Getting Started with Celtix

Table of Contents

Overview

This article shows you how to get started with Celtix. This includes showing you how to set up your development environment and build and run a basic SOAP/HTTP client and server. For information on how to install Celtix, see *The Celtix Installation Guide*, available from the Celtix website, `http://forge.objectweb.org/projects/celtix/`.

This document was written for Celtix Milestone 3; as Celtix matures future versions may deviate from this the material covered in this document. The Celtix team will endeavor to keep this document as up-to-date as possible.

Setting up Your Environment

Celtix should always be run from an appropriately configured shell. To configure your shell you need to do the following:

1. Set your `CLASSPATH` to pick-up the correct version of the JDK and the correct versions of the Celtix JAR files.
2. Set your `PATH` to ensure you are using the correct Java compilers and the correct version of the Celtix tools.
3. If you are using Ant as a build system, then you will need to pick up the correct version of Ant as well.

Setting the _HOME Variables

Many open-source projects follow the useful convention of having an `_HOME` environment variable to describe the location of the project installation on the file system; when using Celtix you will want to set `JAVA_HOME`, `CELTIX_HOME` and `ANT_HOME` appropriately. The `CELTIX_HOME` variable should be set to the root of your Celtix installation.

Setting the PATH Variable

To ensure that you pick up the correct version of the Java compiler, add the directory `JAVA_HOME/bin` to the `PATH` environment variable. Running **`java -version`** at the prompt will verify that you are picking up the correct version of Java; for Celtix, you should be using JDK 1.5.0 or higher.

To pick up the Celtix tools add the directory `CELTIX_HOME/bin` to the `PATH` environment variable. This ensures that you will use the Celtix code generation tools like `wsdl2java` and `java2wsdl`.

Setting the CLASSPATH Variable

To ensure that you are using the correct version of the Celtix classes, you should add `CELTIX_HOME/lib/celtix.jar` and `CELTIX_HOME/etc` to the `CLASSPATH`.

If you wish to use Ant, then add `ANT_HOME/bin` to the `CLASSPATH`.

Using a Script to Set Up the Shell Environment

Rather than setting these variables for every shell, consider using a `setenvs` script to this for you. An example script, `setenvs.bat`, for use with Windows, is shown below in Codesnap 1.

```
@echo off

REM Ensure that the values for the following variables are
REM set correctly for your installation.

set CELTIX_HOME=c:\bin\celtix-ms3\celtix
set JAVA_HOME=c:\bin\jdk1.5.0
set ANT_HOME=c:\bin\apache-ant-1.6.2

REM You should not have to modify anything below this point.

echo.
echo Take note of the following important variables - are they correct for your
echo system? If not then edit this file and correct them!
echo.
echo CELTIX_HOME = %CELTIX_HOME%
echo JAVA_HOME   = %JAVA_HOME%
echo ANT_HOME    = %ANT_HOME%
echo.

set PATH=%CELTIX_HOME%\bin;%PATH%
set PATH=%JAVA_HOME%\bin;%PATH%
set PATH=%ANT_HOME%\bin;%PATH%

set CELTIX_JAR=%CELTIX_HOME%\lib\celtix.jar
if not exist %CELTIX_JAR% (
      REM Assume it's a source (rather than a binary) distribution of Celtix
      set CELTIX_JAR=%CELTIX_HOME%\build\lib\celtix.jar
)

set CLASSPATH=%CELTIX_JAR%;%CLASSPATH%

title Celtix Shell
```

Codesnap 1: `setenvs.bat`

Celtix Development Environment

Developing applications with Celtix code is no different from developing with any other Java library or API. You just need to set the `CLASSPATH` appropriately and begin coding. You can develop with your favorite editor, IDE (Integrated Development Environment) or build system. In this section, we recommend two open-source tools used extensively by developers of Celtix:

- Eclipse
- Ant

Using Celtix with Eclipse

Eclipse (available from `http://www.eclipse.org`) provides an excellent Java IDE for Celtix development. We recommend using Eclipse 3.1.1 or higher, as Celtix requires support for Java 1.5 language constructs that is not available in earlier versions of Eclipse.

Eclipse provides a way to store “User Libraries”; collections of JARs and classes that can be reused across projects. Create a user library for Celtix by navigating to the “User Libraries” dialog box in Eclipse.

