name="sendMessage">
        <input message="tns:sendMessageSoapIn"/>
        <output message="tns:sendMessageSoapOut"/>
    </operation>
</portType>
<binding name='sendMessageSoapBinding'
type='tns:sendMessageInterface'>
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document" />
    <operation name="sendMessage">
        <soap:operation soapAction="sendMessage" style="document"
/>
        <input>
            <soap:body use="literal" />
        </input>
        <output>
            <soap:body use="literal" />
        </output>
    </operation>
</binding>
<service name='wsIntegrationTest_WebServiceServer_Service'>
    <port name='wsIntegrationTest_WebServiceServer_Server'
binding='tns:sendMessageSoapBinding'>
        <soap:address location="http://0.0.0.0:5555/x12parser/" />
    </port>

```

```

    </service>
</definitions>

```

7.3. The *sendMessage* Interface

ChainBuilder ESB will provide the ability to create a web service using the HTTP component. The HTTP component will act as an HTTP server that provides a single operation. This operation is called "sendMessage" and acts as a generic wrapper to pass any kind of data to ChainBuilder ESB. The HTTP component will be able to return a WSDL that describes this service and the ChainBuilder ESB IDE will provide the ability to export the WSDL as described in previous section.

In the CCSL section, we give two examples of the sendMessage formatted data and sendMessageResponse formatted data within a SOAP wrapper. Here we will discuss the sendMessage format in details as shown in following:

```

<sendMessage>
  <messageType>STRING</messageType>
  <message>This is the actual message</message>
  <properties>
    <property>
      <name>prop1</name>
      <value>value 1</value>
    </property>
    <property>
      <name>prop2</name>
      <value>value 2</value>
    </property>
  </properties>
</sendMessage>

```

The "messageType" element may contain the value "XML", "STRING" or "BASE64". If set to "XML", then the contents of the "message" element is treated as XML data and will eventually be placed in the body of a Normalized Message. If messageType is set to "STRING", then the contents of "message" is treated as text data and is set as an attachment in a Normalized Message. If it is set to "BASE64", then the contents of "message" is treated as base 64 encoded binary data. The data is decoded and placed in an attachment of a Normalized Message.

The "properties" element can contain multiple "property" elements. Each property that is received in the sendMessage request will be set as a property to the MessageExchange.

The web service can be configured as In-Only or In-Out. If configured as In-Out, then a response in the sendMessageResponse format will be sent of the form:

```

<sendMessageResponse>
  <messageType>STRING</messageType>
  <message>This is the actual message</message>
</sendMessageResponse>

```


If the "out" message in the MessageExchange contains an XML record, then the messageType is set to "XML". If it is a String record, the type is set to "STRING" and if it is a binary record, then the type is set to "BASE64". The contents of the message element will be either the XML record, String data or base64 encoded binary data

8. Component Reference

All ChainBuilder ESB components use WSDL v1.0 deployment. Component parameters are specified in WSDL extensions. Some components may use additional XML configuration files for component configuration. A service unit may contain one or more .wsdl files. Each file will enable one or more endpoints on a component.

8.1. File Binding Component

8.1.1. Overview

The File binding component defines “input” and “output” extension elements for reading and writing files. File component extensions use the “<http://cbesb.bostechcorp.com/wsdl/file/1.0>” namespace. One interesting feature of the file component is its ability to operate as a consumer with an in-out MEP. Messages are read from an input directory and in-out exchanges are generated. The out messages returned are written to another directory. This makes the file component effective for testing in-out flows.

8.1.2. Description

In input mode, the File binding component will read data from file in the local file system and generate JBI MessageExchanges from the data. As files are processed, up to five different folders may be used:

- Source Directory - The location monitored for new data files.
- Stage Directory - When a data file to be processed is found in the source directory, it is moved to the stage directory where it is opened for reading.
- Archive Directory - When the file is finished being read, it can optionally be moved from the stage directory to an archive directory.
- Hold Directory - If an error occurs during processing, the file may optionally be moved to the hold directory so a user may determine the course of action to correct the problem.
- Reply Directory – For in-out message exchanges, the out message is written to the reply directory.

In output mode, MessageExchanges are received by the component to be written out to files in the local file system. As files are processed, two directories are used:

- Stage Directory - A temporary location used when writing to a file.
- Destination Directory - When a file is complete, it is moved to this directory and made available to an external application.

8.1.3. Configuration Settings

The following table shows the WSDL configuration settings for Input mode:

Name	Required	Description	Default
defaultMep	Y	Type of message exchange to initiate.	
sourceDir	Y	Source location of data files	
stageDir	Y	Staging directory where files are moved for processing.	
archiveDir	N	Archive location to place files when done processing.	
holdDir	N	Hold location to place files when an error occurs during processing.	
scanInterval	N	Value in milliseconds that determines how often the source directory is scanned for new data files.	5000 (5 seconds)
hold	N	Boolean value. True means that if an error occurs while processing a file, the file will be moved to the Hold directory. If value is set to false, no special processing will be done when an error occurs. If true, then HoldDir must be specified.	false
filePattern	N	Glob style file pattern to determine which files in the Source directory will be processed. Only files that match the pattern will be processed.	* (all files)
twoPass	N	Boolean value. Two pass mode causes the component to check the size of the files in the Source directory, wait for a set interval and check the sizes again. Only files that did not change size during the interval will be processed. This is to prevent processing a file that is still being written to by an external application. True enables two pass mode, false disables it.	false
twoPassInterval	N	Value in milliseconds to wait between scans during Two Pass mode. This is only used when TwoPass is set to true.	2000 (2 seconds)

fileCompleteAction	N	Value determines what to do with the file after all data is read from it. Acceptable values are: delete - File is deleted archive - File is moved to archive directory	delete
recordsPerMessage	N	Integer value that determines the number of records from a file will be placed in an individual Normalized Message. 0 indicates that all records in the file will be placed in a single message. Any value > 0 will be the maximum number of records placed in a single message.	0
readStyle	N	Value determines how to read a record from the file. Acceptable values are: raw - entire file contents is one record. newline - each line in the file is one record.	raw
recordType	N	Value determines the type of data each record contains. Acceptable values are: xml - each record is well formed XML string - each record is character data binary - each record is binary data	string
charset	N	Value is the name of the charset to use to read in character data.	system default
archiveFilePattern	N	Describes a file pattern to use to rename the file when being archived. This can be used to add a date/time stamp to the file. If the value is null, then the file is not renamed when it is moved to archiveDir. The pattern may contain literal characters as well as the following macros that will be replaced with values at runtime: {DATE} - The system date formatted as yyyyymmdd {TIME} - The system time formatted as hhmmss {BASENAME} - The original file's base name (name without extension). {EXT} - The original file's extension. {COUNT} - An automatically incremented value that starts from 1 when the component is started.	null
replyDir	required	Directory where the out message is	

	only for MEP's with an out message	written.	
replyCharset	N	Value is the name of the charset to use to write character data to the file.	system default
replyWriteStyle	N	Value determines how records are written to a file. Acceptable values are: raw - Each record from a Normalized Message is written to an individual file. newline - Each record from a Normalized Message is written to the same file separated by a newline.	raw
replyFilePattern	Y	Describes a file pattern to use to name the file when being written. This can be used to add a date/time stamp to the file. The pattern may contain literal characters as well as the following macros that will be replaced with values at runtime: {DATE} - The system date formatted as yyyyymmdd {TIME} - The system time formatted as hhmmss {BASENAME} - The original file's base name (name without extension). {EXT} - The original file's extension. {COUNT} - An automatically incremented value that starts from 1 when the component is started.	

The following table shows the WSDL configuration setting for the Output mode:

Name	Required	Description	Default
defaultMep	Y	Type of message exchange to accept as a producer.	
destDir	Y	Destination location for data files	
stageDir	Y	Staging directory where files are created and written to.	
charset	N	Value is the name of the charset to use to write character data to the file.	system default
writeStyle	N	Value determines how records are written to a file. Acceptable values are: raw - Each record from a Normalized Message is written to an individual file.	raw

		newline - Each record from a Normalized Message is written to the same file separated by a newline.	
filePattern	Y	<p>Describes a file pattern to use to name the file when being written. This can be used to add a date/time stamp to the file. The pattern may contain literal characters as well as the following macros that will be replaced with values at runtime:</p> <p>{DATE} - The system date formatted as yyyyymmdd</p> <p>{TIME} - The system time formatted as hhmmss</p> <p>{BASENAME} - The original file's base name (name without extension).</p> <p>{EXT} - The original file's extension.</p> <p>{COUNT} - An automatically incremented value that starts from 1 when the component is started.</p>	

8.1.4. Example

This shows an example of a WSDL file for the File Binding Component.

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions name='FileReader'
  targetNamespace='http://bostechcorp.com/wsd1/file/'
  xmlns:tns='http://bostechcorp.com/wsd1/file/'
  xmlns:file='http://cbesb.bostechcorp.com/wsd1/file/1.0'
  xmlns='http://schemas.xmlsoap.org/wsd1/'
  xmlns:jbi='http://servicemix.org/wsd1/jbi/'>

  <portType name='FileInterface'>
  </portType>

  <binding name='FileBinding' type='tns:FileInterface'>
    <file:binding />
  </binding>

  <service name='FileService'>
    <port name='FileReader' binding='tns:FileBinding'>
      <file:input defaultMep='in-out'
        sourceDir="inbox"
        stageDir="inbox-stage"
        archiveDir="inbox-archive"
        hold="false"
        holdDir=""
      />
    </port>
  </service>
</definitions>
```

```

        filePattern="*"
        twoPass="false"
        twoPassInterval="3000"
        fileCompleteAction="delete"
        recordsPerMessage="1"
        readStyle="raw"
        recordType="string"
        charset="utf-8"
        archiveFilePattern="*"
        readDirectoryPath="inbox"
        readStagePath="inbox-stage"
        readArchivePath="inbox-archive"
        readHold="false"
        readType="string"
        scanInterval="5000"
        readIntervalMilliseconds="5000"
        replyDir="replybox"
        replyFilePattern="FOO_{TIME}"/>
    <jbi:endpoint role="consumer" defaultOperation="tns:Echo"/>
</port>
</service>
</definitions>

```

```

<?xml version='1.0' encoding='UTF-8'?>
<definitions name='FileReader'
    targetNamespace='http://bostechcorp.com/cbesb/fo-example2'
    xmlns:tns='http://bostechcorp.com/cbesb/fo-example2'
    xmlns:file='http://cbesb.bostechcorp.com/wsd1/file/1.0'
    xmlns='http://schemas.xmlsoap.org/wsd1/'
    xmlns:jbi='http://servicemix.org/wsd1/jbi/'>

    <portType name='FileInterface'>
    </portType>

    <binding name='FileBinding' type='tns:FileInterface'>
        <file:binding />
    </binding>

    <service name='FileWriter'>
        <port name='FileWriter' binding='tns:FileBinding'>
            <file:output defaultMep='in-out'
                destDir="outbox"
                stageDir="outbox-stage"
                charset="utf-8"
                writeStyle="raw"
                filePattern="Contact_{TIME}.xml"
            />
        </port>
    </service>
</definitions>

```

8.2. Http Binding Component

8.2.1. Overview

The HTTP component defines the “binding” and “address” elements for specifying an endpoint and operation. These use the “<http://schemas.xmlsoap.org/wsdl/http/>” namespace as a prefix.

The HTTP component is modified based on the Servicemix’s HTTP component (<http://servicemix.org/site/servicemix-http.html>) and provides the HTTP and SOAP binding. It can act as an HTTP client and has an integrated HTTP server based on open source Jetty 6 technology.

