IONA®

Celtix Enterprise

Getting Started with Celtix Enterprise

Version 1.0
December 2006

Making Software Work Together™

Getting Started with Celtix Enterprise

IONA Technologies

Version 1.0

Published December 4, 2006

Table of Contents

List of Figures

List of Examples

Preface

What is Covered in This Book

This book introduces you to some of the major components of Celtix Enterprise. In particular, for each of the container types provided by Celtix, this guide explains the basic architecture of the container and describes how to deploy a simple demonstration to the container.

Who Should Read This Book

This book is aimed at developers and deployers who want to gain a rapid overview of the major components of Celtix Enterprise.

The Celtix Enterprise Library

The Celtix Enterprise documentation library consists of the following books:

- Installing the Celtix Enterprise Binary Distribution describes the prerequisites for installing Celtix Enterprise and the procedures for installing Celtix Enterprise from the binary distribution. This is the preferred installation procedure for most users.
- Installing Celtix Enterprise from the Source Distribution describes the prerequisites for installing Celtix Enterprise and the procedures for installing Celtix Enterprise from the source distribution. This is only recommended for advanced users.
- An Introduction to Celtix Enterprise describes the components that make up Celtix Enterprise and how the work together. It also describes many of the concepts and techniques involved in SOA.
- Getting Started with Celtix Enterprise describes how to get up and running using Celtix Enterprise using a detailed example of creating and deploying a service.
- Using Celtix Enterprise provides detailed information on using Celtix Enterprise to develop and deploy Java services.
- Celtix Enterprise Command Reference is a quick reference to the commands you need when developing and deploying services with Celtix Enterprise.

The Celtix Enterprise GUI tools also include on-line help. To access it select Help → Help Contents. The help for the Celtix Enterprise GUI tools is in the section entitled SOA Tools Platform Developer Guide.

In addition to the above books, you may also want to read the documentation for each of the components that Celtix Enterprise bundles. This documentation is available from the projects responsible for developing the component.

Getting the Latest Version

The latest updates to the Celtix Enterprise documentation can be found at http://www.iona.com/support/docs.

Compare the version dates on the web page for your product version with the date printed on the copyright page of the PDF edition of the book you are reading.

Searching the Celtix Enterprise Library

You can search the online documentation by using the Search box at the top right of the documentation home page:

http://www.iona.com/support/docs

To search a particular library version, browse to the required index page, and use the Search box at the top right, for example:

http://www.iona.com/support/docs/celtix/1.0

You can also search within the PDF versions of each book. To search within a PDF version of a book, in Adobe Acrobat, select Edit → Find, and enter your search text.

Additional IONA Resources

The IONA Knowledge Base [http://www.iona.com/support/knowledge_base/index.xml] (`http://www.iona.com/support/knowledge_base/index.xml`) contains helpful articles written by IONA experts about Inferno and other products.

The IONA Update Center [http://www.iona.com/support/updates/index.xml] (`http://www.iona.com/support/updates/index.xml`) contains the latest releases and patches for IONA products.

If you need help with this or any other IONA product, go to IONA Online Support [http://www.iona.com/support/index.xml] (`http://www.iona.com/support/index.xml`).

Comments, corrections, and suggestions on IONA documentation can be sent to `<docs-support@iona.com>`.

Open Source Project Resources

Apache Incubator CXF

Web site: http://incubator.apache.org/cxf/

User's list: `<cxf-user@incubator.apache.org>`

Apache Incubator Qpid

Web site: http://incubator.apache.org/qpid/

User's list: `<qpid-user@incubator.apache.org>`

Apache Tomcat

Web site: http://tomcat.apache.org/

User's list: `<users@tomcat.apache.org>`

ActiveMQ

Web site: http://www.activemq.org/site/home.html

User's list: `<activemq-users@geronimo.apache.org>`

Apache Incubator ServiceMix

Web site: http://servicemix.org/site/home.html

User's list: `<servicemix-users@geronimo.apache.org>`

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

`fixed width`	Fixed width (Courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `IT_Bus::AnyType` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
`Fixed width italic`	Fixed width italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`YourUserName`*
Italic	Italic words in normal text represent *emphasis* and introduce *new terms*.
Bold	Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the **User Preferences** dialog.

Keying conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
`%`	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
`#`	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
`>`	The notation > represents the MS-DOS or Windows command prompt.
`...`	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
`[ ]`	Brackets enclose optional items in format and syntax descriptions.
`{ }`	Braces enclose a list from which you must choose an item in format and syntax descriptions.
`|`	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in `{}` (braces).

Admonition conventions

This book uses the following conventions for admonitions:

🗎	Notes display information that may be useful, but not critical.
🔔	Tips provide hints about completing a task or using a tool. They may also provide information about workarounds to possible problems.
❗	Important notes display information that is critical to the task at hand.
▼	Cautions display information about likely errors that can be encountered. These errors are unlikely to cause damage to your data or your systems.
✖	Warnings display information about errors that may cause damage to your systems. Possible damage from these errors include system failures and loss of data.

Chapter 1. Basic Celtix Demonstrations

This chapter explains how to set up your environment and gives a quick overview of the getting started demonstrations. It is recommended that you read this chapter before proceeding to build and run the demonstrations.

Before You Start

Before you start, you need to ensure that your environment variables and path are correctly set up to access the requisite components of Celtix Enterprise.

Prerequisites

To run the demonstrations described in this document, you need to have the following product installed:

- *Java Platform, Standard Edition 5.0 (that is, JDK 1.5.x)*—you can download the latest JDK from Java SE Downloads [http://java.sun.com/javase/downloads/index.jsp].

Windows environment

Example 1.1, "Windows Environment for Celtix" shows an example of a script that sets the environment variables you need to run the demonstrations on Windows.

Example 1.1. Windows Environment for Celtix

```
@echo off
REM Celtix Enterprise Environment

set CELTIX_HOME=CeltixInstallDir
set JAVA_HOME=JDKInstallDir
set ANT_HOME=%CELTIX_HOME%\tools\ant

set CLASSPATH=.;%CELTIX_HOME%\lib\cxf-incubator.jar;.\build\classes

call "%CELTIX_HOME%\bin\celtix_env.bat"

echo Set Celtix Enterprise Environment Variables
```

Where `CeltixInstallDir` is the directory where you installed Celtix Enterprise, and `JDKInstallDir` is the directory where you installed Sun's Java Platform, Standard Edition.

UNIX environment

Example 1.2, “UNIX Environment for Celtix” shows an example of a script that sets the environment variables you need to run the demonstrations on UNIX.