Window → Preferences → Java → Build Path → User Libraries

Add the file `celtix.jar` to the user library. If you are using a binary distribution of Celtix, this can be found in `CELTIX_HOME/lib/celtix.jar`; if you are using a source distribution of Celtix, it can be found in `CELTIX_HOME/build/lib/celtix.jar`.

At the time of writing, Eclipse is unable to pickup the manifest classpath present in `celtix.jar`. As a result you will also have to explicitly add *all* the JAR files for JAX-WS to your user library. For a binary distribution, these files will reside under `CELTIX_HOME/lib/jaxws-ri/20051104/lib`. In a source distribution, they can be found under `CELTIX_HOME/tools/jaxws-ri/20051104/lib`.

After you have created a user library for Celtix, you can add it to the Java project build path and Eclipse will auto-compile your code.

Using Celtix with Ant

Many Java developers will be familiar with the Ant build system, downloaded from `http://ant.apache.org`. Ant build-files provide an effective build system for Celtix – if you wish to use Ant, then you may wish to use the build-file below in CodeSnap 2 as a starting-point template. The build file imports the `common_build.xml` file used by the Celtix samples. This build file offers a number of features:

- The variable `codegen.notrequired` is true if no XSD or WSDL files in the `wsdl.dir` directory have changed since the last run of `wsdl2java`. If you do not declare `wsdl.dir` as a property then the default value `./wsdl` is used.
- The `wsdl2java` task can be used to generate Java code.
- The `celtixrun` task can be used to run a Java class with appropriate `CLASSPATH` and JVM argument settings for use with Celtix.

```
<project default="build">
    <!-- Import generic celtix build.xml file -->
    <property environment="env"/>
    <import file="${env.CELTIX_HOME}/samples/common_build.xml"/>

    <target name="generate.code" unless="codegen.notrequired">
      <echo message="Generating code using wsdl2java..."/>
       <wsdl2java file="HelloWorld.wsdl"/>
      <touch file="${codegen.timestamp.file}"/>
    </target>

    <!-- Targets to run the client and server -->
    <target name="helloworld.Server" depends="build">
        <celtixrun classname="helloworld.Server"/>
    </target>

    <target name="helloworld.Client" depends="build">
        <celtixrun classname="helloworld.Client"/>
    </target>
</project>
```

CodeSnap 2: Sample `build.xml` file for use with Celtix.

Directory Structure of a Celtix Project

There are a number of useful conventions for laying out the directory structure of a Celtix project. While you do not have to follow these conventions, it is beneficial to do so. For the purposes of this getting started guide, we will assume that you this directory structure has been adhered to.

A project typically contains the following directories:

- `build/classes` contains compiled Java classes, including those generated by `wsdl2java`.
- `build/src` contains Java source code generated by `wsdl2java`.
- `src` contains Java source code.
- `wsdl` contains WSDL files.

Some other directories are also common:

- `cfg` (or `conf`) contains configuration information
- `etc` contains miscellaneous files.
- `lib` contains JAR files needed for compilation
- `log` contains log files generated at run-time.

The top level project directory contains:

- The Ant build file (`build.xml`).
- Eclipse `.classpath` and `.project` files.
- Any other project-related files.

Writing a SOAP/HTTP Client and Server with Celtix

This section shows how to write a client and server for a simple “Hello, World” program. The tradition of “Hello, World” dates back to 1978 when Kernigan and Ritchie used a “Hello, World” program as the first C program in their book *The C Programming Language*. Their original program simply printed **“Hello, World”** to the screen. The client-server equivalent provides a `sayHello()` method that a client can invoke remotely, sending a string and receiving a string response.

The “Hello, World” interface used here is defined in the WSDL file `HelloWorld.wsdl`. While there is a “Hello, World” demo in the Celtix distribution, it uses a slightly different WSDL contract than that used here. The version used for this demo corresponds to the Java interface shown in CodeSnap 3.

```
public interface HelloWorld {
    public String sayHello(String message);
}
```

