

Using Eclipse to develop grid services

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. Introduction	2
2. Setting up	5
3. Creating and configuring the project	13
4. Adding the project files	22
5. Working with a WS-Resource: Getting properties	29
6. Generate stubs and deploy the grid service	36
7. Running and Debugging using the Tomcat container	39
8. Summary	47

Section 1. Introduction

Who should take this tutorial?

This tutorial illustrates how to use the versatile Eclipse environment to facilitate development of Globus Toolkit V4 (GT4) grid services. The tutorial is written for Web service and grid developers who would like the convenience of orchestrating the whole grid service development process from within the Eclipse IDE on Windows-based platforms.

GT4 grid services can be tedious to develop because they frequently require the developer to juggle many artifacts (source files, WSDLs, client and server stubs, etc., many of which need to be auto-generated) and configuration steps (various iterations of compilation, linking, deployment, etc.). Without an integrated development environment (IDE) such as Eclipse, you must switch between many tools (editors, command shells, file managers, build tools, application containers, etc.) while iterating through the development process. With the right plug-ins and configuration, the Eclipse IDE can be used to manage all of these artifacts within a single project abstraction and coordinate all of the useful development activities from coding to deployment to debugging. By embedding the Apache Tomcat Web services container within Eclipse, any updates to the grid service implementation can be *immediately* reflected in the actively running grid service.

This is a tool tutorial (as opposed to a conceptual or programming tutorial) that walks through the creation of a simple WS-Resource within the Eclipse environment. A general familiarity with Java technology, Eclipse, Web services, the WS-Resource Framework (WSRF), and GT4 is recommended. (See [Resources](#) on page 47 for links to tutorials for more information regarding these subjects.

What is this tutorial about?

This tutorial illustrates how to configure an Eclipse project to coordinate the development of a GT4 grid service. A GT4 grid service is a Web services-based application that manages one or more "stateful resources," making it a WS-Resource in accordance with WSRF.

The WS-Resource model is quickly becoming the fundamental paradigm for the newest generation of service-oriented distributed computing infrastructures. The WSRF is a group of specifications designed to ease the burden of implementing and maintaining complex distributed systems by providing standard ways to interact with "stateful" resources that are abstracted as WS-Resources.

The GT4 is a WSRF-compliant set of software components (and related tools) from which developers can build distributed systems. Because of The Globus Alliance's depth of experience in the grid and distributed computing fields and

the widespread use of previous versions of the tool kit, GT4 is poised to be the premier enabling technology for grid services and applications.

In this tutorial, we are going to use the Eclipse IDE to develop, deploy, and debug a simple stateful Web service that uses WSRF to keep state information. This example Web service (`ProvisionDirService`) exposes the local file system's directory hierarchy to remote grid clients. (A service or collection of services that provision file system data into a grid is a common feature set for many existing grid infrastructure products.) It will allow you to list the contents of the current working directory and change the current working directory.

To get or change the current working directory, the `ProvisionDirService` will expose the `Cwd` (Current working directory) Resource Properties.

The design pattern for this grid service follows the "Singleton with ServiceResourceHome" pattern (as opposed to other patterns like the "Factory/Instance" pattern), meaning that this WS-Resource consists of a Web service paired with only one resource. The singleton resource in this case is a Java class that keeps the current working directory resource property and returns listings for that directory.

Although the operation of `ProvisionDirService` is not too complex, it is useful to illustrate the steps we'll take to configure the Eclipse project in this tutorial, and it also lays the groundwork for creating more complicated grid services. We will cover the following:

- Setting up the required tools and components
- Creating the Eclipse project
- Adding the project files (don't worry -- the sources for these project files are all provided for you in [Resources](#) on page47)
- Creating the build/deploy launch configuration that will facilitate the automatic generation of the remaining artifacts, assembling the Grid Archive (GAR), and deploying the grid service into the Web services container
- Using the launch configuration
- Running and debugging the grid service

(At the time of this writing, the first release of the GT4 IDE, an Eclipse plug-in that handles some of the code generation tasks covered in this tutorial, has been announced, but functionality is still incomplete.)

Prerequisites

To run the example from this tutorial, we'll need to obtain and install the following components/tools. More information regarding each can be found in the Setting up section.

- [Sun Java SDK V1.4.2](http://java.sun.com/j2se/1.4.2/download.html) (<http://java.sun.com/j2se/1.4.2/download.html>)

- [Eclipse IDE](http://www.eclipse.org/downloads/index.php) (http://www.eclipse.org/downloads/index.php)
 - [Apache Jakarta Tomcat V5.0](http://jakarta.apache.org/tomcat/) (http://jakarta.apache.org/tomcat/)
 - [Sysdeo Eclipse Tomcat Launcher Plug-in V3.0](http://www.sysdeo.com/eclipse/tomcatPlugin.html) (http://www.sysdeo.com/eclipse/tomcatPlugin.html)
 - [GT4 WS Core](#)
 - [globus-build-service](http://gsbt.sourceforge.net/content/view/14/31/) (http://gsbt.sourceforge.net/content/view/14/31/)
-

About the author

Duane Merrill has been developing grid computing and distributed data integration platforms for more than five years. He has been a contributor to the Legion Project at the University of Virginia and a core developer for the Avaki Corporation's distributed enterprise information integration product Avaki. He is currently obtaining his doctorate in computer science at the University of Virginia. He can be reached at duane@duanemerrill.com.