Example 1.2. UNIX Environment for Celtix

```
#!/bin/sh
# Celtix Enterprise Environment

CELTIX_HOME=CeltixInstallDir
export CELTIX_HOME

JAVA_HOME=JDKInstallDir
export JAVA_HOME

ANT_HOME=$CELTIX_HOME/tools/ant
export ANT_HOME

CLASSPATH=.:$CELTIX_HOME/lib/cxf-incubator.jar:./build/classes
export CLASSPATH

. $CELTIX_HOME/bin/celtix_env.sh

echo Set Celtix Enterprise Environment Variables
```

Where *`CeltixInstallDir`* is the directory where you installed Celtix Enterprise, and *`JDKInstallDir`* is the directory where you installed Sun’s Java Platform, Standard Edition.

Getting Started Demonstrations

To get started with Celtix, you need to gain familiarity with the various components that make up Celtix Enterprise. The getting started demonstrations have been selected to give you a quick tour of the features and components available.

Standalone demonstration

The standalone demonstration describes how to deploy a service in standalone mode, using the Celtix ASE to instantiate and activate the service.

See Chapter 2, *Standalone Service* .

Container demonstrations

The following container demonstrations are described in this guide:

- the section called “Servlet container demonstration” .
- the section called “JBI container demonstration” .

Servlet container demonstration

The servlet container demonstration describes how to deploy a service into a servlet container, such as Apache Tomcat.

See Chapter 3, *Service in a Servlet Container* .

JBI container demonstration

The JBI container demonstration describes how to deploy a service into a *Java Business Integration (JBI)* container, such as Apache Incubator ServiceMix.

See Chapter 4, *Service in a JBI Container* .

Greeter Service WSDL

The demonstrations described in this document are based on the Greeter service. For reference, this section provides a listing of the Greeter service’s WSDL contract, which consists of a logical part (the Greeter port type), a SOAP binding, and a HTTP port.

Logical part of Greeter WSDL

Example 1.3, “Greeter Port Type and Associated Schema” shows the logical part of the Greeter WSDL. The Greeter interface is common to most of the demonstrations discussed in this document.

Example 1.3. Greeter Port Type and Associated Schema

```
<wsdl:definitions name="HelloWorld"
targetNamespace="http://apache.org/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://apache.org/hello_world_soap_http"
    xmlns:x1="http://apache.org/hello_world_soap_http/types"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:types>
        <schema targetNamespace="http://apache.org/hello_world_soap_http/types"
            xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://apache.org/hello_world_soap_http/types"
            elementFormDefault="qualified">
        <simpleType name="MyStringType">
        <restriction base="string">
            <maxLength value="30" />
        </restriction>
        </simpleType>

            <element name="sayHi">
                <complexType/>
            </element>
            <element name="sayHiResponse">
                <complexType>
                    <sequence>
                        <element name="responseType" type="string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMe">
                <complexType>
                    <sequence>
                        <element name="requestType" type="tns:MyStringType"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMeResponse">
                <complexType>
                    <sequence>
                        <element name="responseType" type="string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMeOneWay">
                <complexType>
                    <sequence>
                        <element name="requestType" type="string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="pingMe">
                <complexType/>
            </element>
            <element name="pingMeResponse">
                <complexType/>
            </element>
            <element name="faultDetail">
```

```
                <complexType>
                    <sequence>
                        <element name="minor" type="short"/>
                        <element name="major" type="short"/>
                    </sequence>
                </complexType>
            </element>
        </schema>
    </wsdl:types>
    <wsdl:message name="sayHiRequest">
        <wsdl:part element="x1:sayHi" name="in"/>
    </wsdl:message>
    <wsdl:message name="sayHiResponse">
        <wsdl:part element="x1:sayHiResponse" name="out"/>
    </wsdl:message>
    <wsdl:message name="greetMeRequest">
        <wsdl:part element="x1:greetMe" name="in"/>
    </wsdl:message>
    <wsdl:message name="greetMeResponse">
        <wsdl:part element="x1:greetMeResponse" name="out"/>
    </wsdl:message>
    <wsdl:message name="greetMeOneWayRequest">
        <wsdl:part element="x1:greetMeOneWay" name="in"/>
    </wsdl:message>
    <wsdl:message name="pingMeRequest">
        <wsdl:part name="in" element="x1:pingMe"/>
    </wsdl:message>
    <wsdl:message name="pingMeResponse">
        <wsdl:part name="out" element="x1:pingMeResponse"/>
    </wsdl:message>
    <wsdl:message name="pingMeFault">
        <wsdl:part name="faultDetail" element="x1:faultDetail"/>
    </wsdl:message>

    <wsdl:portType name="Greeter">
        <wsdl:operation name="sayHi">
            <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
            <wsdl:output message="tns:sayHiResponse" name="sayHiResponse"/>
        </wsdl:operation>

        <wsdl:operation name="greetMe">
            <wsdl:input message="tns:greetMeRequest" name="greetMeRequest"/>
            <wsdl:output message="tns:greetMeResponse" name="greetMeResponse"/>
        </wsdl:operation>

        <wsdl:operation name="greetMeOneWay">
          <wsdl:input message="tns:greetMeOneWayRequest" name="greetMeOneWayRequest"/>

        </wsdl:operation>
        <wsdl:operation name="pingMe">
            <wsdl:input name="pingMeRequest" message="tns:pingMeRequest"/>
            <wsdl:output name="pingMeResponse" message="tns:pingMeResponse"/>
            <wsdl:fault name="pingMeFault" message="tns:pingMeFault"/>
        </wsdl:operation>
    </wsdl:portType>
  ...
</wsdl:definitions>
```

SOAP binding for Greeter

Example 1.4, “SOAP Binding for the Greeter Interface” shows the SOAP binding, `Greeter_SOAPBinding`, for the Greeter interface.

Example 1.4. SOAP Binding for the Greeter Interface

```
<wsdl:definitions name="HelloWorld"
targetNamespace="http://apache.org/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://apache.org/hello_world_soap_http"
    xmlns:x1="http://apache.org/hello_world_soap_http/types"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:binding name="Greeter_SOAPBinding" type="tns:Greeter">
    ...
       <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>


        <wsdl:operation name="sayHi">
            <soap:operation soapAction="" style="document"/>
            <wsdl:input name="sayHiRequest">
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="sayHiResponse">
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>

        <wsdl:operation name="greetMe">
            <soap:operation soapAction="" style="document"/>
            <wsdl:input name="greetMeRequest">
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="greetMeResponse">
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>

        <wsdl:operation name="greetMeOneWay">
            <soap:operation soapAction="" style="document"/>
            <wsdl:input name="greetMeOneWayRequest">
                <soap:body use="literal"/>
            </wsdl:input>
        </wsdl:operation>

        <wsdl:operation name="pingMe">
            <soap:operation style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
            <wsdl:fault name="pingMeFault">
                <soap:fault name="pingMeFault" use="literal"/>
            </wsdl:fault>
        </wsdl:operation>

    </wsdl:binding>
    ...
</wsdl:definitions>
```

HTTP port for Greeter

Example 1.5, “HTTP Port for the Greeter Interface” shows the definition of a HTTP port for the Greeter interface with a SOAP binding.