8.2.2. Description

Please refer to the ServiceMix web site (<http://servicemix.org/site/servicemix-http.html>) for the description of various configuration setting.

8.2.3. Example

This shows an example of an HTTP WSDL file.

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions name='Consumer'
  targetNamespace='http://bostechcorp.com/cbesb/UseCase4'
  xmlns:tns='http://bostechcorp.com/cbesb/UseCase4'
  xmlns:http='http://schemas.xmlsoap.org/wsdl/http/'
  xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
  xmlns='http://schemas.xmlsoap.org/wsdl/'
  xmlns:jbi='http://servicemix.org/wsdl/jbi/'>

  <portType name='ConsumerInterface'>
  </portType>

  <binding name='ConsumerHttpBinding' type='tns:ConsumerInterface'>
    <http:binding verb="POST"></http:binding>
  </binding>

  <service name='HTTPConsumer'>
    <port name='HTTPServer-8192' binding='tns:ConsumerHttpBinding'>
      <http:address location="http://:8192/WeatherService"/>
      <jbi:endpoint role="consumer" defaultMep='in-out' />
    </port>
  </service>
</definitions>
```

8.3. JMS Binding Component

8.3.1. Overview

The JMS binding component is a component to send and receive messages from an IBM Websphere MQ Server (v5.3 and above) or other JMS 1.1-compliant server.

Here are the main features:

- Put a request message into message destination (queue or topic) with option to retrieve the reply message from reply destination using Correlation ID matching the original request message id.
- Retrieve a request message from message destination with option to put a reply message into reply destination with Correlation ID equal to the request message ID.
- Support different message read style(xml/raw/newline)
- Support character set encoding
- SOAP 1.1 and 1.2 support (supported in later release)
- MIME attachments (supported in later release)
- WS-Addressing support (supported in later release)
- Support for all MEPs as consumers or providers

The JMS binding component can be configured as a consumer or provider endpoint. When JMS is configured as a consumer endpoint, it reads a message from a destination and creates MessageExchanges that are routed to the NMR. When the defaultMep is set to “InOut”, a reply message with its Correlation ID equal to the request message ID is put into the reply destination.

If the JMS binding component is configured as a provider endpoint, it receives a MessageExchange from the NMR and creates a JMS message and puts it into the destination. In this case, if the defaultMep is set to “InOut”, it will retrieve a reply message from the optional reply destination with matching correlation ID equal to request message ID

The JMS binding component defines “config” extension elements for reading and writing messages from and to a queue. JMS component extension uses the “http://cbesb.bostechcorp.com/wsdl/jms/1.0” namespace.

Note : If you use the JMS component to connect with IBM Websphere MQ sever, you need to copy three jar files - com.ibm.mq.jar, com.ibm.mqjms.jar, dhibcore.jar from MQ installation’s Java\lib into %CBESB_HOME%\apache-servicemix\lib\optional directory.

8.3.2. Description

The following parameters can be configured using a properties file or a JMX console.

Name	Required	Description	Default
userId	N	User id to create a JMS connection	
password	N	Password for the above user id	
replyTimeout	N	Time in milliseconds to wait for a reply before failing. Any value less than or equal to zero means infinite wait.	0
jndiInitialContextFactory	N	Default JNDI InitialContext factory	“com.sun.jndi.fscontext.ReffSContextFactory”
jndiProviderUrl	N	Default JNDI provider url	“file:/C:/CBESB/JndiDir”
jndiConnectionFactoryName	N	Default JNDI name to lookup the JMS ConnectionFactory	

The following table shows the WSDL configuration setting:

Name	Required	Description	Default
role	Y	Either “consumer” or “provider”	“consumer”
defaultMep	Y	The type of message exchange pattern (MEP) as defined in WSDL spec; The allowed value is: <ul style="list-style-type: none"> • InOut • InOnly • ReliableIn The type of DefaultMep dictates if it needs to handle reply message	“InOnly”
defaultOperation	N	Reserved for future use	
jndiInitialContextFactory	N	Default JNDI InitialContext factory	“com.sun.jndi.fscontext.ReffSContextFactory”
jndiProviderUrl	N	Default JNDI provider url	“file:/C:/CBESB/JndiDir”
jndiConnectionFactoryName	Y	Default JNDI name to lookup the JMS ConnectionFactory	
destinationStyle	Y	Either “queue” or “topic”; the “topic” is used to support pub/sub	“queue”
targetDestinationName	Y	The destination to use for putting or retrieving message depending on role	

		type;	
replyDestinationName	N	Only used when the value of DefaultMep is "InOut"; It specifies the destination to put or retrieve reply message; the correlation ID needs always to be used;	
replyTimeout	N	Time in milliseconds to wait for a reply before failing. Any value less than or equal to zero means infinite wait.	0
recordsPerMessage	N	Integer value that determines the number of records from a JMS message will be placed in an individual Normalized Message. 0 indicates that all records in the message will be placed in a single message. Any value > 0 will be the maximum number of records placed in a single message.	0
readStyle	N	Value determines how to read a record from the JMS message. Acceptable values are: raw - The entire JMS message contents is one record. newline - each line in the message is one record.	raw
recordType	N	Value determines the type of data each record contains. Acceptable values are: xml - each record is well formed XML string - each record is character data binary - each record is binary data	string
writeStyle	N	Value determines how records are written to the destination. Acceptable values are: raw - Each record from a Normalized Message is put as a individual message. newline - Each record from a Normalized Message is put into one message separated by a newline.	raw
charset	N	Value is the name of the charset to use to read in character data.	system default

8.3.3. Example

This shows an example of a JMS WSDL file.

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions name='JMSDemo'
  targetNamespace='http://jms.bostechcorp.com/Test'
  xmlns:tns='http://jms.bostechcorp.com/Test'
  xmlns:jms='http://cbesb.bostechcorp.com/wsd1/jms/1.0'
  xmlns='http://schemas.xmlsoap.org/wsd1/'>

  <portType name='JMSInterface'></portType>

  <binding name='JMSBinding' type='tns:JMSInterface'>
    <jms:binding />
  </binding>

  <service name='JMSInboundService'>
    <port name='JMSInbound' binding='tns:JMSBinding'>
      <jms:config role="consumer"
        defaultMep="in-out"
        destinationStyle="queue"
          jndiConnectionFactoryName="ivtQCF"
          targetDestinationName="postcard"
          replyDestinationName="default"
          recordsPerMessage="0"
          readStyle="raw"
          recordType="string"
          charset="ISO8859-1" />
    </port>
  </service>
</definitions>
```

8.4. FTP Binding Component

8.4.1. Overview

The FTP binding component defines “input” and “output” extension elements for reading and writing files on a remote FTP server. FTP component extensions use the “http://cbesb.bostechcorp.com/wsd1/ftp/1.0” namespace. One interesting feature of the FTP component is its ability to operate as a consumer with an in-out MEP. Messages are read from an input directory and in-out MessageExchanges are generated. The out messages returned are written to another directory. This makes the FTP component effective for testing in-out flows.

8.4.2. Description

In input mode, the FTP binding component will retrieve files from an FTP server and place them in a specified directory on the local file system. Then the files will be read from the local file system and JBI MessageExchanges will be created using the data in the files. As files are processed, up to six different folders may be used:

- Source Directory – the directory on the FTP server to retrieve files from.
- Transfer Directory – While a file is being transferred, it is written to this directory. This directory is in the local file system as ChainBuilder ESB server. When the component is started, this directory is checked for any existing files. These files are considered incomplete transfers and are deleted at startup.
- Stage Directory – When a transfer is complete, the file is moved from the Transfer directory, to the Stage directory. This directory is in the local file system as ChainBuilder ESB server. The file is read from this directory and the data in the file is converted into MessageExchanges.
- Archive Directory - When the file is finished being read, it can optionally be moved from the stage directory to an archive directory. This directory is in the local file system as ChainBuilder ESB server.
- Hold Directory - If an error occurs during processing, the file may optionally be moved to the hold directory so a user may determine the course of action to correct the problem. This directory is in the local file system as ChainBuilder ESB server.
- Reply Directory – For in-out message exchanges, the out message is uploaded to the specified reply directory on the FTP server.

In output mode, MessageExchanges are received by the component to be written out to files in the local file system. Then, the file is transferred to the FTP server. As files are processed, three directories are used:

- Stage Directory - A temporary location used when writing to a file. This directory is in the local file system as ChainBuilder ESB server.
- Transfer Directory – Another temporary directory used when uploading a file. This directory is in the local file system as ChainBuilder ESB server.
- Destination Directory – The directory on the FTP server to store the files.

8.4.3. Configuration Settings

The following table shows the WSDL configuration settings for Input mode:

Name	Required	Description	Default
defaultMep	Y	Type of message exchange to initiate.	In-only

ftpHost	Y	Hostname or IP address of FTP server	
ftpUser	Y	User name to login to FTP server	
ftpPassword	Y	Password to login to FTP server	
cmdFile	N	Specify the XML based command file described in Scripted Mode.	
ftpConnectMode	N	Value should be “active” or “passive”	Passive
ftpTransferMode	N	Value should be “ascii” or “binary”	Binary
sourceDir	Y	Directory on FTP server where files are to be retrieved.	
transferDir	Y	Directory where files are downloaded to.	
stageDir	Y	Staging directory where files are moved for processing.	
archiveDir	N	Archive location to place files when done processing.	
holdDir	N	Hold location to place files when an error occurs during processing.	
scanInterval	N	Value in milliseconds that determines how often the source URL is scanned for new data files.	60000 (1 minute)
hold	N	Boolean value. True means that if an error occurs while processing a file, the file will be moved to the Hold directory. If value is set to false, no special processing will be done when an error occurs. If true, then HoldDir must be specified.	False
filePattern	N	Glob style file pattern to determine which files in the Source URL will be processed. Only files that match the pattern will be processed.	* (all files)
twoPass	N	Boolean value. Two pass mode causes the component to check the size of the files in the Source URL, wait for a set interval and check the sizes again. Only files that did not change size during the interval will be processed. This is to prevent processing a file that is still being written to by an external application. True enables two pass mode, false disables it.	False

twoPassInterval	N	Value in milliseconds to wait between scans during Two Pass mode. This is only used when TwoPass is set to true.	2000 (2 seconds)
fileCompleteAction	N	Value determines what to do with the file after all data is read from it. Acceptable values are: delete - File is deleted archive - File is moved to archive directory	Delete
recordsPerMessage	N	Integer value that determines the number of records from a file will be placed in an individual Normalized Message. 0 indicates that all records in the file will be placed in a single message. Any value > 0 will be the maximum number of records placed in a single message.	0
readStyle	N	Value determines how to read a record from the file. Acceptable values are: raw - The entire file contents is one record. newline - each line in the file is one record.	Raw
recordType	N	Value determines the type of data each record contains. Acceptable values are: xml - each record is well formed XML string - each record is character data binary - each record is binary data	String
charset	N	Value is the name of the charset to use to read in character data.	system default
archiveFilePattern	N	Describes a file pattern to use to rename the file when being archived. This can be used to add a date/time stamp to the file. If the value is null, then the file is not renamed when it is moved to archiveDir. The pattern may contain literal characters as well as the following macros that will be replaced with values at runtime:	null