Section 2. Setting up

The Architecture

The Eclipse tool integration platform is a very popular, very extensible, well-documented IDE that can be configured to host all of the useful development activities from coding to deployment to debugging. We will use its Java Project abstraction to manage the artifacts (source files, WSDLs, client and server stubs, deployment configuration files, etc.) required by the `ProvisionDirService`.

The coding aspect of development will be handled in the Java Perspective of the Java Project. This Eclipse perspective provides the organization and editor views to manage the `ProvisionDirService` sources.

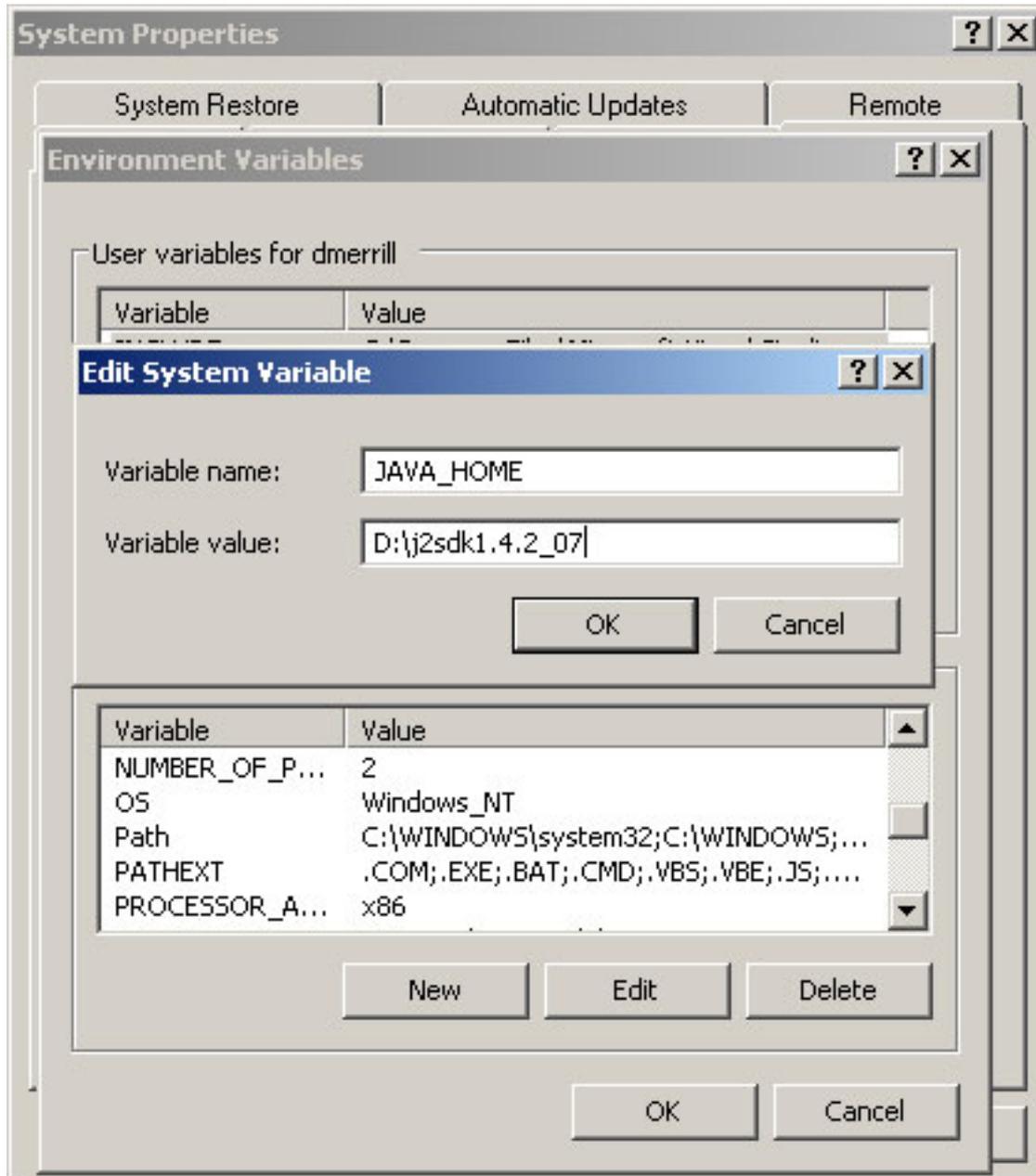
The artifact generation/service deployment process will be handled by the creation of a Launch Configuration that uses the Ant buildfiles supplied from the `globus-build-service` and `GT4 WS-Core` distributions. With a single mouse-click, this Launch Configuration will automate the process of:

- Generating the service stubs and parameter data structure classes
- Performing some tweaking to the service WSDL
- Assembling the service Grid Archive (GAR), which contains all the files and information the Web Services container needs to deploy our service
- Deploying the GAR into the GT4 directory tree
- Deploying the GT4 WSRF into Tomcat

To completely contain the iterative (develop/deploy/debug) development process within Eclipse, we need a mechanism for running and debugging the `ProvisionDirService`. As a GT4 grid service, the `ProvisionDirService` needs to be run within and managed by a Web services container. Although the GTK ships with its own stand-alone Web services container, this tutorial illustrates the use of the popular Apache Tomcat servlet container to host our sample grid service because of its rich feature set that extends beyond the simple hosting of grid services. We use the Sysdeo Tomcat Launcher plug-in to integrate Tomcat into the Eclipse environment.