Example 1.5. HTTP Port for the Greeter Interface

```
<wsdl:definitions name="HelloWorld"
targetNamespace="http://apache.org/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://apache.org/hello_world_soap_http"
    xmlns:x1="http://apache.org/hello_world_soap_http/types"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    ...
    <wsdl:service name="SOAPService">
        <wsdl:port binding="tns:Greeter_SOAPBinding" name="SoapPort">
            <soap:address location="http://localhost:9000/SoapContext/SoapPort"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Greeter Service Implementation

This section shows a sample implementation of the `Greeter` interface. Due to the portability of the Celtix ASE programming API, this same code can be deployed into a wide variety of containers, including a servlet container, a JBI container, and a Spring container.

GreeterImpl class

The `GreeterImpl` class shown in Example 1.6, “The GreeterImpl Class” provides the implementation of the `Greeter` interface.

Example 1.6. The GreeterImpl Class

```
package demo.hw.server;

import java.util.logging.Logger;
import org.apache.hello_world_soap_http.Greeter;
import org.apache.hello_world_soap_http.PingMeFault;
import org.apache.hello_world_soap_http.types.FaultDetail;

1@javax.jws.WebService(portName = "SoapPort", serviceName = "SOAPService",
  targetNamespace = "http://apache.org/hello_world_soap_http",
  endpointInterface = "org.apache.hello_world_soap_http.Greeter")

2public class GreeterImpl implements Greeter {

    private static final Logger LOG =
        Logger.getLogger(GreeterImpl.class.getPackage().getName());

3    public String greetMe(String me) {
        LOG.info("Executing operation greetMe");
        System.out.println("Executing operation greetMe");
        System.out.println("Message received: " + me + "\n");
        return "Hello " + me;
    }

    public void greetMeOneWay(String me) {
        LOG.info("Executing operation greetMeOneWay");
        System.out.println("Executing operation greetMeOneWay\n");
        System.out.println("Hello there " + me);
    }
    public String sayHi() {
        LOG.info("Executing operation sayHi");
        System.out.println("Executing operation sayHi\n");
        return "Bonjour";
    }

    public void pingMe() throws PingMeFault {
        FaultDetail faultDetail = new FaultDetail();
        faultDetail.setMajor((short)2);
        faultDetail.setMinor((short)1);
        LOG.info("Executing operation pingMe, throwing PingMeFault
exception");
        System.out.println("Executing operation pingMe, throwing PingMeFault
exception\n");
        throw new PingMeFault("PingMeFault raised by server", faultDetail);
    }
}
```

The `GreeterImpl` class follows the JAX-WS standard to implement the `Greeter` port type. In outline, the preceding code example can be explained as follows:

1. The `@jaxws` annotation preserves information from the original WSDL contract; specifically recording the WSDL port type, service name, and port name associated with The `GreeterImpl` class.

2. The `GreeterImpl` class provides an implementation of the `org.apache.hello_world_soap_http.Greeter` base class, which is generated automatically by the `wsdl2java` tool.

3. Each of the operations from the `Greeter` interface—`greetMe`, `greetMeOneWay`, `sayHi`, and `pingMe`—are implemented by the `GreeterImpl` class. These operation impementations can be called by Web service clients once the `GreeterImpl` class has been deployed into a suitable container.

Chapter 2. Standalone Service

This chapter describes how to build and run a demonstration that illustrates a Web service running in standalone mode—that is, without needing to be deployed into a container.

Standalone Demonstration

The standalone demonstration is a simple client/server application, where operation invocations are transmitted over the SOAP/HTTP protocol.

Demonstration location

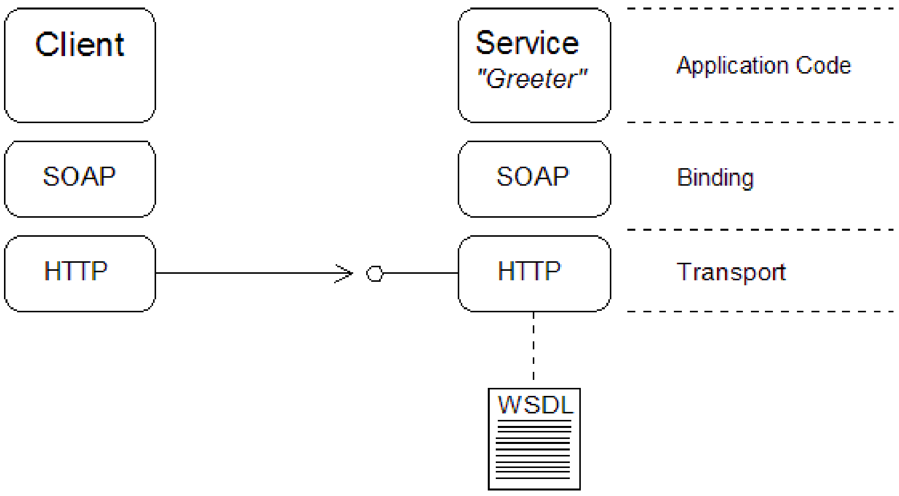
The CXF standalone demonstration is located in the following directory:

```
CeltixInstallDir/samples/service_creation/hello_world
```

Demonstration overview

Figure 2.1, “Overview of the Standalone Demonstration” shows the main components of the standalone demonstration. This is a straightforward client/server Web service application.

Figure 2.1. Overview of the Standalone Demonstration



Standalone server

The server program is a Web service that accepts requests through the SOAP/HTTP protocol. The server's WSDL contract is defined in the section called "Greeter Service WSDL" . Because the server is implemented in standalone mode, you need to program Celtix ASE to create an instance of the service explicitly, as discussed in the section called "Main Function for a Standalone Service" .

Client

The client program is a standalone Web client, implemented using Celtix ASE. The client implementation invokes each of the Greeter interface's operations in sequence: `sayHi`, `greetMe`, `greetMeOneWay`, and `pingMe`.

SOAP binding

The SOAP binding for the Greeter service is defined in the WSDL contract, as shown in the section called "SOAP binding for Greeter" .

HTTP transport

The HTTP transport for the Greeter service is defined in the WSDL contract, as shown in the section called "HTTP port for Greeter" .

Main Function for a Standalone Service

The Celtix ASE product is capable of running Web services in a standalone mode. In order to run a Celtix ASE standalone service, you must provide just a few lines of code to define a `main()` function for the server program.

Server main() function

Example 2.1, "Server main() Function for the Greeter Service" shows the code for a simple Celtix ASE `main()` function. This is all the code that is required to launch the Greeter service as a standalone service.

Example 2.1. Server main() Function for the Greeter Service

```
package demo.hw.server;

import javax.xml.ws.Endpoint;
```

```
public class Server {

    protected Server() throws Exception {
        System.out.println("Starting Server");

        Object implementor = new GreeterImpl();
        String address = "http://localhost:9000/SoapContext/SoapPort";
        Endpoint.publish(address, implementor);
    }

    public static void main(String args[]) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

The `Server.main()` function creates a new `Server` instance and then goes to sleep for five minutes, giving the background thread a chance to process incoming requests.