		<p>{DATE} - The system date formatted as yyyyymmdd</p> <p>{TIME} - The system time formatted as hhmmss</p> <p>{BASENAME} - The original file's base name (name without extension).</p> <p>{EXT} - The original file's extension.</p> <p>{COUNT} - An automatically incremented value that starts from 1 when the component is started.</p>	
replyFtpHost	required only for MEP's with an out message	Host name or IP address of FTP server to send replies to.	Same as ftpHost
replyFtpUser	required only for MEP's with an out message	User to login as to send replies.	Same as ftpUser
replyFtpPassword	required only for MEP's with an out message	Password to login to send replies	Same as ftpPassword
replyFtpConnectMode	N	Value should be "active" or "passive"	Passive
replyFtpTransferMode	N	Values should be "ascii" or "binary"	Binary
replyDir	required only for MEP's with an out message	Directory on FTP server to place files containing reply messages.	
replyCharset	N	Value is the name of the charset to use to write character data to the file.	system default
replyWriteStyle	N	Value determines how records are written to a file. Acceptable values are: raw - Each record from a Normalized Message is written to an individual file. newline - Each record from a Normalized Message is written to the same file separated by a newline.	Raw
replyFilePattern	required only for MEP's with an out message	Describes a file pattern to use to name the file when being written. This can be used to add a date/time stamp to the file. The pattern may contain literal	

		<p>characters as well as the following macros that will be replaced with values at runtime:</p> <ul style="list-style-type: none">{DATE} - The system date formatted as yyyyymmdd{TIME} - The system time formatted as hhmmss{BASENAME} - The original file's base name (name without extension).{EXT} - The original file's extension.{COUNT} - An automatically incremented value that starts from 1 when the component is started.	
--	--	---	--

The following table shows the WSDL configuration settings for Output mode:

Name	Required	Description	Default
defaultMep	Y	Type of message exchange to accept as a producer.	In-only
ftpHost	Y	Hostname or IP address of FTP server	
ftpUser	Y	User name to login to FTP server	
ftpPassword	Y	Password to login to FTP server	
cmdFile	N	Specify the XML based command file described in Scripted Mode.	
ftpConnectMode	N	Value should be "active" or "passive"	Passive
ftpTransferMode	N	Value should be "ascii" or "binary"	Binary
destDir	Y	Destination location on FTP server for data files	
transferDir	Y	Local directory used when transferring files.	
stageDir	Y	Staging directory where files are created and written to.	
charset	N	Value is the name of the charset to use to write character data to the file.	system default
writeStyle	N	Value determines how records are written to a file. Acceptable values are: raw - Each record from a Normalized Message is written to an individual file. newline - Each record from a Normalized Message is written to the same file separated by a newline.	Raw
filePattern	Y	Describes a file pattern to use to name the file when being written. This can be used to add a date/time stamp to the file. The pattern may contain literal characters as well as the following macros that will be replaced with values at runtime: {DATE} - The system date formatted as yyymmdd {TIME} - The system time formatted as hhmmss {BASENAME} - The original file's base name (name without extension). {EXT} - The original file's extension. {COUNT} - An automatically	

		incremented value that starts from 1 when the component is started.	
fileCompleteAction	N	Value determines what to do with the file after it is transferred to the FTP server. Acceptable values are: delete - File is deleted archive - File is moved to archive directory	Delete
archiveDir	Required when fileCompleteAction = archive	Directory to place the local copy of the file when done transferring.	
archiveFilePattern	N	Describes a file pattern to use to rename the file when being archived. This can be used to add a date/time stamp to the file. If the value is null, then the file is not renamed when it is moved to archiveDir. The pattern may contain literal characters as well as the following macros that will be replaced with values at runtime: {DATE} - The system date formatted as yyymmdd {TIME} - The system time formatted as hhmmss {BASENAME} - The original file's base name (name without extension). {EXT} - The original file's extension. {COUNT} - An automatically incremented value that starts from 1 when the component is started.	Null

8.4.4. Example

This shows an example of a WSDL file for the FTP Binding Component.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name='FTP_Reader_Input'
  targetNamespace='http://bostechcorp.com/SU/FTP_Reader'
  xmlns:tns='http://bostechcorp.com/SU/FTP_Reader'
```

```

    xmlns:ftp='http://cbesb.bostechcorp.com/wsdl/ftp/1.0'
    xmlns='http://schemas.xmlsoap.org/wsdl/'>
<portType name='FTP_Reader_Interface'>
</portType>
<binding name='FTP_ReaderBinding' type='tns:FTP_Reader_Interface'>
  <ftp:binding/>
</binding>
<service name='FTP_Reader_Service'>
  <port name='FTP_Reader_Input' binding='tns:FTP_ReaderBinding'>
    <ftp:input
      recordType="string"
      ftpTransferMode="binary"
      ftpHost="10.10.10.100"
      scanInterval="10000"
      hold="false"
      sourceDir="/tmp/ftptest/inbox"
      twoPass="false"
      fileCompleteAction="delete"
      ftpUser="usr"
      replyFtpTransferMode="binary"
      replyFtpConnectMode="passive"
      ftpConnectMode="passive"
      filePattern="*"
      charset=""
      stageDir="ftp_stage"
      ftpPassword="password"
      replyCharset=""
      readStyle="raw"
      replyWriteStyle="raw"
      recordPerMessage="0"
      defaultMep="in-only"
      transferDir="ftp_transfer"
    />
  </port>
</service>
</definitions>

```

8.5. Scripting Support in FTP Binding Component

The scripting support in the FTP component is achieved by sending messages to the FTP component which contain a type of script which specifies the actions to take. Since this is triggered by sending a message to the component, the script mode uses Output mode.

The message sent to the component consists of a root tag named "ftp_request" that belongs to the target namespace of the component, "http://cbesb.bostechcorp.com/wsdl/ftp/1.0".

Inside the ftp_request tag is another tag named "commands" which contains a sequence of elements that represent an operation to execute.

```

<ftp_request xmlns="http://cbesb.bostechcorp.com/wsdl/ftp/1.0">
  <commands>
    <connect host="download.foo.com" />
    <login name="test" password="download123" />
  </commands>
</ftp_request>

```

```

        <changeWorkingDir>data/out</changeWorkingDir>
        <get>orders.txt</get>
        <logout />
        <disconnect />
    </commands>
</ftp_request>

```

The operations are executed in the order they appear in the message. If an error occurs during the execution of an operation, then none of the remaining operations are executed. When complete, a response message may be returned if the Message Exchange is In-Out. The format of the response message is a root element named "ftp_response" that belongs to the same namespace as the request message. The ftp_response message contains the following sub-elements:

status - Boolean value that indicates whether the request was executed successfully or not.

result - If status = true, then any output generated by the commands is placed in this element. See the individual operations to see sample output.

error - If status = false, then the cause of the failure is placed in the error element.

```

<ftp_response xmlns="http://cbesb.bostechcorp.com/wsdl/ftp/1.0">
    <status>true</status>
    <result></result>
</ftp_response>

```

```

<ftp_response xmlns="http://cbesb.bostechcorp.com/wsdl/ftp/1.0">
    <status>false</status>
    <error>Host not found</error>
</ftp_response>

```

The rest of this section provides the complete list of command elements implemented by the FTP Binding Component for its scripting support.

8.5.1. connect

Opens a connection to an FTP server.

Attributes:

Name	Required	Description
Host	No	Hostname of FTP server, if not supplied ftpHost setting is used from Output configuration.
Port	No	Port number to connect to. If not specified, default FTP port is used.

Value:

None

Examples:

```
<connect />
```

```
<connect host="download.foo.com" />
<connect host="download.bar.com" port="10023" />
```

8.5.2. disconnect

Closes a connection to an FTP server.

Attributes:

None

Value:

None

Example:

```
<disconnect />
```

8.5.3. login

Logs into the FTP server.

Attributes:

Name	Required	Description
User	No	Username to log in as. If not specified, then ftpUser value is used from Output configuration.
Password	No	Password used to log in. If not specified then ftpPassword is used from Output configuration.

Value:

None

Examples:

```
<login />
<login user="test" password="download123" />
```

8.5.4. logout

Logs out from an FTP server.

Attributes:

None

Value:

None

Example:

```
<logout />
```

8.5.5. siteCommand

Sends a site specific command to the FTP server.

Attributes:

None

Value:

The SITE command to be send to the FTP server

Example:

```
<siteCommand>LSTFMT 0</siteCommand >
```

NOTE : Please refer to the site for a list of valid SITE command for [IBM iSeries](#). It varies from server to server.

8.5.6. setConnectionMode

Sets the connection mode to the specified value.

Attributes:

None

Value:

Either "active" or "passive"

Example:

```
<setConnectionMode>passive</setConnectionMode>
```

8.5.7. setTransferMode

Sets the transfer mode to the specified value.

Attributes:

None

Value:

Either "ascii" or "binary"

Example:

```
<setTransferMode>binary</setTransferMode>
```

8.5.8. changeWorkingDir

Changes the current working directory to the specified directory.

Attributes:

None

Value:

The path to change to.

Example:

```
<changeWorkingDir>data/in</changeWorkingDir>
```

8.5.9. changeToParentDir

Changes the working directory to the parent of the current working directory.

Attributes:

None

Value:

None

Example:

```
<changeToParentDir />
```

8.5.10. get

Retrieves the specified file from the FTP server.

Attributes:

Name	Required	Description
localName	No	The file name stored into the local file system; if omitted, the specified file name will be used to store file in the default local directory

Value:

The path to the file in the FTP server

Examples:

```
<get localName="c:\data\in.txt">
  in/in.xml
</get>
```

8.5.11. put

Transfers the specified file to the FTP server.

Attributes:

Name	Required	Description
remoteName	No	The file name stored into the FTP server; if omitted, the specified file name will be used to store file in the default remote FTP directory

Value:

The path to the file in local file system

Examples:

```
<put remoteName="in/in.xml">
  C:\in\in.txt
</put>
```

8.5.12. deleteFile

Deletes the specified file from the FTP server.

Attributes:

None

Value:

The path to the file to be deleted.

Example:

```
<deleteFile>data/out/foo.txt</deleteFile>
```

8.5.13. rename

Rename the specified remote file in the FTP server.

Attributes:

Name	Required	Description
toName	Yes	The new file name of the remote file.

Value:

The name to the remote file to be renamed.

Example:

```
<rename toName="data/out/bar.txt">data/out/foo.txt</rename>
```

8.5.14. createDirectory

Create the specified directory in the FTP server.

Attributes:

None

Value:

The path to the directory to be created.

Example:

```
<createDirectory>data/out</createDirectory >
```

8.5.15. removeDirectory

Remove the specified directory in the FTP server (if empty).

Attributes:

None

Value:

The path to the directory to be removed.

Example:

```
<removeDirectory>data/out</removeDirectory >
```

8.5.16. mget

Retrieves the specified file(s) from the FTP server based on glob expression file pattern

Attributes:

N/A

Value:

The glob expression file pattern specified the files in the FTP server

Examples:

```
<mget> in/*.xml/ </mget>
```

8.5.17. mput

Transfers the specified file(s) to the FTP server based on glob expression file pattern

Attributes:

N/A

Value:

The glob expression file pattern specified the files in local file system

Examples:

```
<mput > C:\in\*.txt</mput>
```

8.5.18. mDeleteFiles

Deletes the specified file(s) from the FTP server based on the glob expression file pattern.

Attributes:

None

Value:

The glob expression file pattern specified the file(s) in the FTP server to be deleted.

Example:

```
<mDeleteFiles>data/out/*.txt</mDeleteFiles >
```

8.5.19. changeLocalWorkingDir

Changes the current local working directory to the specified directory in local file system.

Attributes:

None

Value:

The path to change to.

Example:

```
<changeLocalWorkingDir>data/in</changeLocalWorkingDir>
```

8.5.20. deleteLocalFile

Deletes the specified file from the local file system.

Attributes:

None

Value:

The path to the file to be deleted.

Example:

```
<deleteLocalFile>data/out/foo.txt</deleteLocalFile>
```

8.5.21. renameLocal

Rename the specified local file.

Attributes:

Name	Required	Description
toName	Yes	The new name of the local file.

Value:

The name to the local file to be renamed.

Example:

```
<renameLocal toName="data/out/bar.txt">data/out/foo.txt</renameLocal>
```

8.5.22. createLocalDirectory

Create the specified directory in the local file system.

Attributes:

None

Value:

The path to the local directory to be created.

Example:

```
<createLocalDirectory>data/out</createLocalDirectory >
```

8.5.23. removeLocalDirectory

Remove the specified directory in the local file system (if empty).

Attributes:

None

Value:

The path to the directory to be removed.

Example:

```
<removeLocalDirectory>data/out</removeLocalDirectory >
```

8.5.24. mDeleteLocalFiles

Deletes the specified file(s) from the local file system based on the glob expression file pattern.

Attributes:

None

Value:

The glob expression file pattern specified the file(s) in the local file system to be deleted.

Example:

```
<mDeleteLocalFiles>data/out/*.txt</mDeleteLocalFiles >
```

8.6. Sequencing Service Engine

8.6.1. Overview

The Sequencing service engine (SE) is capable of executing a sequence of services one by one with the output of one service being provided as the input to the next one.

The service that sends a message exchange to the sequencing engine may use either an In-Only or In-Out MEP (Message Exchange Pattern), but all services that are included in the sequence must use an In-Out MEP. This must be enforced because in order to provide a source message to the next service in the list, a result message must be provided by the previous service.

The sequencing component defines a “config” tag in the “http://cbesb.bostechcorp.com/wsd1/sequencer/1.0” namespace. In addition to the WSDL, a sequencing component uses an xml file to describe the services to invoke. This list is called a service list. A single service unit may contain multiple service lists, so different sequences of services can be supported.

8.6.2. Description

The Service List configuration file is an XML based file that contains the services to invoke and the order to invoke them.

The servicelist element contains one or more target elements. The order of the targets determines the order which they will be invoked. Each target element must contain one of the following elements:

- service - The namespace qualified name of the service to be invoked.
- interface - The namespace qualified name of the interface to be invoked.

Each target element may also contain these optional elements:

- operation - If the service provides multiple operations, then the operation to invoke may be specified.
- timeout - A timeout period (in milliseconds) may be specified. If no timeout period is specified, the sequencer will wait an indefinite period of time for the called service to return.

8.6.3. Example

This shows an example of a WSDL and service list file for a sequencer component.

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions name='Sequencer'
  targetNamespace='http://bostechcorp.com/cbesb/UseCase4'
  xmlns:tns='http://bostechcorp.com/cbesb/UseCase4'
```

```

xmlns:sequencer='http://cbesb.bostechcorp.com/wsd/sequencer/1.0'
      xmlns='http://schemas.xmlsoap.org/wsd/'>

  <portType name='WeatherSequencerInterface'>
  </portType>

  <binding name='WeatherSequencerBinding'
type='tns:WeatherSequencerInterface'>
    <sequencer:binding />
  </binding>

  <service name='WeatherSequencerService'>
    <port name='WeatherSequencerEndpoint'
binding='tns:WeatherSequencerBinding'>
      <sequencer:config role="provider"
serviceList="serviceList.xml" />
    </port>
  </service>
</definitions>

<servicelist
xmlns="http://cbesb.bostechcorp.com/sequencer/servicelist/1.0"
  xmlns:ex="http://bostechcorp.com/cbesb/UseCase4"
  xmlns:ndfd="http://www.weather.gov/forecasts/xml/DWMLgen/wsd/ndf
dXML.wsd">
  <target>
    <service>ex:WeatherMapService</service>
  </target>
  <target>
    <service>ndfd:ndfdXML</service>
  </target>
  <target>
    <service>ex:XSLTService</service>
  </target>
</servicelist>

```

8.7. Content Based Router (CBR) Service Engine

8.7.1. Overview

The *CBR* service engine examines the message content and routes the message onto a different channel based on data contained in the message. The routing can be based on a number of criteria such as existence of fields, specific field values etc.

The service that sends a MessageExchange to the CBR may use either an in-only or in-out MEP (Message Exchange Pattern). If the MessageExchange is delivered to a single target, then either an in-only or in-out MEP may be used. If the MessageExchange is delivered to multiple targets, then only an in-only MEP may be used.

The CBR component defines a “config” tag in the “http://cbesb.bostechcorp.com/wsdl/cbr/1.0” namespace. In addition to the WSDL, a CBR component uses an XML file to describe the routing rules that a MessageExchange will flow through. The file is called a routing rules file. A single service unit may contain multiple routing rules, so different routing can be supported.

8.7.2. CBR Message Identification

The Message (Transaction) Identification is abbreviated as the TrxId. The concept of the TrxId is to examine the content of the in message of a MessageExchange and return the identifier. The TrxId will be set in the metadata of the MessageExchange and is passed along to the rest of the routing flow.

The following lists the supported TrxId types:

Fixed

Fixed type TrxId uses the length and offset to determine the TrxId of a MessageExchange. Typically, this method is used to determine the TrxId for proprietary non-XML data. Since non-XML data is stored as attachments in a message, the CBR will operate on attachments to retrieve the TrxId. If there are multiple attachments in the input message, only the first one will be used,

If the message attachment contains the data “ORD00123”, and you specify the fixed TrxId length as 3 and offset as 0, then the string “ORD” will be returned as the TrxId.

The following is the XML fragment in the WSDL for the fixed TrxId:

```
<trxId type='fixed'
      offset='0' length='3' />
```

CSV (Comma Separated Format)

CSV type TrxId uses a delimiter and index to determine the TrxId of a MessageExchange. Typically, this method is used to determine the TrxId for proprietary non-XML data. Since non-XML data is stored as attachments in a message, the CBR will operate on attachments to retrieve the TrxId. If there are multiple attachments in the in message, only the first one will be used,

If the message attachment contains the data “john,smith,male,25”, and you specify the CSV TrxId’s delimiter as “,” and index as 3, then the string “male” will be returned as the TrxId.

The following is the XML fragment in the WSDL for the CSV TrxId:

```
<trxId type='csv'
```

```
delimiter=', ' index='3' />
```

X12

X12 type TrxId is used to determine the TrxId of a MessageExchange containing EDI X12 data. The CBR will look into the ST segment in the X12 data to return the transaction type. Since non-XML X12 data is stored as attachments in a message, the CBR will operate on the attachments to retrieve the TrxId. If there are multiple attachments in the message, only the first attachment will be used.

The following is the XML fragment in the WSDL for the X12 TrxId:

```
<trxId type='x12' />
```

HL7

HL7 type TrxId is used to determine the TrxId of a MessageExchange containing HL7 data. The CBR will look into the MSH segment in the HL7 data to return the transaction type. Since non-XML HL7 data is stored as attachments in a message, the CBR will operate on the attachments to retrieve the TrxId. If there are multiple attachments in the message, only the first attachment will be used.

The following is the XML fragment in the WSDL for the HL7 TrxId:

```
<cbr:trxId type='hl7' />
```

The HL7 TrxID is scheduled to be supported in a future release.

XPath

XPath type TrxId uses XPath to determine routing. In this case, the CBR assumes the incoming message contains XML data. The CBR will not attempt to look into the XML data to determine transaction type. In fact, there will not be any concept of transaction type. This is a special kind of TrxId. The CBR will use the XPath processor to perform routing.

The following is the XML fragment in the WSDL for the XPath TrxId:

```
<trxId type='xpath' />
```

The second form of using XPath as TrxID is more of a traditional TrxID. It uses XPath as an expression to extract the TrxId.

The following is the XML fragment in the WSDL for the XPath TrxId with expression attribute:

```
<trxId type='xpath' expression='Orders/Order' />
```

Script

Script type TrxId allows the user to write a script to determine the TrxID of a MessageExchange. The mechanism here is similar to the UPOC framework defined in the CCSL. The method defined in the script class should always return a string to indicate the result of the TrxId.

The following is the XML fragment in the WSDL for the script TrxId:

```
<trxId type='script' scriptEngine='groovy'
scriptClass='MyTrxIdClass' scriptMethod='trxIdMethod' />
```

8.7.3. CBR Routing Rules

The routingRules element in a routing rules XML file contains one or more routingRule elements. Each routingRule element contains an optional expression element and one target. Each expression element can be a static string or a regular expression or just have an XPath attribute. Each target element must contain one of the following elements:

- service - The namespace qualified name of the service to be invoked.
- interface - The namespace qualified name of the interface to be invoked.

Each target element may also contain these optional elements:

- operation - If the service provides multiple operations, then the operation to invoke may be specified.
- timeout - A timeout period (in milliseconds) may be specified. If no timeout period is specified, the sequencer will wait in indefinite period of time for the called service to return.

If the optional expression is missing, the target is the default target which means all message exchanges will be routed to that endpoint no matter what.

The CBR supports three flavors of expression. The XPath expression is used by the XPath TrxId method. The static string expression is used for exact matching. The regular expression based expression is for regular expression matching.

Example 1: XPath expression

```
<routingRules
  xmlns="http://cbesb.bostechcorp.com/cbr/routingrules" >
  <routingRule>
    <expression type="XPath">
      count(/test:echo) = 1
    </expression>
    <target>
      <interface>transformInterface</interface>
      <timeout>3000</timeout>
    </target>
  </routingRule>
</routingRules>
```

```

        </target>
    </routingRule>
</routingRule>
<!-- there is no expression, this is the default destination -->
    <target>
        <service>transformService2</service>
    </target>
</routingRule>
</routingRules>

```

Example 2: Static string expression

```

<routingRules
  xmlns="http://cbesb.bostechcorp.com/cbr/routingrules" >
  <routingRule>
    <expression type="Exact">FOO</expression>
    <target>
      <interface>transformInterface</interface>
      <timeout>3000</timeout>
    </target>
  </routingRule>
  <routingRule>
    <expression type="Exact">BAR</expression>
    <target>
      <service>transformService2</service>
    </target>
  </routingRule>
</routingRules>

```

Example 3 Regular Expression based

```

<routingRules
  xmlns="http://cbesb.bostechcorp.com/cbr/routingrules" >
  <routingRule>
    <expression type="RegExp">ADT_A0{1-9}</expression>
    <target>
      <interface>transformInterface</interface>
      <timeout>3000</timeout>
    </target>
  </routingRule>
  <routingRule>
    <expression type="RegExp">ORD.*</expression>
    <target>
      <service>transformService2</service>
    </target>
  </routingRule>
</routingRules>

```

8.7.4. Example

This shows an example of a WSDL file for a Content Based Router component.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name='CBR'
  targetNamespace='http://bostechcorp.com/SU/CBR'
  xmlns:tns='http://bostechcorp.com/SU/CBR'
  xmlns:cbr='http://cbesb.bostechcorp.com/wsdl/cbr/1.0'
  xmlns='http://schemas.xmlsoap.org/wsdl/'>
  <portType name='CBR_Interface'>
  </portType>
  <binding name='CBRBinding' type='tns:CBR_Interface'>
    <cbr:binding/>
  </binding>
  <service name='CBR_Service'>
    <port name='CBR' binding='tns:CBRBinding'>
      <cbr:config
        role="provider"
        routingRules="CBR.xml"
        defaultMep="in-out">
        <trxId
          type="fixed"
          length="3"
          offset="0"
        />
      </cbr:config>
    </port>
  </service>
</definitions>
```

The following shows an example of routing rules file:

```
<routingRules
  xmlns="http://cbesb.bostechcorp.com/cbr/routingrules/1.0"
  xmlns:unit1="http://bostechcorp.com/SU/OrdFile"
  xmlns:unit2="http://bostechcorp.com/SU/StatusFile"
  >
  <routingRule>
    <expression type="Exact">ORD</expression>
    <target>
      <service>unit1:OrdFile_Service</service>
      <timeout>-1</timeout>
    </target>
  </routingRule>
  <routingRule>
    <expression type="Exact">STA</expression>
    <target>
      <service>unit2:StatusFile_Service</service>
      <timeout>-1</timeout>
    </target>
  </routingRule>
</routingRules>
```

8.8. Parser Service Engine

8.8.1. Overview

The Parser service engine (SE) component will parse a NormalizedMessage's non-XML message attachment in a MessageExchange into an XML representation based on the MDL message definition. The Parser component defines a “config” tag in the “http://cbesb.bostechcorp.com/wsdl/parser/1.0” namespace.

When a MessageExchange is received by the Parser service engine, the non-XML data from the message is stored as attachments in the NormalizedMessage. The attachments are used as the source for the message parsing. The Parser SE component can be configured to parse fixed, variable or hierarchical data based on the MDL definition, or parse standard EDI data based on the standard EDI X12 definitions. The XML (DOM) representation of parsed result is placed in a new out message of the MessageExchange as the result.

8.8.2. Description

The following table shows the WSDL configuration setting for the Parser SE component:

Name	Required	Description	Default
defaultMep	Y	Type of message exchange to accept as a producer.	in-out is only valid choice
parserType	N	Value to determine what parser to use. The acceptable values are: <ul style="list-style-type: none"> Mdl – Use the ChainBuilder ESB parser for parsing proprietary formatted (fixed, variable and hierarchical) message. X12 – Use the X12 parser to parse standard X12 EDI message. 	Mdl
msgDef	Y	Value to specify the message definition required by different parser. Depending on parseType, the msgDef has different format: <ul style="list-style-type: none"> Mdl : the format is “ProjName::MdlFilePath:Message”. The “MdlFilePath” is the relative file path for a Mdl file. The “Message” is a valid message 	

		<p>definition in the Mdl file.</p> <ul style="list-style-type: none"> • X12: the format is “ProjName::X12ver/Variant/Msg Type”. The variant needs to be a one defined in either ChainBuilder ESB project or Service Assembly project. If variant is empty, the standard X12 message type is used. E.g, the msgDef of “ESB::004010/dell/m270” specify the message type m270 of an X12 v004010 variant named “dell” defined in ESB project. 	
--	--	--	--

8.8.3. Example

This shows an example of a WSDL file for a Parser component.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name='Parser'
  targetNamespace='http://bostechcorp.com/SU/Parser'
  xmlns:tns='http://bostechcorp.com/SU/Parser'
  xmlns:parser='http://cbesb.bostechcorp.com/wsd/Parser/1.0'
  xmlns='http://schemas.xmlsoap.org/wsd/'>
  <portType name='Parser_Interface'>
  </portType>
  <binding name='ParserBinding' type='tns:Parser_Interface'>
    <parser:binding/>
  </binding>
  <service name='Parser_Service'>
    <port name='Parser' binding='tns:ParserBinding'>
      <parser:config
        role="provider"
        defaultMep="in-out"
        msgDef="JBI::src/formats/demo.mdl:FixedMsg"
        parserType="mdl"
      />
    </port>
  </service>
</definitions>
```

8.9. Transformation Service Engine

8.9.1. Overview

The Transformer SE component will perform message transformation based on a TRN map definition file created by the ChainBuilder ESB Map Editor. The source or target message

definition in a TRN file can be XSD, MDL or X12 definition. The Transformer SE component defines a “config” element in the “http://cbesb.bostechcorp.com/wsdl/transformer/1.0” namespace. The transformations do require several other files, some of which are dynamically compiled.

When a MessageExchange is received by the Transformer SE, the XML content or its attachment is used as the source for the transformation based on a specified TRN file. The result of transformation is used to construct a new out message for the MessageExchange as the content or attachment. The MessageExchange will be sent back to the originating component.

8.9.2. Description

The following table shows the WSDL configuration setting for the Transformer SE component:

Name	Required	Description	Default
defaultMep	Y	Type of message exchange to accept as a producer.	in-out is only valid choice
trnFile	Y	Value specifies the name of a TRN file which must be deployed in the Service Unit. The format is “ProjName::TrnFilePath”. The “TrnFilePath” is the relative file path for a TRN file.	

8.9.3. Example

This shows an example of a WSDL file for a Transformer component.

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions name='Transformer'
  targetNamespace='http://bostechcorp.com/cbesb/UseCase4'
  xmlns:tns='http://bostechcorp.com/cbesb/UseCase4'
  xmlns:transformer='http://cbesb.bostechcorp.com/wsdl/transformer/1.0'
  xmlns='http://schemas.xmlsoap.org/wsdl/'
  >
  <portType name='WeatherMapInterface'>
  </portType>
  <binding name='WeatherMapBinding' type='tns:WeatherMapInterface'>
    <transformer:binding />
  </binding>
</definitions>
```

```

</binding>

<service name='WeatherMapService'>
  <port name='WeatherMapEndpoint'
binding='tns:WeatherMapBinding'>
    <transformer:config role="provider"
trnFile="WeatherService::src/xlate/GeocodeToNdfdByDayRequest.trn" />
  </port>
</service>
</definitions>

```

8.10. XSLT Service Engine

8.10.1. Overview

The XSLT SE component will perform message transformation based on an XSLT file. The XSLT SE component defines a “config” element in the “http://cbesb.bostechcorp.com/wsd1/xslt/1.0” namespace.

When a MessageExchange is received by the XSLT component, the XML content is used as the source for the transformation based on a XSLT file. The result of transformation is used to construct a new out message for the MessageExchange as the content or attachment. The MessageExchange will be sent back to the originating component.

8.10.2. Description

The following table shows the WSDL configuration setting for the XSLT SE component:

Name	Required	Description	Default
defaultMep	Y	Type of message exchange to accept as a producer.	in-out is only valid choice
xslLocation	Y	Value specifies a valid XSL transform file.	

8.10.3. Example

This shows an example of a WSDL file for a XSLT component.

```

<?xml version='1.0' encoding='UTF-8'?>
<definitions name='Provider'
  targetNamespace='http://bostechcorp.com/cbesb/UseCase4'

```

```

xmlns:tns='http://bostechcorp.com/cbesb/UseCase4'
xmlns:xslt='http://cbesb.bostechcorp.com/wsd/xslt/1.0'
xmlns='http://schemas.xmlsoap.org/wsd/'>

<portType name='ProviderInterface'>
</portType>

<binding name='ProviderBinding' type='tns:ProviderInterface'>
  <xslt:binding/>
</binding>

<service name='XSLTService'>
  <port name='XSLTEndpoint' binding='tns:ProviderBinding'>
    <xslt:config xslLocation="myxslt.xsl" role="provider"
defaultMep='in-out' />
  </port>
</service>

</definitions>

```

8.11. Script Service Engine

8.11.1. Overview

The Script SE component will perform specific logic as defined in a Groovy script or a POJO (Plain Old Java Objects) class. The Script SE component defines a “config” element in the “http://cbesb.bostechcorp.com/wsd/script/1.0” namespace.

The Script SE can be used in both Consumer and Provider modes. In Provider mode, the only defaultMEP is in-out. When a MessageExchange is received by the Script component, it will execute the defined Groovy script or POJO class on the MessageExchange. The result of this execution is used to construct a new “out” message for the MessageExchange as the content or attachment. The MessageExchange will be sent back to the originating component.

In Consumer mode, the Groovy script or POJO class is executed on the timer basis, the Groovy script or POJO class is responsible to create a MessageExchange with the “in” message set. The MessageExchange is sent to NMR. If the defaultMEP is In-Only, there is nothing the script needs to do at that point. If the defaultMEP is In-Out, the MessageExchange will be received by the Script component from NMR. The run() method defined in the script or POJO class will be executed on the received MessageExchange.

8.11.2. Description

The following table shows the WSDL configuration setting for the Script SE component:

Name	Required	Description	Default
defaultMep	Y	Type of message exchange to accept as a producer.	in-out
role	Y	Whether is a consumer or provider.	provider
triggerTime	N	The timer in milliseconds to trigger the calling of the scripting method. This attribute is only valid when the role is the consumer. The trigger time allows setting up a simple time driven consumer by implementing a time() method in the user's script or class. If a triggerTime is not specified then the user is expected to set up an appropriate listener thread in their start() method and shut it down in their stop() method.	
type	Y	Script type. We support to types: <ul style="list-style-type: none"> ● Groovy: Use the Groovy engine as scripting. ● Pojo: Use Java (Plain Old Java Objects) as way of scripting. 	groovy
class	Y	Java or Groovy class. For Groovy script, this is the file name.	

8.11.3. Deployment Descriptor Example

This shows an example of a WSDL file for a Script component.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name='ScriptPojoTest_Script'
  targetNamespace='http://bostechcorp.com/SU/ScriptPojoTest_Script'
  xmlns:tns='http://bostechcorp.com/SU/ScriptPojoTest_Script'
  xmlns:script='http://cbesb.bostechcorp.com/wsd1/script/1.0'
  xmlns='http://schemas.xmlsoap.org/wsd1/'>
  <portType name='ScriptPojoTest_Script_Interface'>
  </portType>
  <binding name='ScriptPojoTest_ScriptBinding'
type='tns:ScriptPojoTest_Script_Interface'>
  <script:binding/>
  </binding>
  <service name='ScriptPojoTest_Script_Service'>
  <port name='ScriptPojoTest_Script'
binding='tns:ScriptPojoTest_ScriptBinding'>
  <script:config
  role="consumer"
```

```

        triggertime="1000"
        defaultMep="in-out"
        type="Pojo"
        class="com.bostechcorp.cbesb.test.TestScript"
    />
</port>
</service>
</definitions>

```

8.11.4. IScriptObject Class

The POJO class used in the Script component needs to implement the **IScriptObject** interface. The following is the source code for **IScriptObject** interface:

```

package com.bostechcorp.cbesb.runtime.ccs1.lib;

import java.util.LinkedList;
//import java.util.logging.Logger;
import org.apache.commons.logging.Log;

import javax.jbi.component.ComponentContext;
import javax.jbi.messaging.DeliveryChannel;
import javax.jbi.messaging.MessageExchange;

/**
 * Classes for Script Components must implement this interface
 */

public interface IScriptObject {
    /**
     * Start mode can be used for initialization. Consumer endpoints
     * that do not use timed mode should set up a thread to generate
     * exchanges in start mode
     */
    public void start(Log logger, String rootDir, ComponentContext
componentContext,
        DeliveryChannel channel) throws Exception;

    /**
     * The stop method should clean up resources
     */
    public void stop(Log logger, String rootDir, ComponentContext
componentContext,
        DeliveryChannel channel) throws Exception;

    /**
     * The run method is called when an exchange is sent to the
     endpoint.
     */
    public void run(Log logger, String rootDir, ComponentContext
componentContext,
        DeliveryChannel channel, MessageExchange exchange)
throws Exception;

    /**

```

```

    * The time method is called for a timed mode consumer endpoint.
    * It returns a linked list of exchanges to be sent from the
    * endpoint.
    *
    */

    public LinkedList time(Log logger, String rootDir,
ComponentContext componentContext,
        DeliveryChannel channel, MessageExchange exchange)
throws Exception;
}