By this point, it is assumed that you have downloaded and installed the Sun Java SDK V1.4.2. (This tutorial was written using V1.4.2.07.) Be sure to set the `JAVA_HOME` system environment variable to the root of the SDK installation. This can be done by right-clicking My Computer on the Windows desktop, selecting Properties, selecting the Advanced tab, clicking the Environment Variables button and adding `JAVA_HOME` to the list of System variables, as shown in Figure 1.

Figure 1. Setting the `JAVA_Home` system environment variable



The remainder of this section reviews the remaining required tools and gives an explanation of what each tool is for, as well as helpful information on setup and configuration.

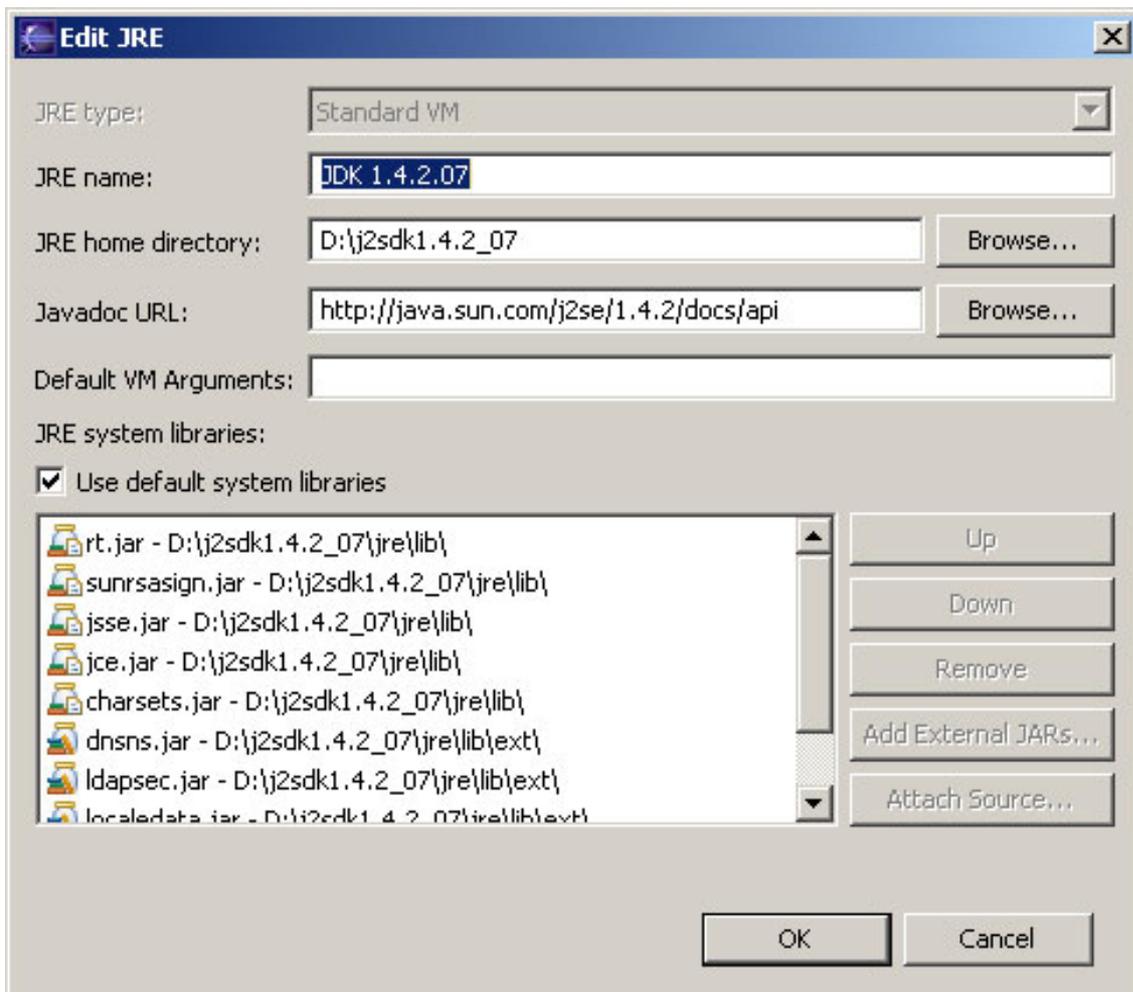
Eclipse

The Eclipse Platform is an extensible open-source tool integration platform for "anything and yet nothing in particular." It provides building blocks and a foundation for constructing and running integrated software development tools. for this tutorial, we will be using it as a Java IDE.

The Eclipse Platform distribution should be obtained and unzipped into the local file system. For this tutorial, Eclipse V3.0.2 was installed into the `\dev\eclipse` directory.