The `Server()` constructor is responsible for launching the Greeter service. It creates a `GreeterImpl` instance and then starts the service by calling the `Endpoint.publish()` method. The `publish()` method kicks off a background thread to process incoming requests to the Greeter service and starts listening on the IP port specified by `address`.

Build and Run the Standalone Demonstration

Follow the instructions in this section to build and run the CXF standalone demonstration. These instructions assume that you have already configured your environment as described in the section called “Before You Start” .

Build the demonstration

Build the standalone demonstration as follows:

1. Open a command prompt and change directory to *`CeltixInstallDir`*`/samples/service_creation/hello_world`.

2. Enter the following command to build the demonstration:

   ```
   ant build
   ```

The command produces output similar to the following:

```
Buildfile: build.xml

maybe.generate.code:

generate.code:
     [echo] Generating code using wsdl2java...
    [mkdir] Created dir:
C:\Programs\Celtix\samples\service_creation\hello_world\build\classes
    [mkdir] Created dir:
C:\Programs\Celtix\samples\service_creation\hello_world\build\src
     [java] wsdl2java -verbose -d
C:\Programs\Celtix\samples\service_creation\hello_world\build\
src C:\Programs\Celtix\samples\service_creation\hello_world\wsdl/hello_world.wsdl
     [java] wsdl2java - 2.0-M1-IONA-SNAPSHOT

    [touch] Creating
C:\Programs\Celtix\samples\service_creation\hello_world\build\src\.CODEGEN_
DONE

compile:
    [javac] Compiling 16 source files to
C:\Programs\Celtix\samples\service_creation\hello_world
\build\classes

build:

BUILD SUCCESSFUL
Total time: 6 seconds
```

Building the demonstration consists essentially of two steps:

a. Generating stub code from the `hello_world.wsdl` file, using the `wsdl2java` utility.

b. Compiling the application code and the WSDL stub code using the `javac` compiler.

Run the server

Run the standalone server as follows:

Enter the following command from within the `samples/service_creation/hello_world` directory in order to run the server:

```
ant server
```

The command produces output similar to the following:

```
Buildfile: build.xml

maybe.generate.code:

compile:

build:

server:
     [java] Starting Server
     [java] Server ready...
```

Run the client

Run the client as follows:

1. Open a new command prompt and change directory to *`CeltixInstallDir`*`/samples/service_creation/hello_world`.

2. Enter the following command to run the client:

   ```
   ant client
   ```

   The command produces output similar to the following:

   ```
   Buildfile: build.xml

   maybe.generate.code:

   compile:

   build:

   client:
        [java]
   file:/C:/Programs/Celtix/samples/service_creation/hello_world/wsdl/hello_world.wsdl

        [java] Invoking sayHi...
        [java] Server responded with: Bonjour

        [java] Invoking greetMe...
        [java] Server responded with: Hello fbolton

        [java] Invoking greetMe with invalid length string, expecting
   exception...
        [java] Invoking greetMeOneWay...
        [java] No response from server as method is OneWay
   ```

```
     [java] Invoking pingMe, expecting exception...
     [java] Expected exception: PingMeFault has occurred: PingMeFault
raised by
server
     [java] FaultDetail major:2
     [java] FaultDetail minor:1

BUILD SUCCESSFUL
Total time: 3 seconds
```

Chapter 3. Service in a Servlet Container

This chapter describes how to build and run a demonstration that illustrates a Web service running in a servlet container.

Servlet Container Demonstration

The servlet container demonstration shows how to deploy a service into a standard Web server, using a CXF servlet that acts as an adapter for Web services.

Demonstration location

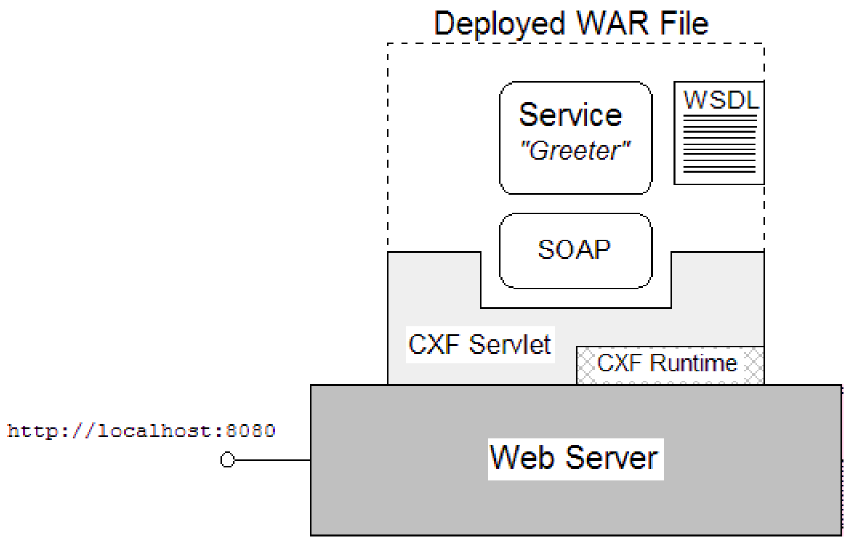
The servlet container demonstration is located in the following directory:

```
CeltixInstallDir/samples/service_creation/hello_world
```

Demonstration overview

Figure 3.1, “Overview of the Servlet Container Demonstration” shows the main components of the servlet container demonstration.

Figure 3.1. Overview of the Servlet Container Demonstration



Web server

The Web server shown in Figure 3.1, “Overview of the Servlet Container Demonstration” can be any Web server that is capable of acting as a servlet container—for example, Apache Tomcat. When a Web server is used as the container, all of the hosted services are accessed through the same IP port. For example, the default IP port for Tomcat is `8080`, which gives a base URL of `http://`*`Hostname`*`:8080`.