CodeSnap 3: `HelloWorld` *interface.*

The full WSDL contract is shown in CodeSnap 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--WSDL file template-->
<!--(c) 2005, IONA Technologies, Inc.-->
<definitions name="HelloWorld.wsdl"
      targetNamespace="http://www.celtix.org/courseware/HelloWorld"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:tns="http://www.celtix.org/courseware/HelloWorld"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <schema targetNamespace="http://www.celtix.org/courseware/HelloWorld"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <element name="sayHello">
        <complexType>
          <sequence>
            <element maxOccurs="1" minOccurs="1" name="message"
                     nillable="true" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
      <element name="sayHelloResponse">
        <complexType>
          <sequence>
            <element maxOccurs="1" minOccurs="1" name="return"
                     nillable="true" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </types>
  <message name="sayHello">
    <part element="tns:sayHello" name="parameters"/>
  </message>
  <message name="sayHelloResponse">
    <part element="tns:sayHelloResponse" name="parameters"/>
  </message>
  <portType name="HelloWorld">
    <operation name="sayHello">
      <input message="tns:sayHello" name="sayHello"/>
      <output message="tns:sayHelloResponse" name="sayHelloResponse"/>
    </operation>
  </portType>
  <binding name="HelloWorld_DocLiteral_SOAPBinding" type="tns:HelloWorld">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sayHello">
      <soap:operation soapAction="" style="document"/>
      <input name="sayHello">
        <soap:body use="literal"/>
      </input>
      <output name="sayHelloResponse">
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="HelloWorldService">
    <port binding="tns:HelloWorld_DocLiteral_SOAPBinding" name="SOAPOverHTTP">
      <soap:address location="http://localhost:9090/helloworld"/>
    </port>
  </service>
</definitions>
```

CodeSnap 4: `HelloWorld.wsdl` *WSDL contract.*

Generating Java Code for a WSDL Contract

The `wsdl2java` command-line tool (available in `CELTIX_HOME/bin`) is used to generate Java support code form your WSDL contract. The most commonly used command line options are:

- `-keep` keeps the generated Java source code (it gets deleted by default).
- `-p namespace=pkg:` generated types is the WSDL namespace `namespace` will be placed in the Java package `pkg`.
- `-d build/classes` puts compiled code into the destination directory `build/classes`.
- `-s build/src` puts generated Java code into the source directory `build/src`.

For example, to generate Java code for the `HelloWorld.wsdl` contract, you might use:

```
wsdl2java -s src -d classes -keep ./wsdl/HelloWorld.wsdl
```

Some users prefer not to keep the generated source code. If you wish to only generate compiled Java classes then omit the flags **`-s src -keep`**.

When the destination package is not specified with the `-p` option (as in the above example), then the package is derived from the target namespace used in the WSDL document, using an algorithm defined in the JAX-WS specification. For example, the namespace used in the sample `HelloWorld.wsdl` contract is `http://www.celtix.org/courseware/Helloworld;` this is converted to a package name as follows:

- The leading `http://www` is stripped
- The order of the domain name is reversed, giving `org.celtix`
- The remaining components in the URL path are converted to lower case and appended using a `.` separator, giving: `org.celtix.courseware.helloworld`

Browsing the Generated Code

The `wsdl2java` tool generates many files. In this example you should only take interest in two:

- the service endpoint interface, `HelloWorld`, contains methods that correspond to the operations in the WSDL contract.
- the service class `HelloWorldService`, contains suitable constructors and methods that correspond to the ports defined in the WSDL contract.

Using your favorite editor or Java IDE, browse these classes and become familiar with their contents.

Implementing the Servant

To implement the servant, create a Java class that implements the service endpoint interface. In the case of the Hello World example this will be `org.celtix.courseware.helloworld.HelloWorld`. A commonly used convention suggests that you should name your implementation class with the same name as the service endpoint interface, suffixed with `Impl`. A sample implementation is shown below in CodeSnap 5.

```
package helloworld
import org.celtix.courseware.helloworld.HelloWorld;

public class HelloWorldImpl implements HelloWorld
{
      public String sayHello(String arg0)
      {
            System.out.println("sayHello(" + arg0 + ")");
            return "Hello right back at ya!";
      }
}
```

CodeSnap 5: Servant implementation class: `helloworld.HelloWorldImpl` .

Writing the Server Mainline

A server mainline typically does at least the following:

- Create the servant object; and
- Create and publish the servant object's endpoint using `Endpoint.publish()`.

A sample server mainline is shown below in CodeSnap 6.

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

void main(String[] args)
{
    Object helloWorldImpl = null;
    String address = "http://localhost:9090/helloworld";

    bus = Bus.init();
    helloWorldImpl = new HelloWorldImpl();
    Endpoint.publish(address, helloWorldImpl);
}
```

CodeSnap 6: Server mainline class: `helloworld.Server`.