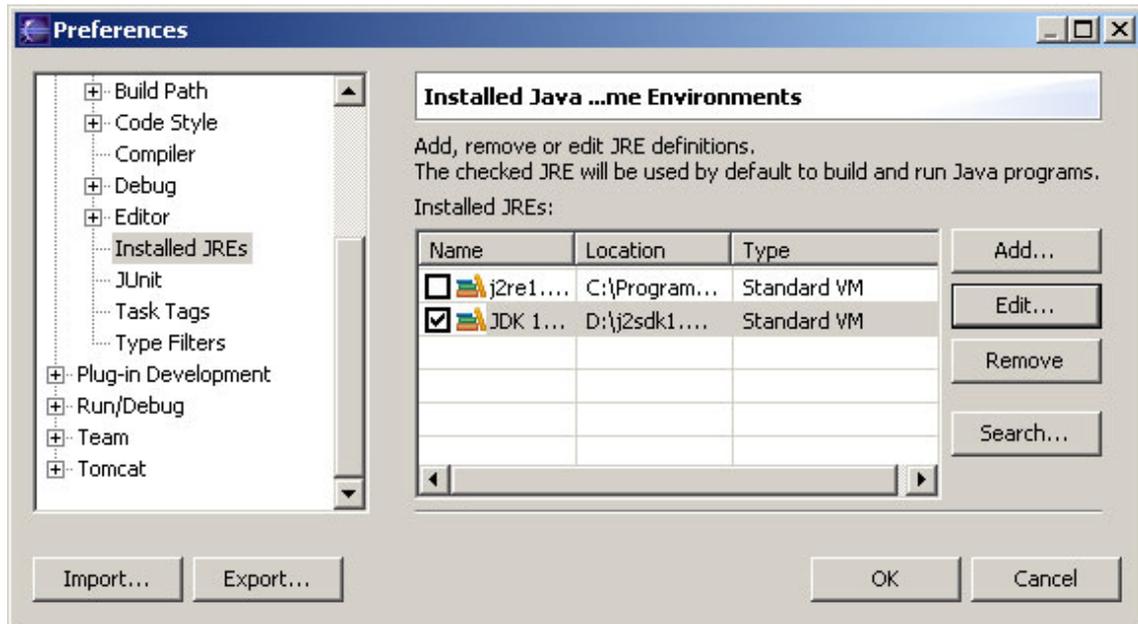
Eclipse normally uses the Java Runtime Environment (JRE) installed in the `\Program Files\Java` directory, rather than a full JDK, but Tomcat must have access to a JDK. If you have not done so already, add the JDK to the list of installed JREs that Eclipse knows about. Go to the **Window > Preferences** menu and open the Eclipse Preferences dialog. Select the **Java > Installed JREs** page shown in Figure 2. Click **Add** to add the JDK to the list.

Figure 2. Add the JDK to Installed JREs



Check it to make it the default, as shown in Figure 3.

Figure 3. Setting the default JDK



Click **OK** to finish the configuration.

Tomcat

Apache Jakarta Tomcat is the servlet container used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies. In this tutorial, we use Tomcat as the Web services container for the GT4 WSRF Web application.

The Tomcat distribution should be obtained and unzipped into the local file system. For this tutorial, we'll install V5.0.28 in the \dev\jakarta-tomcat-5.0.28 directory.

Before we can run the "admin" and "manager" programs in Tomcat, we must first define a user for this. Insert the following line into the \dev\jakarta-tomcat-5.0.28\conf\tomcat-users.xml:

```
<user username="admin" password="admin" roles="admin,manager" />
```

Sysdeo Tomcat Launcher

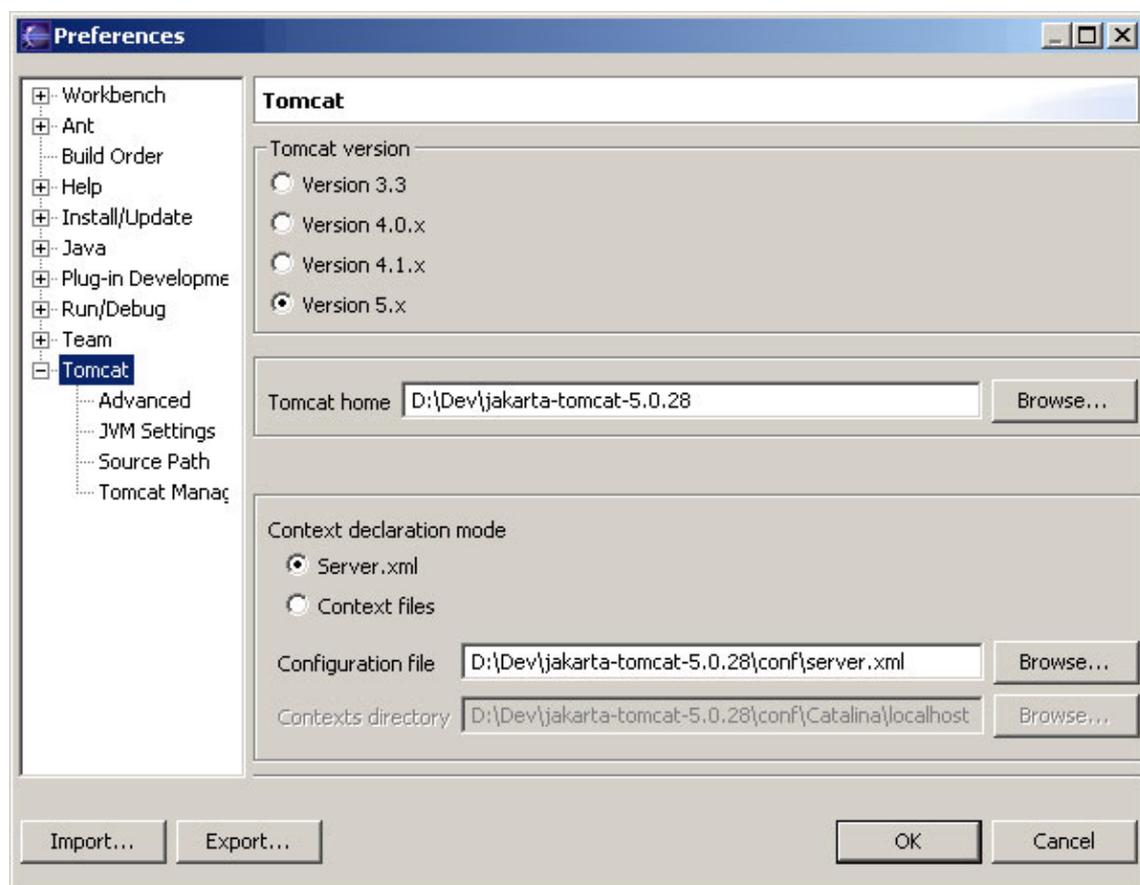
The Sysdeo Tomcat Launcher is an Eclipse plug-in necessary for managing the Tomcat Web services container from within the Eclipse IDE. Although there are several Tomcat plug-ins for Eclipse on the market, the (free) Sysdeo Tomcat Launcher is ostensibly the most popular and well known. The Sysdeo Tomcat