Deployed WAR file

The Greeter service is deployed to the Web server as a Web archive (WAR) file. In addition to configuration files, the WAR file contains the compiled code for the Greeter service, the WSDL stub code, and a copy of the WSDL contract. For more details about the WAR file, see the section called “Deploying to a Servlet Container” .

CXF servlet

The CXF servlet is a standard servlet provided by Celtix ASE that acts as an adapter for Web service endpoints. Each instance of a CXF servlet can host single or multiple service endpoints (see the section called “cxf-servlet.xml file”). The CXF servlet is part of the Celtix ASE runtime and is implemented by the `org.apache.cxf.jaxws.servlet.CXFServlet` class.

The `CXFServlet` class is referenced, but not included in the WAR file. There is no need to include it, because the Web server can extract the `CXFServlet` class from the Celtix ASE runtime.

Celtix ASE runtime

The Celtix ASE runtime Jars must be accessible to the deployed service. Normally, you would need to perform specific steps to install the Celtix ASE runtime in the Web server. In the case of the Web server bundled with Celtix, however, these steps are performed automatically by the Celtix installer at install time.

Greeter service

The application code for the Greeter service is identical to the case of a standalone service—see the section called “Greeter Service Implementation” . The code is not affected by being deployed into a servlet container.

Although the SOAP binding continues to be used to encode messages, the HTTP port specified in the original WSDL contract is irrelevant in this scenario. The HTTP protocol layer is now implemented by the Web server, not the Celtix ASE runtime.

WSDL contract

The original WSDL contract for the service is included in the WAR file. This static copy of the WSDL contract specifies the binding for the Greeter service. The port address in the WSDL contract is ignored, however.

When the Greeter service is initialized by the servlet container, Celtix automatically updates the in-memory copy of the WSDL contract with the correct endpoint address. It is this *updated* copy of the contract that is sent to clients that query the WSDL contract.

Deploying to a Servlet Container

In order to deploy a service to a servlet container, it is necessary to package all of the relevant files into a Web archive (WAR) file. The WAR file is a standard format for packaging Web applications.

.war file

The `helloworld.war` contains the following files and directories:

```
META-INF/
        Manifest.mf
WEB-INF/
        wsdl/
                hello_world.wsdl
        classes/
                *.class
        web.xml
        cxf-servlet.xml
```

WSDL contract

The WSDL contract, `hello_world.wsdl`, is included in the Web archive. The contract specifies a SOAP binding and a HTTP port, but the address in the HTTP port is ignored. A URL constructed from the servlet configuration is used instead of the address in the contract—see the section called “URL for Greeter service” .

Class files

The Web archive includes the class files for the server implementation and the WSDL stub code under the `WEB-INF/classes` directory.

web.xml file

The `web.xml` file instructs Tomcat to load the `org.apache.cxf.jaxws.servlet.CXFServlet` class. This servlet plays the role of a service adapter, enabling you to deploy the Greeter service into the Tomcat Web server.

You do not normally need to edit the contents of the `web.xml` file. For reference, a copy of the standard `web.xml` file is stored in the *`CeltixInstallDir`*`/etc` directory.

cxf-servlet.xml file

The `cxf-servlet.xml` file is used to configure the endpoints that plug in to the CXF servlet. When the CXF servlet starts up, it reads the list of `endpoint` elements in this file and initializes a service endpoint for each one.

In the current example, the `cxf-servlet.xml` file contains just a single `endpoint` element to configure the Greeter service endpoint, as follows:

```
<endpoints>

    <endpoint
        name="hello_world"
        interface="org.apache.hello_world_soap_http.Greeter"
        implementation="demo.hw.server.GreeterImpl"
        wsdl="WEB-INF/wsdl/hello_world.wsdl"
        service="{http://apache.org/hello_world_soap_http}SOAPService"
        port="{http://apache.org/hello_world_soap_http}SOAPPort"
        url-pattern="/hello_world" />

</endpoints>
```

URL for Greeter service

When you deploy the Greeter service into a servlet container, the original address specified in the WSDL contract is ignored and a specially constructed servlet URL is used instead. The constructed URL has the following general form:

```
http://Hostname:Port/Context/CXFServletPat/EndpointPat
```

Where *`Hostname`* and *`Port`* are the host name and IP port where the Web server listens for incoming HTTP messages (typically, you can use `localhost` and `8080` for these values). The servlet *`Context`* is normally equal to the name of the `.war` file. For example, the `helloworld.war` file has a context equal to `helloworld`. The *`CXFServletPat`* pattern is specified by the `url-pattern` element in the `web.xml` file—by default, `services`. The *`EndpointPat`* is determined by the `url-pattern` attribute in the `cxf-servlet.xml` file—by default, `hello_world`.

Using the typical values and defaults, you get the following URL:

```
http://localhost:8080/helloworld/services/hello_world
```

WSDL query URL

Associated with each service endpoint is a *query URL* that is used to download the service's WSDL contract. To obtain the query URL, simply append `?wsdl` to the endpoint URL.

For example, the default query URL for the Greeter service is as follows:

```
http://localhost:8080/helloworld/services/hello_world?wsdl
```

Clients can use the query URL to download an up-to-date copy of a service's WSDL contract. Downloading the WSDL contract is typically necessary, if the server makes dynamic changes to the WSDL contract.

Build and Run the Servlet Container Demonstration

Follow the instructions in this section to build and run the servlet container demonstration. These instructions assume that you have already configured your environment as described in the section called “Before You Start” .

Build the demonstration

Build the servlet container demonstration as follows:

1. Open a command prompt and change directory to `*CeltixInstallDir*/samples/service_creation/hello_world`.

2. Enter the following command to build the `.war` file for the servlet container:

   ```
   ant war
   ```

   The command produces output similar to the following:

   ```
   Buildfile: build.xml

   maybe.generate.code:

   compile:

   build:

   war:
       [mkdir] Created dir:
   C:\Programs\Celtix\samples\service_creation\hello_world\build\war
         [war] Building war:
   C:\Programs\Celtix\samples\service_creation\hello_world\build\war\hell
   oworld.war

   BUILD SUCCESSFUL
   Total time: 2 seconds
   ```

   The result of running this command is a file, `helloworld.war`, which is stored in the `hello_world/build/war` subdirectory.

Deploy the .war file

To deploy the `.war` file, copy the `helloworld.war` file to the Tomcat `webapps` directory, as follows:

Windows:

```
> copy CeltixInstallDir\samples\service_creation\hello_world\build\war\helloworld.war
 CeltixInstallDir\containers\servlet\webapps
```

UNIX:

```
% cp  CeltixInstallDir/samples/service_creation/hello_world/build/war/helloworld.war
CeltixInstallDir/containers/servlet/webapps
```

Start the Web server

To start the Tomcat Web server, enter the following command:

Windows:

```
> tomcat_start
```

UNIX:

```
% tomcat_start.sh
```

As the Tomcat server starts up, it automatically loads and deploys the `helloworld.war` file from the `webapps` directory.