```

8.11.5. Example

The following shows an example of POJO class used in Script component's Provider mode.

```

package com.bostechcorp.cbesb.test;

import java.util.LinkedList;
import org.apache.commons.logging.Log;

import javax.jbi.component.ComponentContext;
import javax.jbi.messaging.DeliveryChannel;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.NormalizedMessage;

import com.bostechcorp.cbesb.runtime.ccs1.nmhandler.NormalizedMessageHandler;
import com.bostechcorp.cbesb.runtime.ccs1.lib.DumpNormalizedMessage;
import com.bostechcorp.cbesb.runtime.ccs1.nmhandler.StringSource;

import com.bostechcorp.cbesb.runtime.ccs1.lib.IScriptObject;

public class TestScript implements IScriptObject {

    public void run(Log log, String rootDir, ComponentContext
componentContext,
        DeliveryChannel channel, MessageExchange exchange)
throws Exception
    {
        log.info("TestScript java class - run");
        NormalizedMessage outMsg = exchange.getMessage("out");
        log.info("Out Message: " +
DumpNormalizedMessage.dump(outMsg));
    }

    public void start(Log log,String rootDir, ComponentContext
componentContext,
        DeliveryChannel channel) throws Exception
    {
    }
}

```

```

    public void stop(Log log, String rootDir,ComponentContext
componentContext,
        DeliveryChannel channel) throws Exception
    {
    }

    public LinkedList time(Log log,String rootDir, ComponentContext
componentContext,
        DeliveryChannel channel, MessageExchange exchange)
throws Exception
    {
        LinkedList sendList = new LinkedList();

        log.info("TestScript java class - time");
        NormalizedMessage inMsg = exchange.getMessage("in");

        //Add some data to the in message
        NormalizedMessageHandler nmh = new
NormalizedMessageHandler(inMsg);
        StringSource strSrc = new
StringSource("ST*270*D10000054~S2S*JE*BLOCK***Q~BHT*AB12*AB~HL*ABCDE**1
1~TRN*AB*ABCDE12345~NM1*VN*2*HEALTHCARE DATA
EXCHANGE*****ZZ*00000000609~REF*F1*3.0~N4*HOPEWELL*VA*23860~PER*PZ**WP*
(610)2191385~PRV*SB*ZZ*541779911~DMG*D8*19740529*M~INS*Y*18~DTP*150*D8*
20000322~DTP*151*D8*20000322~EQ*1~AMT*11*5000.00~REF*REF*F1*3.0~SE*56*D
10000054~");
        nmh.addRecord(strSrc);
        nmh.generateMessageContent();

        sendList.add(exchange);
        return sendList;
    }
}

```

The next example shows a Groovy script used in the Script component's Consumer mode with same functionality as the POJO class listed above.

```

import java.util.LinkedList;
import java.util.logging.Logger;

import javax.jbi.component.ComponentContext;
import javax.jbi.messaging.DeliveryChannel;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.MessageExchangeFactory;
import javax.jbi.messaging.InOnly;
import javax.jbi.messaging.InOut;
import javax.jbi.messaging.NormalizedMessage;

import javax.xml.transform.Source;

import
com.bostechcorp.cbesb.runtime.ccs1.nmhandler.NormalizedMessageHandler;
import com.bostechcorp.cbesb.runtime.ccs1.lib.DumpNormalizedMessage;
import com.bostechcorp.cbesb.runtime.ccs1.nmhandler.StringSource;

```

```

class InOutConsumer {

    def static start(log, rootDir, componentContext, channel) {
        log.info("InOutConsumer groovy script - start");
    }

    def static stop(log, rootDir, componentContext, channel) {
        log.info("InOutConsumer groovy script - stop");
    }

    def static time(log, rootDir, componentContext, channel, exchange)
    {
        log.info("InOutConsumer groovy script - time");
        NormalizedMessage inMsg = exchange.getMessage("in");

        //Add some data to the in message
        NormalizedMessageHandler nmh = new
NormalizedMessageHandler(inMsg);
        StringSource strSrc = new
StringSource("ST*270*D10000054~S2S*JE*BLOCK***Q~BHT*AB12*AB~HL*ABCDE**1
1~TRN*AB*ABCDE12345~NM1*VN*2*HEALTHCARE DATA
EXCHANGE*****ZZ*00000000609~REF*F1*3.0~N4*HOPEWELL*VA*23860~PER*PZ**WP*
(610)2191385~PRV*SB*ZZ*541779911~DMG*D8*19740529*M~INS*Y*18~DTP*150*D8*
20000322~DTP*151*D8*20000322~EQ*1~AMT*11*5000.00~REF*REF*F1*3.0~SE*56*D
10000054~");
        nmh.addRecord(strSrc);
        nmh.generateMessageContent();
    }

    def static run(log, rootDir, componentContext, channel, exchange)
    {
        log.info("InOutConsumer groovy script - run");
        NormalizedMessage outMsg = exchange.getMessage("out");
        log.info("Out Message: " +
DumpNormalizedMessage.dump(outMsg));
    }

    static void main(args) {
    }
}

```

8.12. JDBC Service Engine

8.12.1. Overview

The JDBC SE Component accepts request messages that execute SQL statements. A response message is returned which contains information about the state of the request as well as possible row results. The way in which rows are returned is configurable by setting a "page size". The page size is the maximum number of rows to return in a single message. If a statement returns more rows than the page size, then subsequent requests must be sent to retrieve the remaining rows.

The component will be able to maintain multiple sessions. A session contains the resources needed for a connection to the database. When a new request is received, it may or may not contain a session ID. If there is no session ID in the request, then a new session is created to process the request. The session ID is then returned with the response message. If the request contains a session ID, then the request is processed using that existing session. If a session is idle for a preset timeout interval, then the session should be released to free resources. If a request is received with a session ID that does not exist, then an error response should be returned.

The JDBC SE component defines a "config" element in the "http://cbesb.bostechcorp.com/wsd1/jdbc/1.0" namespace.

8.12.2. Description

The following table shows the WSDL configuration setting for the JDBC SE component:

Name	Required	Description	Default
driver	yes	The fully qualified class name of the JDBC driver. For example: com.microsoft.jdbc.sqlserver.SQLServerDriver	
url	yes	The driver specific URL that specifies the connection. For example: jdbc:Microsoft:sqlserver://SQLHost01:1433;database Name=testdb	
user	no	The user name to use to log into the database.	
password	no	The password to use to log into the database.	
requestHandler	yes	The handler class to use when processing request messages. It is responsible for parsing the request message to create an executable request object. A single instance of this class is used by each endpoint to process all requests.	com.bostechcorp.cbesb.runtime.component.jdbc.processors.JdbcDefaultRequestHandler
execHandler	yes	The handler class to use when executing a request. Each session will have its own instance of this class so	com.bostechcorp.cbesb.run

		session specific data may be kept in the member variables.	time.compon ent.jdbc.proc essors.JdbcD efaultExecHa ndler
autoCommit	no	If set to true, each successful request is committed automatically. If set to false, then the user is responsible for sending a Commit or Rollback to handle processing of transactions.	true
connectionRetries	no	When trying to establish a connection to the database, if there is a failure, it will make this many attempts to connect before erroring out.	3
connectionInterval	no	The number of milliseconds to sleep between reconnect attempts.	3000
transactionTimeout	no	The timeout in milliseconds to keep a transaction open before freeing the resources when there is no activity.	30000 (5 minutes)
defaultPageSize	no	The default number of rows to return in a single response. This may be overridden in the request message.	-1

8.12.3. Deployment Descriptor Example

This shows an example of a WSDL file for a JDBC component.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name='FiletoJdbcUseCase2_JDBC'
  targetNamespace='http://bostechcorp.com/SU/FiletoJdbcUseCase2_JDBC'
  C'
  xmlns:tns='http://bostechcorp.com/SU/FiletoJdbcUseCase2_JDBC'
  xmlns:jdbc='http://cbesb.bostechcorp.com/wsd1/jdbc/1.0'
  xmlns='http://schemas.xmlsoap.org/wsd1/'>
  <portType name='FiletoJdbcUseCase2_JDBC_Interface'>
  </portType>
  <binding name='FiletoJdbcUseCase2_JDBCBinding'
  type='tns:FiletoJdbcUseCase2_JDBC_Interface'>
  <jdbc:binding/>
  </binding>
  <service name='FiletoJdbcUseCase2_JDBC_Service'>
  <port name='FiletoJdbcUseCase2_JDBC'
  binding='tns:FiletoJdbcUseCase2_JDBCBinding'>
  <jdbc:config
  role="provider"
  connectionInterval="3000"
  user="cbesb"
  password="cbesb"
```

```

        autoCommit="false"

requestHandler="com.bostechcorp.cbesb.runtime.component.jdbc.processors
.JdbcDefaultRequestHandler"
        transactionTimeout="30000"
        defaultPageSize="-1"
        connectionRetries="3"

url="jdbc:microsoft:sqlserver://192.168.1.231:1433;database=pubs;Select
Method=Cursor;"

execHandler="com.bostechcorp.cbesb.runtime.component.jdbc.processors.Jd
bcDefaultExecutionHandler"
        driver="com.microsoft.jdbc.sqlserver.SQLServerDriver"
    />
</port>
</service>
</definitions>

```

8.12.4. Message Formats

The JDBC Component uses a request message with the root tag "jdbc_request" and a response message with the root tag "jdbc_response". If an operation spans more than one request-response, then the response will contain a session ID. This session ID must be included in all subsequent requests for the operation.

The request message may contain either a "transaction" tag, an "execute" tag, or a "get_page" tag.

Request Message - transaction

This type of request is used to indicate the beginning or end of a transaction and to perform a commit or rollback. The transaction element must contain a string with one of the following values:

- COMMIT - Commits pending operations for the session.
- ROLLBACK - Rolls back the pending operations for the session.
- BEGIN - Starts a new transaction for the session
- END - Indicates the end of a transaction for the session, if all operations since the beginning of the transaction were executed successfully, then the transaction is committed. If one or more operations failed since the beginning of the transaction, then it is rolled back.

A session ID must be provided with all requests except BEGIN.

Examples:

```

<jdbc_request xmlns="http://cbesb.bostechcorp.com/jdbc/1.0">
  <transaction>BEGIN</transaction>
</jdbc_request>

```



```
<jdbc_request xmlns="http://cbesb.bostechcorp.com/jdbc/1.0"
  sessionId="2399823459">
  <transaction>COMMIT</transaction>
</jdbc_request>

<jdbc_request xmlns="http://cbesb.bostechcorp.com/jdbc/1.0"
  sessionId="2399823459">
  <transaction>ROLLBACK</transaction>
</jdbc_request>

<jdbc_request xmlns="http://cbesb.bostechcorp.com/jdbc/1.0"
  sessionId="2399823459">
  <transaction>END</transaction>
</jdbc_request>
```

Request Message - execute

This type of request executes a SQL statement. It contains a child element called "statement" which contains the actual SQL statement. If the SQL statement is a stored procedure call, it should be wrapped in braces {} so it is executed properly.

The statement element may have a "pageSize" attribute. The value of this attribute is an integer greater than zero or equal to -1. It indicates the number of rows to return in a single response. If -1 is specified, then all rows are returned in a single response.

The SQL statement may contain question mark placeholders that indicate the location where variable values will be substituted. The values are provided in the message in an element called "vars" which contains one or more "var" elements. Each "var" contains a datatype, mode and value.

The datatype is provided as an attribute of the "var" element and must be one of the following values:

- CHAR
- VARCHAR
- LONGVARCHAR
- NUMERIC
- DECIMAL
- BIT
- TINYINT
- SMALLINT
- INTEGER
- BIGINT
- REAL
- FLOAT
- DOUBLE
- BINARY
- VARBINARY
- LONGVARBINARY

- DATE
- TIME
- TIMESTAMP

The mode is also provided as an attribute of the "var" element and must be one of the following values:

- IN - Represents an input value.
- OUT - Represents an output value and will be populated in the response message. This is only used in stored procedure calls.
- INOUT - Represents an input value as well as an output value that will be populated in the response message. This is only used in stored procedure calls.

If the statement being executed is part of a transaction, then the session ID of that transaction must be provided. If the statement is atomic, then no session ID is required.

Examples:

```
<jdbc_request xmlns="http://cbesb.bostechcorp.com/jdbc/1.0">
  <execute>
    <statement pageSize="10">select description, price from Items</statement>
  </execute>
</jdbc_request>
```

```
<jdbc_request xmlns="http://cbesb.bostechcorp.com/jdbc/1.0">
  <execute>
    <statement>select description, price from Items where sku=?</statement>
    <vars>
      <var mode="IN" datatype="VARCHAR">3749207201</var>
    </vars>
  </execute>
</jdbc_request>
```

```
<jdbc_request xmlns="http://cbesb.bostechcorp.com/jdbc/1.0"
sessionId="843054702">
  <execute>
    <statement>delete from stores where stor_id = ?</statement>
    <vars>
      <var mode="IN" datatype="CHAR">1234</var>
    </vars>
  </execute>
</jdbc_request>
```

```
<jdbc_request xmlns="http://cbesb.bostechcorp.com/jdbc/1.0"
sessionId="843054702">
  <execute>
    <statement>{call sp_insertAuthor(?, ?, ?, ?, ?, ?, ?, ?, ?, ?
)}</statement>
    <vars>
      <var datatype="VARCHAR" mode="IN">172-32-1176</var>
      <var datatype="VARCHAR" mode="IN">White</var>
      <var datatype="VARCHAR" mode="IN">Johnson</var>
      <var datatype="CHAR" mode="IN">408 496-7223</var>
      <var datatype="VARCHAR" mode="IN">10932 Bigge Rd.</var>
      <var datatype="VARCHAR" mode="IN">Menlo Park</var>
      <var datatype="CHAR" mode="IN">CA</var>
      <var datatype="CHAR" mode="IN">94025</var>
```

```

    <var datatype="BIT" mode="IN">true</var>
    <var datatype="INTEGER" mode="OUT" />
  </vars>
</execute>
</jdbc_request>

```

Request Message - get_page

This type of request is used to request a page of row results from a previously executed statement. The `get_page` element contains a string value for its content that must be an integer greater than 0 which references an absolute page number or the value "NEXT" or "PREVIOUS". A session ID must be provided in the request.

Examples:

```

<jdbc_request xmlns="http://cbesb.bostechcorp.com/jdbc/1.0"
  sessionId="2399823459">
  <get_page>4</get_page>
</jdbc_request>

```

```

<jdbc_request xmlns="http://cbesb.bostechcorp.com/jdbc/1.0"
  sessionId="2399823459">
  <get_page>NEXT</get_page>
</jdbc_request>

```

```

<jdbc_request xmlns="http://cbesb.bostechcorp.com/jdbc/1.0"
  sessionId="2399823459">
  <get_page>PREVIOUS</get_page>
</jdbc_request>

```

Response Message

The response message contains a unique session ID attribute that can be used to issue additional requests on the same connection. It also contains the following elements:

- `success` - true or false, indicates if the request was executed successfully.
- `error` - (optional) If `success = false`, then this element will be populated with the error that occurred. If `success = true`, then this element will not be present in the response.
- `total_rows` - (optional) Indicates the total number of rows returned by a query.
- `total_pages` - (optional) Indicates the total number of pages returned by the query. This will be equal to `total_rows/pageSize`.
- `current_page` - (optional) Indicates the page number of the rows returned in this response.
- `rows_affected` - (optional) Indicates the number of rows affected by the executed statement.
- `vars` - (optional) If the request had any var elements with `mode = "OUT"` or `"INOUT"`, then they are provided here with their corresponding output value.
- `rows` - (optional) If the request returned row data, it is returned inside this element. One or more "row" elements are contained in the rows element up to a maximum of

page size. Each "row" element contains an element for each column in the row. The column name is used as the child element name and the value of that column is the value of that element.

Examples:

```
<jdbc_response xmlns="http://cbesb.bostechcorp.com/jdbc/1.0"
  sessionId="2399823459">
  <success>true</success>
  <total_rows>1054</total_rows>
  <total_pages>1054</total_pages>
  <current_page>1</current_page>
  <rows>
    <row><productID>759302931</productID><description>USB
    Cable</description><unitPrice>15.99</unitPrice></row>
  </rows>
</jdbc_response>

<jdbc_response xmlns="http://cbesb.bostechcorp.com/jdbc/1.0"
  sessionId="239987490">
  <success>>false</success>
  <error>ERROR 1146 (42S02): Table 'test.foo' doesn't exist</error>
</jdbc_response>

<jdbc_response xmlns="http://cbesb.bostechcorp.com/jdbc/1.0"
  sessionId="2399846783">
  <success>true</success>
  <rows_affected>1</rows_affected>
  <vars>
    <var mode="OUT" datatype="INTEGER">0</var>
  </vars>
</jdbc_response>
```

8.12.5. Message Definition Schema

The following schema defines both the jdbc_request and jdbc_response messages.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://cbesb.bostechcorp.com/jdbc/1.0"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
  targetNamespace="http://cbesb.bostechcorp.com/jdbc/1.0">

  <xsd:element name="jdbc_request">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="transaction" type="transactionType"/>
        <xsd:element name="execute" type="executeType"/>
        <xsd:element name="get_page" type="getPageType"/>
      </xsd:choice>
      <xsd:attribute name="sessionId" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>
```

```

<xsd:simpleType name="transactionType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="COMMIT"/>
    <xsd:enumeration value="ROLLBACK"/>
    <xsd:enumeration value="BEGIN"/>
    <xsd:enumeration value="END"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="executeType">
  <xsd:sequence>
    <xsd:element name="statement" type="statementType"/>
    <xsd:element minOccurs="0" name="vars" type="varsType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="getPageType">
  <xsd:union>
    <xsd:simpleType>
      <xsd:restriction base="xsd:positiveInteger"/>
    </xsd:simpleType>
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="NEXT"/>
        <xsd:enumeration value="PREVIOUS"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>

<xsd:complexType name="statementType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="pageSize" type="xsd:integer"/>
      <xsd:attribute name="keepOpen" type="xsd:boolean"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="varsType">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" name="var" type="varType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="varType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">

```

```

    <xsd:attribute name="datatype" type="datatype" use="required"/>
    <xsd:attribute name="mode" type="modeType" use="required"/>
  </xsd:extension>
</xsd:simpleContent>
</xsd:complexType>

<xsd:simpleType name="datatype">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="CHAR"/>
    <xsd:enumeration value="VARCHAR"/>
    <xsd:enumeration value="LONGVARCHAR"/>
    <xsd:enumeration value="NUMERIC"/>
    <xsd:enumeration value="DECIMAL"/>
    <xsd:enumeration value="BIT"/>
    <xsd:enumeration value="TINYINT"/>
    <xsd:enumeration value="SMALLINT"/>
    <xsd:enumeration value="INTEGER"/>
    <xsd:enumeration value="BIGINT"/>
    <xsd:enumeration value="REAL"/>
    <xsd:enumeration value="FLOAT"/>
    <xsd:enumeration value="DOUBLE"/>
    <xsd:enumeration value="BINARY"/>
    <xsd:enumeration value="VARBINARY"/>
    <xsd:enumeration value="LONGVARBINARY"/>
    <xsd:enumeration value="DATE"/>
    <xsd:enumeration value="TIME"/>
    <xsd:enumeration value="TIMESTAMP"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="modeType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="IN"/>
    <xsd:enumeration value="OUT"/>
    <xsd:enumeration value="INOUT"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:element name="jdbc_response">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="success" type="xsd:boolean"/>
      <xsd:element minOccurs="0" name="error" type="xsd:string"/>
      <xsd:element minOccurs="0" name="total_rows" type="xsd:integer"/>
      <xsd:element minOccurs="0" name="total_pages" type="xsd:integer"/>
      <xsd:element minOccurs="0" name="current_page" type="xsd:integer"/>
      <xsd:element minOccurs="0" name="rows_affected" type="xsd:integer"/>
      <xsd:element minOccurs="0" name="vars" type="varsType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

```

```

        <xsd:element minOccurs="0" name="rows" type="rowsType"/>
    </xsd:sequence>
    <xsd:attribute name="sessionId" type="xsd:string"/>
</xsd:complexType>
</xsd:element>

<xsd:complexType name="rowsType">
    <xsd:sequence>
        <xsd:element maxOccurs="unbounded" name="row" type="xsd:anyType"/>
    </xsd:sequence>
</xsd:complexType>

</xsd:schema>

```

8.12.6. Handler Classes

The JDBC Component has two types of handlers, a request handler and an execution handler. The request handler is used to process the input message and create an executable `JdbcRequest` object. The execution handler is responsible for the actual execution of the request. Depending on the settings contained by the `JdbcRequest`, different methods are called in the execution handler. Each endpoint will have a single instance of the specified request handler, but each session will have its own instance of the specified execution handler. The component will come with a default implementation of each type of handler that performs the behavior described in the previous sections.

The interfaces that the handlers must implement are:

```

package com.bostechcorp.cbesb.runtime.component.jdbc.processors;

import javax.jbi.messaging.NormalizedMessage;

/**
 * An implementation of this interface is used by the JDBC component
 * to process messages received and turn them into a JdbcRequest that
 * can be executed. A single instance of the class is instantiated
 * for each endpoint.
 */
public interface IJdbcRequestHandler {

    /**
     * Called when a request message is received by the endpoint. This
     method
     * should create a new JdbcRequest using the data in the received request
     message.
     * @param inMsg The received request message
     * @return A new JdbcRequest object
     * @throws JdbcException
     */
    public JdbcRequest createRequest(NormalizedMessage inMsg) throws
    JdbcException;

    /**
     * This method is called if an exception occurs during any of the
     processing.