Launcher adds several menu-bar buttons to Eclipse for the starting and stopping of the embedded Tomcat container, and also registers the Tomcat process with the Eclipse debugger.

The Sysdeo Tomcat Launcher plug-in should be obtained and unzipped into the Eclipse plug-ins directory. (Its features will be made available the next time Eclipse is started.)

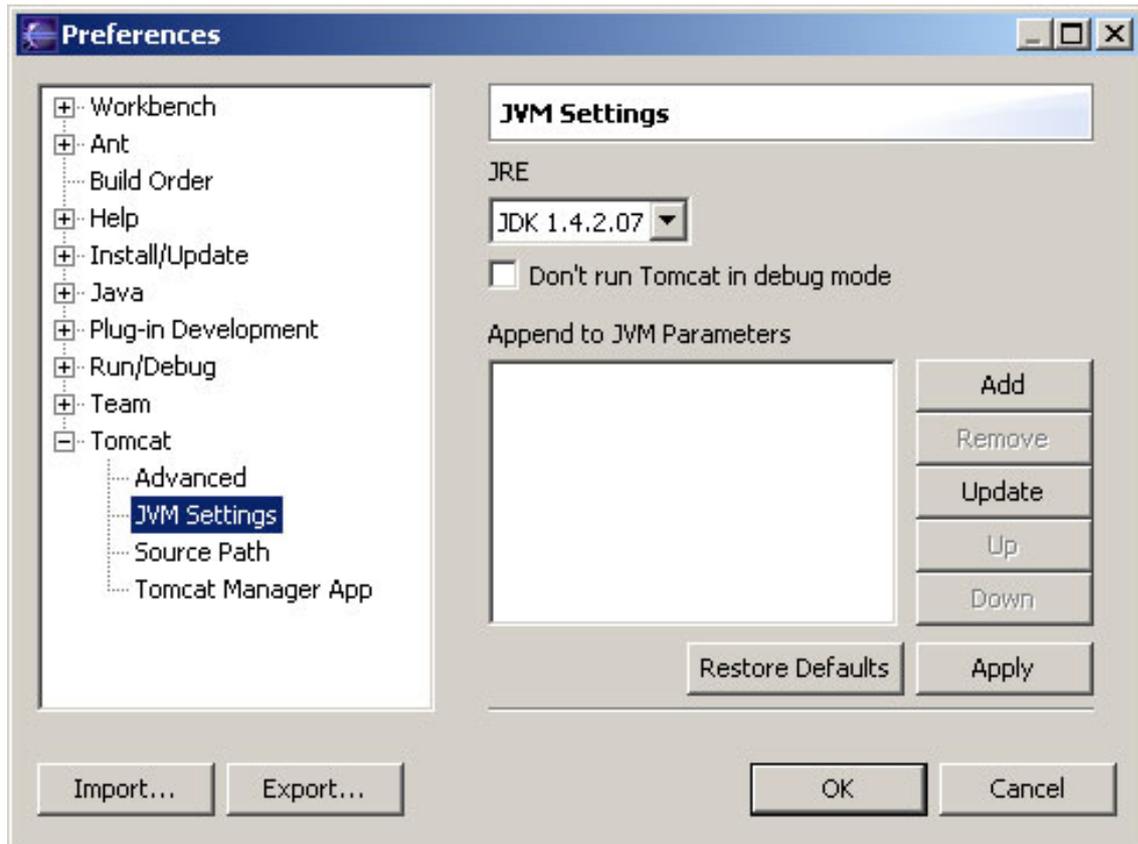
Before using the plug-in, we need to configure several of its preferences. Go to the **Window > Preferences** menu and open the Eclipse Preferences dialog. Select the Tomcat page. Select Version 5.x radio button and fill in the Tomcat Home textbox, as shown in Figure 4:

Figure 4. Setting the Tomcat version



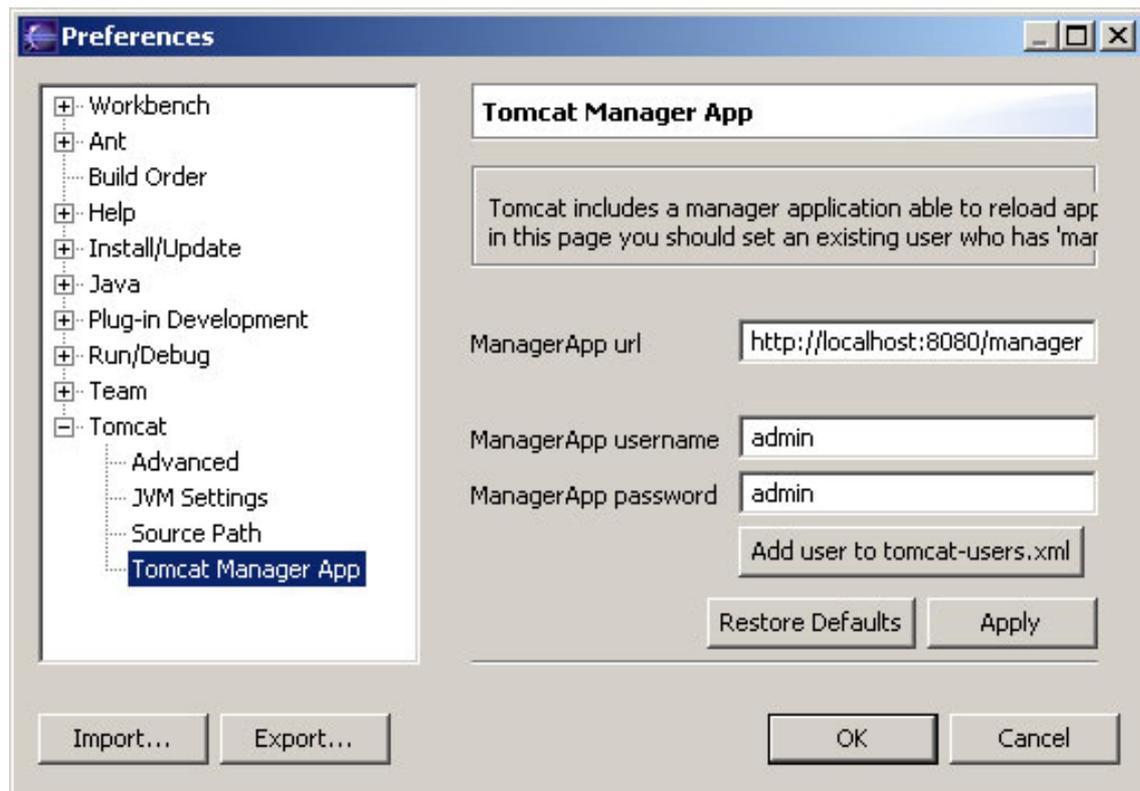
Additionally, ensure that the JVM Settings page (shown in Figure 5) are configured to use the JDK JRE (the Preferences page may need to be closed and reopened if it wasn't closed after setting the JDK as an Installed JRE).

Figure 5. JVM Settings page



Finally, enter the manager role username/password in the Tomcat Manager App page, as shown in Figure 6.

Figure 6. Tomcat Manager App page



GT4 WS-Core

The GT4 WS-Core is the Globus Toolkit implementation of the WSRF and the Web Service Notification (WSN) family of standards. It provides the API and tools for building stateful Web services (WS-Resources).

The GT4 WS-Core distribution should be obtained and unzipped to the local file system. Download the V3.9.5 source-code distribution (so we can step into and view the WS-Core source while debugging) and unzip it into `\dev\GTK`.

Additionally, we need to set a `GLOBUS_LOCATION` environment variable to `\dev\GTK`; follow the same instructions as for setting the `JAVA_HOME` environment variable described earlier in this section.

To build the WS-Core from source, you must first obtain the Apache Ant build tool from (<http://ant.apache.org/bindownload.cgi>). To build it, open a CMD shell and type the following:

```
D:\>cd %GLOBUS_LOCATION%\ws-core-3.9.5
D:\Dev\GTK\ws-core-3.9.5>ant all
```

This command builds and installs the WS-Core into `\dev\GTK`. If necessary, you can find further build and administrative instructions for the WS-Core in the [WS-Core Administrators Guide](#).

The globus-build-service

The `globus-build-service` distribution is a Ant buildfile (and related shell scripts) to assist with the building of GAR files. This buildfile is a variation of the one used in the GT4 Programmer's Tutorial (see [Resources](#) on page 47). Obtain the distribution and unzip these files into the `\dev\GTK\etc` directory.