Run the client

To run the client using `ant`, you need to provide the *base URL* of the deployed servlet as a parameter. The base URL is simply an URL of the form `http://Hostname:Port` that accesses the Tomcat root page. For example, if the base URL is `http://localhost:8080`, you can run the client with the following `ant` command:

```
ant client-servlet -Dbase.url=http://localhost:8080
```

Alternatively, instead of using the `ant` command, you can run the client directly using the `java` command. Assuming that the `./build/classes` directory is on your CLASSPATH, you can change directory to the `samples/service_creation/hello_world` directory and enter the following command:

Windows:

```
> java -Djava.util.logging.config.file=%CELTIX_HOME%\etc\logging.properties
demo.hw.client.Client http://localhost:8080/helloworld/services/hello_world?wsdl
```

UNIX:

```
% java -Djava.util.logging.config.file=$CELTIX_HOME/etc/logging.properties
demo.hw.client.Client http://localhost:8080/helloworld/services/hello_world?wsdl
```

In this case, the parameter provided to the client is the WSDL query URL for the Greeter service, not just the base URL. This command produces output similar to the following:

```
http://localhost:8080/helloworld/services/hello_world?wsdl
Invoking sayHi...
Server responded with: Bonjour

Invoking greetMe...
Server responded with: Hello fbolton

Invoking greetMe with invalid length string, expecting exception...

Invoking greetMeOneWay...
No response from server as method is OneWay

Invoking pingMe, expecting exception...
Expected exception: PingMeFault has occurred: PingMeFault raised by
server
FaultDetail major:2
FaultDetail minor:1
```

Chapter 4. Service in a JBI Container

This chapter describes how to build and run a demonstration that illustrates a Web service running in a JBI container.

JBI Container Demonstration

The JBI container demonstration shows how to deploy a service into a JBI container. Celtix provides a dedicated JBI service engine that enables services to plug in to the JBI container.

Demonstration location

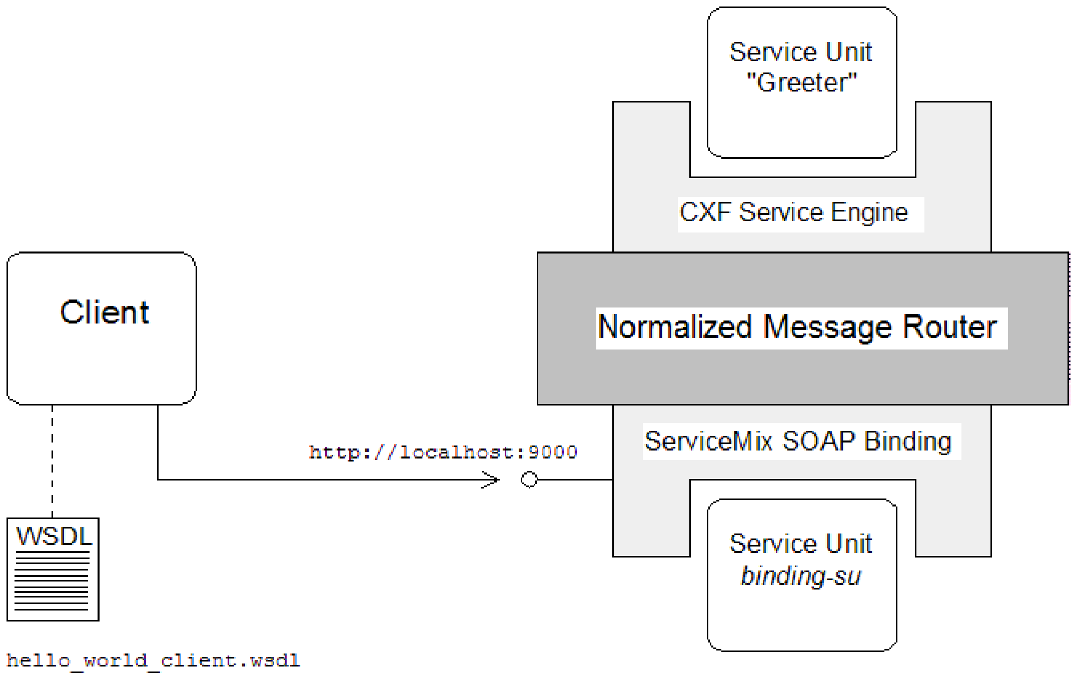
The JBI container demonstration is located in the following directory:

```
CeltixInstallDir/samples/service_creation/integration/JBI/internal_provider_external_consumer
```

Demonstration overview

Figure 4.1, “Overview of the JBI Container Demonstration” shows the main components of the JBI container demonstration.

Figure 4.1. Overview of the JBI Container Demonstration



Normalized message router

The *normalized message router (NMR)* is the core element of a JBI container. It is responsible for routing all messages between deployed JBI components. Messages sent through the router are always formatted as *normalized messages*.

The basic idea of an NMR is that all communication between deployed components occurs using normalized messages, which have an XML format closely modelled on the original WSDL message descriptions. Consequently, an NMR is a highly-optimized bus for the exchange of messages between Web services. For a service described in WSDL, the processing typically required to construct a normalized message is absolutely minimal.

CXF service engine

A *service engine (SE)* is a JBI component that enables you to deploy services (or *service providers* in JBI terminology) and client programs (or *service consumers* in JBI terminology).

The CXF service engine is an SE that has been specifically designed to facilitate the deployment of Celtix ASE services in the JBI container. The CXF service engine is available as part of the Celtix ASE runtime.

SOAP+HTTP binding

Communication with external Web service clients is enabled by the Apache Incubator ServiceMix SOAP+HTTP binding component. In contrast to the Celtix ASE standalone demonstration (see Chapter 2, *Standalone Service*), the JBI container demonstration does *not* use Celtix ASE's built-in SOAP binding and HTTP transport. Instead, request messages are received by the ServiceMix SOAP+HTTP binding component and then routed to the CXF service engine through the NMR. Reply messages follow the reverse route.

Service unit

A *service unit* is the basic functional unit of a user application. For example, a service unit can encapsulate a service provider or a service consumer. A service unit can also be used to encapsulate configuration details for a target component—for example, as shown in Figure 4.1, "Overview of the JBI Container Demonstration" , the `binding-su` service unit is used to configure the ServiceMix SOAP+HTTP binding.