Writing the Client Mainline

A client mainline typically performs the following tasks:

- Declare the service `QName` and the location of the WSDL file.
- Create the service, using a generated constructor.
- Use the service to create a proxy to the remote service implementation.
- Invoke on the service.

An example client is shown below in CodeSnap 7.

```
package helloworld;

import java.io.File;
import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import org.objectweb.celtix.Bus;
import org.objectweb.celtix.BusException;

public static void main(String[] args)
{
  QName serviceName = new QName("http://www.celtix.org/courseware/HelloWorld",
      "HelloWorldService");

  URL wsdlURL = null;
  String wsdlFileLocation = "./wsdl/HelloWorld.wsdl";
  try {
    wsdlURL = new File(wsdlFileLocation).toURL();
  }
  catch (MalformedURLException e) {
    System.out.println("Error creating a URL from file '" +
      wsdlFileLocation + "'; details: " + e);
  }

  HelloWorldService helloWorldService =
    new HelloWorldService(wsdlURL, serviceName);

  helloWorld helloWorld = helloWorldService.getSOAPOverHTTP();

  String response = helloWorld.sayHello("Hello!");
}
```

CodeSnap 7: Server mainline class: `helloworld.Client`

Running the Client and Server

Running Celtix applications directly from the command-line

To run the client and server you must have your `CLASSPATH` variable set to include `CELTIX_HOME/lib/celtix.jar` and `CELTIX_HOME/etc`.

The Celtix runtime uses the `java.util.logging` framework; you can configure Celtix logging levels by pointing the JVM to a `logging.properties` file by defining the JVM system variable `java.util.logging.config.file`. Celtix provides a default `logging.properties` file in the `etc` directory, so you can use:

```
-Djava.util.logging.config.file=%CELTIX_HOME%/etc/logging.properties
```

You can run your classes from the prompt as shown:

```
java -Djava.util.logging.config.file=... helloworld.Client
java -Djava.util.logging.config.file=... helloworld.Server
```

To avoid repetitive typing create a startup script for your client and server; examples for the Windows operation system are shown below in CodeSnap 8, and CodeSnap 9 below.

```
@echo off

setlocal

if "%CELTIX_HOME%" == "" (
      echo You must set environment variable CELTIX_HOME to run this script.
      goto :eof
)

set BASE_DIR=%~dp0

set CELTIX_JAR=%CELTIX_HOME%\lib\celtix.jar
if not exist %CELTIX_JAR% (
      REM Assume it's a source (rather than a binary) distribution of Celtix
      set CELTIX_JAR=%CELTIX_HOME%\build\lib\celtix.jar
)

set CLASSPATH=%BASE_DIR%\build\classes;%CELTIX_JAR%;%CELTIX_HOME%\etc;%CLASSPATH%

set JVM_ARGS=-Djava.util.logging.config.file=%CELTIX_HOME%/etc/logging.properties

java %JVM_ARGS% helloworld.Client %*
```

CodeSnap 8: Script to run client.

```
@echo off

setlocal

if "%CELTIX_HOME%" == "" (
      echo You must set environment variable CELTIX_HOME to run this script.
      goto :eof
)

set BASE_DIR=%~dp0

set CELTIX_JAR=%CELTIX_HOME%\lib\celtix.jar
if not exist %CELTIX_JAR% (
      REM Assume it's a source (rather than a binary) distribution of Celtix
      set CELTIX_JAR=%CELTIX_HOME%\build\lib\celtix.jar
)

set CLASSPATH=%BASE_DIR%\build\classes;%CELTIX_JAR%;%CELTIX_HOME%\etc;%CLASSPATH%

set JVM_ARGS=-Djava.util.logging.config.file=%CELTIX_HOME%/etc/logging.properties

java %JVM_ARGS% helloworld.Server %*
```

CodeSnap 9: Script to run server.

Running Celtix applications using Ant

You can also run Celtix using targets from an Ant build file. The Ant targets shown in CodeSnap 2 show how a generic `-run.celtix` target can be reused for targets start the Hello World client and server. With these rules in place, you can now start the client and server using:

```
ant helloworld.Client
ant helloworld.Server
```