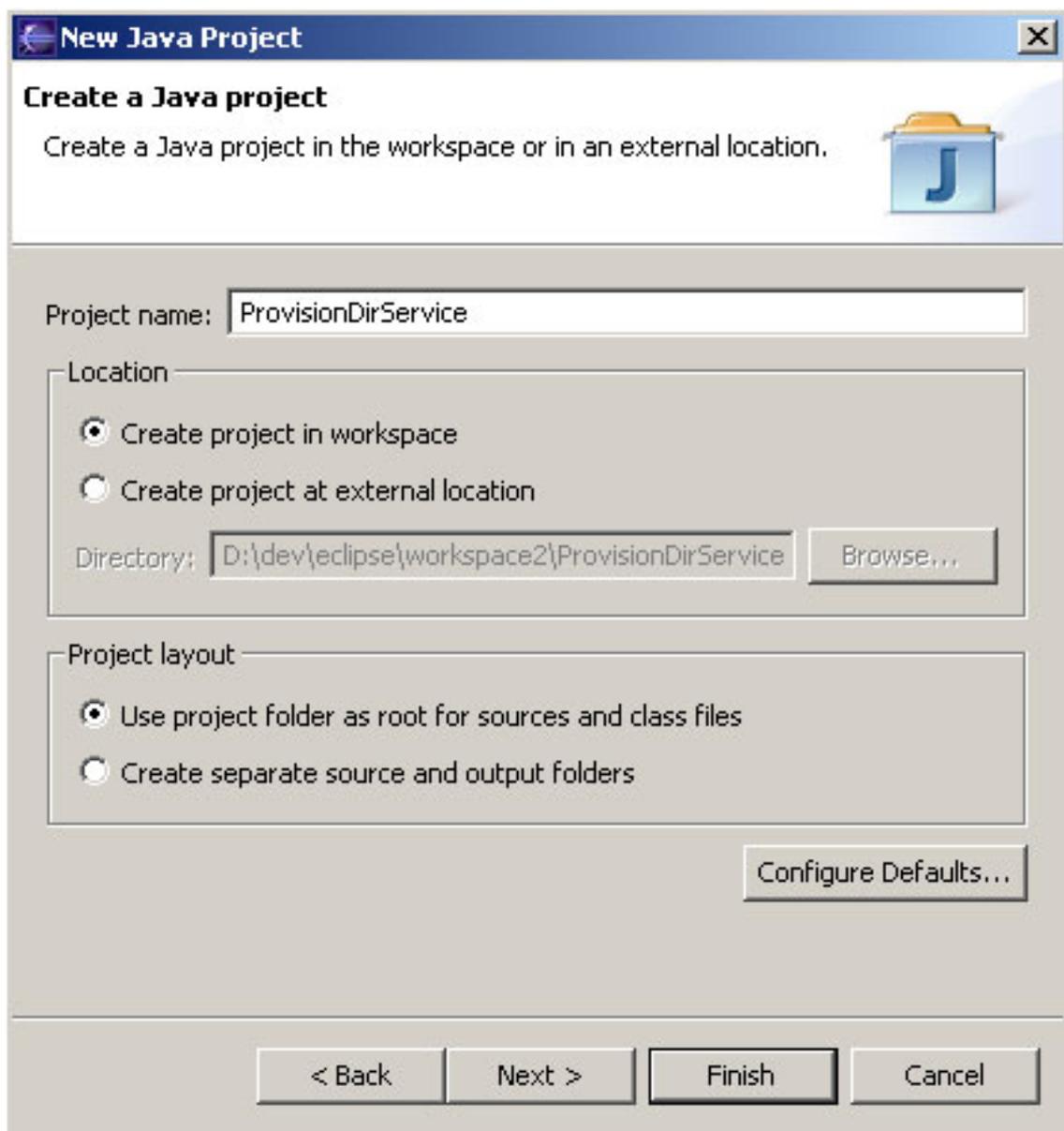
The next section illustrates how to create a new Java Project for our `ProvisionDirService` and shows how to prepare the project for the addition of the service source files.

Section 3. Creating and configuring the project

Create a new project

We start by creating a new Java Project called `ProvisionDirService`. Select **File > New > Project...** and select **Java > Java Project** from the selection wizard. Click **Next** and enter `ProvisionDirService` in the Project Name textbox. Accept the remaining project creation defaults by clicking **Finish**.