WSDL contract for the client side

In this scenario, the client program has a WSDL contract that differs from the WSDL contract used on the server side. The client WSDL contract configures the client to use a SOAP binding and a HTTP transport. The server WSDL contract, on the other hand, configures the Greeter service to use a binding and transport for sending and receiving normalized messages—see the section called "xformat binding" for details.

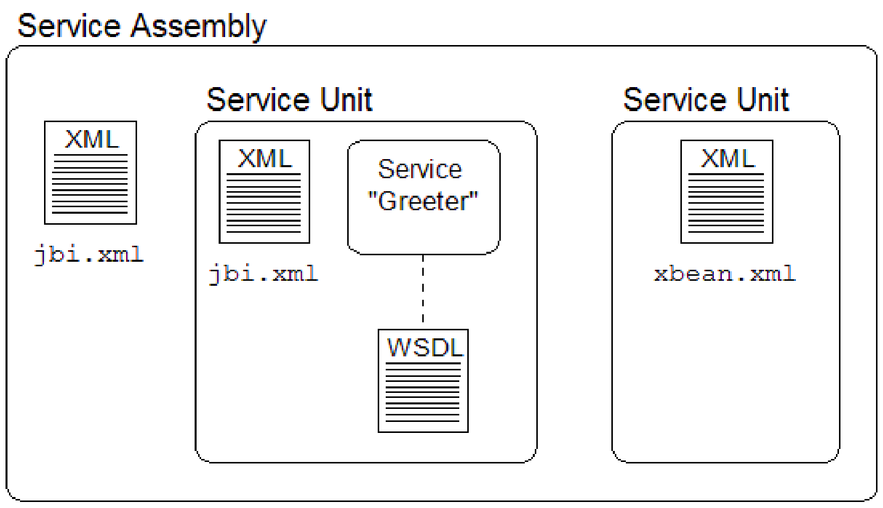
The JBI Service Assembly

In order to deploy a service to a JBI container, it is necessary to package all of the relevant files into a *service assembly* file. The service assembly is essentially an aggregation of one or more service units.

Service assembly archive

Figure 4.2, "Example of a JBI Service Assembly" shows an overview of the service assembly archive used for the current demonstration. The archive is packaged as a `.zip` file and consists of two service units and a deployment descriptor, `jbi.xml`.

Figure 4.2. Example of a JBI Service Assembly



Service assembly deployment descriptor

The service assembly deployment descriptor, `jbi.xml`, consists of a sequence of service unit descriptions. For each service unit, the descriptor specifies the constituent files and indicates which target JBI component the service should be deployed into. In the current scenario, two service units are provided, as follows:

- *Greeter service unit*—deployed into the CXF service engine, and
- `binding-su` *service unit*—deployed into the ServiceMix SOAP+HTTP binding component.

Service unit for the Greeter service

The service unit for the Greeter service contains the following parts:

- `jbi.xml` *deployment descriptor*—this deployment descriptor is consumed by the CXF service engine, which is responsible for instantiating and activating the Greeter service.
- *Greeter service implementation*—the class files that implement the Greeter service, including WSDL stub code.
- *WSDL contract*—the server-side copy of the contract is defined to use the `xformat` binding and the `jbi` transport, which are designed to receive and send messages in normalized message format.

xformat binding

The `xformat` binding is a special binding type that marshals request and reply messages in the normalized message format. Example 4.1, "xformat Binding Element for the Greeter Service" shows the `xformat` binding used for the Greeter service deployed in the CXF service engine.

Example 4.1. xformat Binding Element for the Greeter Service

```
<wsdl:definitions name="HelloWorld"
                  targetNamespace="http://apache.org/hello_world"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://apache.org/hello_world"
                  xmlns:x1="http://apache.org/hello_world/types"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xformat="http://cxf.apache.org/bindings/xformat"
                  xmlns:jbi="http://apache.org/transport/jbi">

...

<wsdl:binding name="Greeter_SOAPBinding" type="tns:Greeter">
    <xformat:binding />

    <wsdl:operation name="sayHi">
        <wsdl:input name="sayHiRequest" />
        <wsdl:output name="sayHiResponse" />
    </wsdl:operation>

    <wsdl:operation name="greetMe">
        <wsdl:input name="greetMeRequest" />
        <wsdl:output name="greetMeResponse" />
    </wsdl:operation>

    <wsdl:operation name="greetMeOneWay">
        <wsdl:input name="greetMeOneWayRequest" />
    </wsdl:operation>

    <wsdl:operation name="pingMe">
        <wsdl:input />
        <wsdl:output />
        <wsdl:fault name="pingMeFault" />
        </wsdl:operation>
</wsdl:binding>
```

jbi transport

The `jbi` transport is responsible for interfacing with the NMR, passing messages back and forth in normalized message format. Example 4.2, "jbi Endpoint for the Greeter Service" shows the definition of the `jbi` endpoint for the Greeter service.

Example 4.2. jbi Endpoint for the Greeter Service

```
<wsdl:service name="HelloWorldService">
    <wsdl:port binding="tns:Greeter_SOAPBinding" name="SoapPort">
        <jbi:address location="http://localhost:9000/SoapContext/SoapPort"/>
    </wsdl:port>
</wsdl:service>
```

Service unit for the SOAP+HTTP binding

The service unit for the SOAP+HTTP binding contains a single file, `xbean.xml`, which configures a SOAP/HTTP endpoint that exposes the Greeter service to external consumers.

xbean.xml file

Example 4.3, "Configuration of the SOAP+HTTP Binding Component" shows the contents of the `xbean.xml` file which configures the SOAP+HTTP binding to open a HTTP listening port and route incoming requests to the Greeter service.

Example 4.3. Configuration of the SOAP+HTTP Binding Component

```
<beans xmlns:http="http://servicemix.apache.org/http/1.0"
       xmlns:demo="urn:servicemix:soap-binding"
       xmlns:sns1="http://apache.org/hello_world">

    <http:endpoint service="sns1:HelloWorldService"
        endpoint="SoapPort"
        interfaceName="sns1:Greeter"
        role="consumer"
        locationURI="http://localhost:9000/"
        defaultMep="http://www.w3.org/2004/08/wsdl/in-out"
                       soapVersion="1.1"
                       soap="true"
                       />
</beans>
```

Build and Run the JBI Container Demonstration

Follow the instructions in this section to build and run the JBI container demonstration. These instructions assume that you have already configured your environment as described in the section called “Before You Start” .

Build the demonstration

Build the servlet container demonstration as follows:

1. Open a command prompt and change directory to *CeltixInstallDir*/samples/service_creation/integration/JBI/internal_provider_external_consumer.