```

```

        * The handler implementation can take the thrown exception and create a
response
        * message to send back to the service consumer.
        * @param e The Exception that was thrown.
        * @param outMsg Response message that will be returned to consumer.
        */
    public void handleError(Exception e, NormalizedMessage outMsg, long
sessionId);
}

```

```

package com.bostechcorp.cbesb.runtime.component.jdbc.processors;

import javax.jbi.messaging.NormalizedMessage;
import com.bostechcorp.cbesb.runtime.jdbc.JdbcSession;
import com.bostechcorp.cbesb.runtime.component.jdbc.JdbcEndpoint;

/**
 * The interface that all JDBC execution handlers must implement.
 * Each session will have its own instance of this class.
 */
public interface IJdbcExecutionHandler {

    /**
     * Called when the instance of the handler is created to perform
     * any necessary initialization.
     * @param endpoint
     */
    public void initialize(JdbcEndpoint endpoint);

    /**
     * Called when a Begin Transaction type of request is received.
     * @param session The JdbcSession that contains the connection to
     * the database.
     * @param outMsg Response message that will be returned to consumer.
     * @throws JdbcException
     */
    public void processTransactionBegin(JdbcSession session,
NormalizedMessage outMsg) throws JdbcException;

    /**
     * Called when an End Transaction type of request is received.
     * @param session The JdbcSession that contains the connection to
     * the database.
     * @param outMsg Response message that will be returned to consumer.
     * @throws JdbcException
     */
    public void processTransactionEnd(JdbcSession session, NormalizedMessage
outMsg) throws JdbcException;

    /**
     * Called when a Commit Transaction type of request is received.
     * @param session The JdbcSession that contains the connection to
     * the database.
     * @param outMsg Response message that will be returned to consumer.
     * @throws JdbcException
     */
}

```



```
    public void processTransactionCommit(JdbcSession session,
NormalizedMessage outMsg) throws JdbcException;

    /**
     * Called when a Rollback Transaction type of request is received.
     * @param session The JdbcSession that contains the connection to
     * the database.
     * @param outMsg Response message that will be returned to consumer.
     * @throws JdbcException
     */
    public void processTransactionRollback(JdbcSession session,
NormalizedMessage outMsg) throws JdbcException;

    /**
     * Called when an Execute type of request is received.
     * @param session The JdbcSession that contains the connection to
     * the database.
     * @param request The request object that contains the SQL statement to
execute.
     * @param outMsg Response message that will be returned to consumer.
     * @throws JdbcException
     */
    public void processExecute(JdbcSession session, JdbcRequest request,
NormalizedMessage outMsg) throws JdbcException;

    /**
     * Called when a get_page type of request is recieved asking for a
specific page number.
     * @param session The JdbcSession that contains the connection to
     * the database.
     * @param pageNumber The page number to retrieve.
     * @param outMsg Response message that will be returned to consumer.
     * @throws JdbcException
     */
    public void processGetPage(JdbcSession session, int pageNumber,
NormalizedMessage outMsg) throws JdbcException;

    /**
     * Called when a get_page type of request is received asking for the next
page.
     * @param session The JdbcSession that contains the connection to
     * the database.
     * @param outMsg Response message that will be returned to consumer.
     * @throws JdbcException
     */
    public void processGetNextPage(JdbcSession session, NormalizedMessage
outMsg) throws JdbcException;

    /**
     * Called when a get_page type of request is received asking for the
previous page.
     * @param session The JdbcSession that contains the connection to
     * the database.
     * @param outMsg Response message that will be returned to consumer.
     * @throws JdbcException
     */
    public void processGetPreviousPage(JdbcSession session, NormalizedMessage
outMsg) throws JdbcException;
}

```


9. ChainBuilder ESB Community

ChainForge.net is the internet's premier destination to share ChainBuilder and JBI knowledge with your peers.

Join the ChainBuilder ESB Community:

<http://www.chainforge.net/community>

As a member you can view content and contribute to a Forum:

<http://www.chainforge.net/community/forums.html>

Read ChainBuilder ESB related Blogs:

<http://www.chainforge.net/blogs>

Appendix A HTTP UPOC Groovy Source Code

This is the source code for the present UPOC used in samples/UseCase4 on the HTTP server endpoint.

```
import com.bostechcorp.cbesb.runtime.ccslib.*;
import java.util.logging.Logger;

import javax.jbi.component.ComponentContext;
import javax.jbi.messaging.DeliveryChannel;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.NormalizedMessage;

import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import com.bostechcorp.cbesb.runtime.ccslib.nmhandler.StringSource;
import
com.bostechcorp.cbesb.runtime.ccslib.nmhandler.NormalizedMessageHandler;
import javax.xml.transform.stream.StreamSource;

// This is the present user script from the weather service example
// (UseCase4).
// The incoming message contains HTML form data with a "zip" field.
// First, we convert the message content to a String and extract the
// zipcode.
// Then we use an HTTP GET to the Yahoo geocode service to get the
// latitude
// and longitude (see
// http://developer.yahoo.com/maps/rest/V1/geocode.html).
// Next we combine the geocode data with today's date to create an xml
// request.
// Finally, we put the message content with the new xml.
def HTTPPresentend(log, context, componentContext, channel, exchange) {
    LinkedList sendList = new LinkedList()

    // Print some banners to make it easier to find in the log.
    // We should really use the log object but this is easier.
    println "\n\n"
    println "Running HTTPPresentend UPOC in groovyscript"
    println "\n\n"

    // First, convert the message content into a String form. We only
    // support StreamSource
    // since that's what we always get from the HTTP component.
    NormalizedMessage inMessage = exchange.getMessage("in")
    Source content = inMessage.getContent();
    String stringContent

    if (content instanceof StreamSource) {
        try {
```

```

        StringBuffer stringBufferContent = new StringBuffer()
        InputStream is = content.getInputStream()
        int inChar
        while ((inChar = is.read()) > 0)
stringBufferContent.append((char)inChar)
            stringBufferContent = new String(stringBufferContent);
        }
        catch (Exception e) {
            println "Exception converting content to String:
"+e+"\n"
            // throw back the exception so that CCSL can deal
with it
            throw e
        }
    } else throw new Exception("Error, expected StreamSource content
but got "+content);

    // Extract the zip code from the String content
    println("in message content=["+stringContent+"]")
    int startZipField = stringContent.indexOf("&zip");
    if (startZipField < 0)
        throw new Exception("zip field was not found in the message
data")
    int startZipValue = stringContent.indexOf('=', startZipField+1);
    if (startZipValue < 0)
        throw new Exception("zip field was not found in the message
data")
    String zipString=stringContent.substring(startZipValue+1);
    println "zipString=["+zipString+"]"

    // Convert the zipcode into a Yahoo geocode xml message
    String zipcodeInfo = getZipcodeInformation(zipString);

    // Doctor up the zip code info with today's date
    Calendar today = Calendar.getInstance();
    String todayString = ""+today.get(Calendar.YEAR)+"-
"+(today.get(Calendar.MONTH)+1)+"-"+today.get(Calendar.DAY_OF_MONTH)
    zipcodeInfo =
zipcodeInfo.replace("<Latitude", "<DateToday>"+todayString+"</DateToday>
<Latitude");

    // Update the message content. The NormalizedMessageHandler adds
the ChainBuilder ESB
    // DataEnvelope. First null out the content so that the HTML
content doesn't break
    // anything. The NormalizedMessageHandler determines the record
type from the object type.
    // newMessageContent must be a StreamSource or a DOMSource for it
to be
    // enveloped as XML by the NormalizedMessageHandler. That's why
the next line is so long.
    StreamSource newMessageContent = new StreamSource((new
StringSource(zipcodeInfo)).getInputStream());

```

```

        exchange.getMessage("in").setContent(null)
        NormalizedMessageHandler nmh = new
NormalizedMessageHandler(exchange.getMessage("in"))
        nmh.addRecord(newMessageContent);
        nmh.generateMessageContent()

        // See what it looks like. DumpNormalizedMessage is in the
ccsl.lib package
        println "\n\nTVOLLE - NEW
MESSAGE\n"+DumpNormalizedMessage.dump(exchange.getMessage("in"))+"\n\n\
n"

        // Send back the updated exchange
        sendList.add(exchange)
        return sendList
    }

// This uses an HTTP GET call to the Yahoo geocode application to
convert the zipcode into
// a more complete data record.
def getZipcodeInformation (zip) {
    URL url = new
URL("http://api.local.yahoo.com/MapsService/V1/geocode?appid=YahooDemo&
zip="+zip)
    HttpURLConnection huc = (HttpURLConnection) url.openConnection();
    huc.setRequestMethod("GET")
    huc.connect()
    InputStream is = huc.getInputStream()
    StringBuffer response = new StringBuffer()
    int code = huc.getResponseCode()
    if (code >= 200 && code < 300) {
        int ch;
        while((ch=is.read())>0) {
            response.append((char)ch)
        }
    }
    huc.disconnect()
    return new String(response)
}
}

```

Appendix B. Log output from HTTP UPOC

This shows the log output generated from the groovy script. A zip code comes in and an enveloped xml record goes out.

Running HTTPPresent UPOC in groovyscript

```
in message content=[ofield=&zip=43040]
zipString=[43040]
```

TVOLLE - NEW MESSAGE

```
content=javax.xml.transform.dom.DOMSource@58d7c2
<?xml version="1.0" encoding="UTF-8"?>
<DataEnvelope>
<XMLRecord>
<ResultSet xmlns="urn:yahoo:maps"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:yahoo:maps http://api.
local.yahoo.com/MapsService/V1/GeocodeResponse.xsd">
<Result precision="zip">
<DateToday>2006-11-11</DateToday>
<Latitude>40.2483</Latitude>
<Longitude>-83.3671</Longitude>
<Address/>
<City>MARYSVILLE</City>
<State>OH</State>
<Zip>43040</Zip>
<Country>US</Country>
</Result>
</ResultSet>
</XMLRecord>
</DataEnvelope>
```

```
property(javax.jbi.messaging.protocol.headers){Connection=keep-alive,
Content-Length=20, Host=localhost:8192, User-Agent=Mozilla/4.0
(compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.0.3705; .NET
CLR 2.0.50727), Accept-Encoding=gzip, deflate, Accept-Language=en-us,
Content-Type=application/x-www-form-urlencoded, Cache-Control=no-cache,
Accept=image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/x-shockwave-flash, application/vnd.ms-excel,
application/vnd.ms-powerpoint, application/msword, /*/*}
getSecuritySubject=null
```

```
DEBUG - DeliveryChannelImpl - Send ID:TVOLLE-P25-2458-
1163296001773-1:0 in DeliveryChannel{servicemix-http}
DEBUG - DeliveryChannelImpl - Sent: InOut[
id: ID:TVOLLE-P25-2458-1163296001773-1:0
```


Appendix C. Error Database Schema

```

create table Error(
    ErrorId bigint not null generated always as identity
        constraint pk_Error primary key,
    ErrorDateTime timestamp not null,
    ExceptionString varchar(512) not null,
    StackTrace varchar(4096) not null,
    ExchangeId bigint not null constraint un_01 unique);

create table Exchange(
    ExchangeId bigint not null generated always as identity
        constraint pk_Exchange primary key,
    Role varchar(8) not null,
    EndpointService varchar(256) not null,
    EndpointName varchar(256) not null,
    ExchangeContainerId varchar(256) not null,
    InterfaceName varchar(256),
    Operation varchar(256),
    Pattern varchar(256),
    Service varchar(256),
    ExchangeStatus varchar(10) not null);

create table ExchangeProperty(
    ExchangeId bigint not null constraint fk_ExchangeProperty
        references Exchange on delete cascade on update
        restrict,
    Name varchar(256) not null,
    Value varchar(2048) not null,
    primary key (ExchangeId, Name));

create table NormalizedMessage(
    ExchangeId bigint not null constraint fk_NormalizedMessage
        references Exchange on delete cascade on update
        restrict,
    Type varchar(5) not null,
    Content clob(2 g),
    primary key (ExchangeId, Type));

create table MessageProperty(
    ExchangeId bigint not null,
    Type varchar(5) not null,
    Name varchar(256) not null,
    Value varchar(256) not null,
    primary key (ExchangeId, Type, Name),
    foreign key (ExchangeId, Type) references NormalizedMessage
        on delete cascade on update restrict);

create table Attachment(
    MessageId bigint not null,
    Type varchar(5) not null,
    Name varchar(256) not null,
    ContentType varchar(10) not null,
    primary key (MessageId, Type, Name),
    foreign key (MessageId, Type) references NormalizedMessage
        on delete cascade on update restrict);

create table ByteContent(
    MessageId bigint not null,

```

```
Type varchar(5) not null,  
Name varchar(256) not null,  
Content blob(2 g),  
primary key (MessageId, Type, Name),  
foreign key (MessageId, Type, Name) references Attachment  
on delete cascade on update restrict);  
  
create table StringContent(  
MessageId bigint not null,  
Type varchar(5) not null,  
Name varchar(256) not null,  
Content clob(2 g),  
primary key (MessageId, Type, Name),  
foreign key (MessageId, Type, Name) references Attachment  
on delete cascade on update restrict);
```