2. Enter the following command at the command prompt:

   ```
   ant build
   ```

Start ServiceMix

Start and prepare the ServiceMix JBI container as follows:

1. Enter the following command to start Apache Incubator ServiceMix (in the directory `internal_provider_external_consumer`):

   Windows:

   ```
   servicemix_start
   ```

   UNIX:

   ```
   servicemix_start.sh
   ```

   ServiceMix should produce output similar to the following:

   ```
   servicemix.bat: Ignoring predefined value for SERVICEMIX_HOME
   Starting Apache ServiceMix ESB: 3.0-incubating

   Loading Apache ServiceMix from servicemix.xml on the CLASSPATH
   INFO  - ConnectorServerFactoryBean     - JMX connector available
   at: service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
   INFO  - JBIContainer                   - ServiceMix 3.0-incubating
   JBI Container
    (ServiceMix) is starting
   INFO  - JBIContainer                   - For help or more informations
   ```

```
please see: http://incubator.apache.org/servicemix/
INFO  - ComponentMBeanImpl             - Initializing component:
#SubscriptionManager#
INFO  - DeploymentService              - Restoring service assemblies
INFO  - JBIContainer                   - ServiceMix JBI Container
(ServiceMix) started
```

During start-up, ServiceMix creates the following subdirectories, under the *`CeltixInstallDir`*`/containers/jbi` directory:

- `data`
- `deploy`
- `install`

2. Open a new command prompt and change directory to *`CeltixInstallDir`*`/samples/service_creation/integration/JBI/internal_provider_external_consumer`.

3. Deploy the ServiceMix shared component by copying the relevant zip file to the ServiceMix `deploy` directory, as follows:

   Windows:

```
> copy
%CELTIX_HOME%\containers\jbi\components\servicemix-shared-3.0-incubating-installer.zip
 %CELTIX_HOME%\containers\jbi\deploy
```

   UNIX:

```
% cp
$CELTIX_HOME/containers/jbi/components/servicemix-shared-3.0-incubating-installer.zip
 $CELTIX_HOME/containers/jbi/deploy
```

   Where it is assumed that the ServiceMix container is running in the `internal_provider_external_consumer` directory. The running ServiceMix container should produce output similar to the following:

```
INFO  - AutoDeploymentService          - Directory: deploy: Archive
changed: processing servicemix-shared-3.0-incubating-installer.zip ...
INFO  - AutoDeploymentService          - Directory: deploy: Finished
installation of archive:  servicemix-shared-3.0-incubating-installer.zip
```

4. Deploy the ServiceMix SOAP/HTTP binding component by copying the relevant zip file to the ServiceMix `deploy` directory, as follows:

   Windows:

   ```
   > copy
   %CELTIX_HOME%\containers\jbi\components\servicemix-http-3.0-incubating-installer.zip
    %CELTIX_HOME%\containers\jbi\deploy
   ```

   UNIX:

   ```
   % cp
   $CELTIX_HOME/containers/jbi/components/servicemix-http-3.0-incubating-installer.zip
    $CELTIX_HOME/containers/jbi/deploy
   ```

   The running ServiceMix container should produce output similar to the following:

   ```
   INFO  - AutoDeploymentService          - Directory: deploy: Archive
   changed: processing servicemix-http-3.0-incubating-installer.zip ...
   INFO  - jetty                          - Logging to
   org.apache.servicemix.http.jetty.JCLLogger@6e56ae
   via org.apache.servicemix.http.jetty.JCLLogger
   INFO  - ComponentMBeanImpl             - Starting component: servicemix-http
   INFO  - ComponentMBeanImpl             - Initializing component:
   servicemix-http

   INFO  - AutoDeploymentService          - Directory: deploy: Finished
   installation of archive:  servicemix-http-3.0-incubating-installer.zip
   ```

5. Deploy the ServiceMix JMS binding component by copying the relevant zip file to the ServiceMix `deploy` directory, as follows:

   Windows:

   ```
   > copy
   %CELTIX_HOME%\containers\jbi\components\servicemix-jms-3.0-incubating-installer.zip
    %CELTIX_HOME%\containers\jbi\deploy
   ```

   UNIX:

   ```
   % cp
   $CELTIX_HOME/containers/jbi/components/servicemix-jms-3.0-incubating-installer.zip
   $CELTIX_HOME/containers/jbi/deploy
   ```

   The running ServiceMix container should produce output similar to the following:

   ```
   INFO  - AutoDeploymentService          - Directory: deploy: Archive
   changed: processing servicemix-jms-3.0-incubating-installer.zip ...
   ```

```
INFO  - ComponentMBeanImpl             - Starting component: servicemix-jms
INFO  - ComponentMBeanImpl             - Initializing component:
servicemix-jms
INFO  - AutoDeploymentService          - Directory: deploy: Finished
installation of archive:  servicemix-jms-3.0-incubating-installer.zip
```

6. Enter the following command to install and start the CXF Service Engine:

   Windows:

   ```
   > servicemix_install_ca
   %CELTIX_HOME%\samples\service_creation\integration\JBI\internal_provider_external_consumer\service-engine\build\lib\cxf-service-engine.jar
   ```

   UNIX:

   ```
   % servicemix_install_ca.sh
   $CELTIX_HOME/samples/service_creation/integration/JBI/internal_provider_external_consumer/service-engine/build/lib/cxf-service-engine.jar
   ```

Deploy and start the service assembly

Enter the following command to deploy and start the service assembly:

Windows:

```
> servicemix_deploy_sa
%CELTIX_HOME%\samples\service_creation\integration\JBI\internal_provider_external_consumer\service-assembly\build\lib\cxf-service-assembly.zip
```

UNIX:

```
% servicemix_deploy_sa.sh
$CELTIX_HOME/samples/service_creation/integration/JBI/internal_provider_external_consumer/service-assembly/build/lib/cxf-service-assembly.zip
```

Run the client

To run the client, enter the following command (from the directory `internal_provider_external_consumer`):

```
ant client
```

This command produces output similar to the following:

```
Buildfile: build.xml

client:

client:
```

```
     [java]
file:/C:/Programs/Celtix/samples/service_creation/integration/JBI/internal_provider_external_consumer/service-unit/./wsdl/hello_world_client.wsdl

     [java] Invoking sayHi...
     [java] Server responded with: Bonjour

     [java] Invoking greetMe...
     [java] Server responded with: Hello YourName


BUILD SUCCESSFUL
Total time: 6 seconds
```

Undeploy the service assembly

Enter the following command to undeploy the service assembly (from the directory `internal_provider_external_consumer`):

Windows:

```
> servicemix_undeploy_sa cxf-service-assembly.zip
```

UNIX:

```
% servicemix_undeploy_sa.sh cxf-service-assembly.zip
```