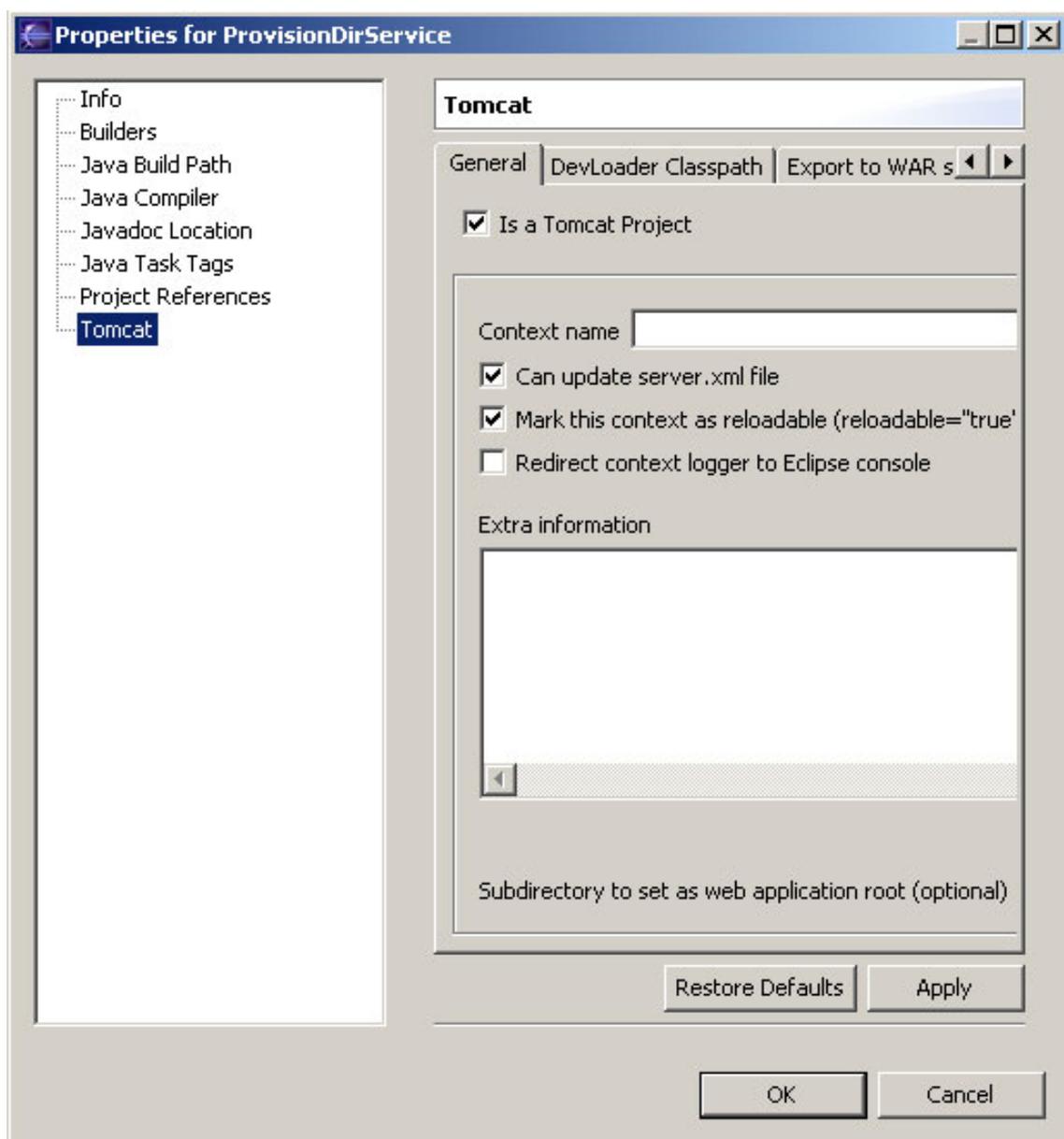
Figure 7. New Java project dialog



Make the project a Tomcat project

The first thing we need to do is to make this project a "Tomcat Project." Doing so enables Tomcat to run from `.class` files inside our project as soon as they are compiled by Eclipse (which happens every time they are saved). Hence, minor changes to the service logic will be reflected in the running service without having to regenerate or redeploy any GARs. Open the project properties page shown in Figure 8 (select Properties from the project's pop-up menu), select the Tomcat page, and check the "Is a Tomcat Project" checkbox.

Figure 8. Project properties page



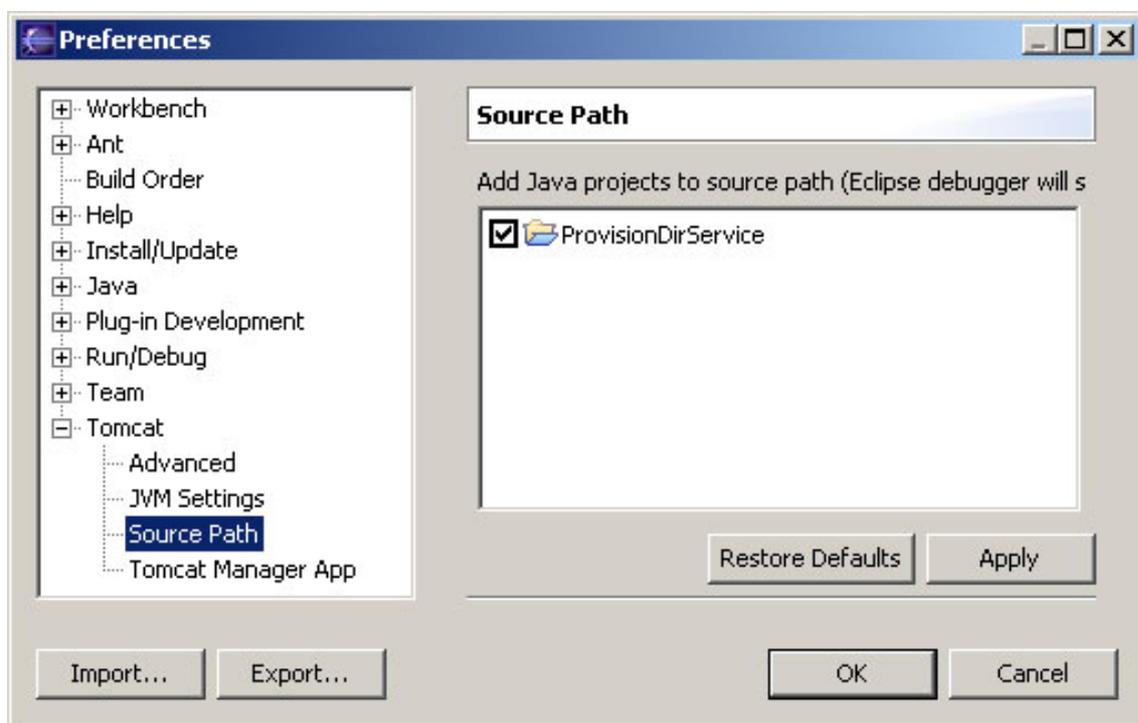
Click **OK**.

Add project source to Tomcat source path

Now we need to add the project's source to the Tomcat source path. The last step allows Tomcat to reload any updates to the implementation. This step allows the live debugger to pull up the fresh source when debugging.

Go to **Window > Preferences** and select the **Tomcat > Source Path** page (shown in Figure 9). Select the checkbox next to our `ProvisionDirService` project.

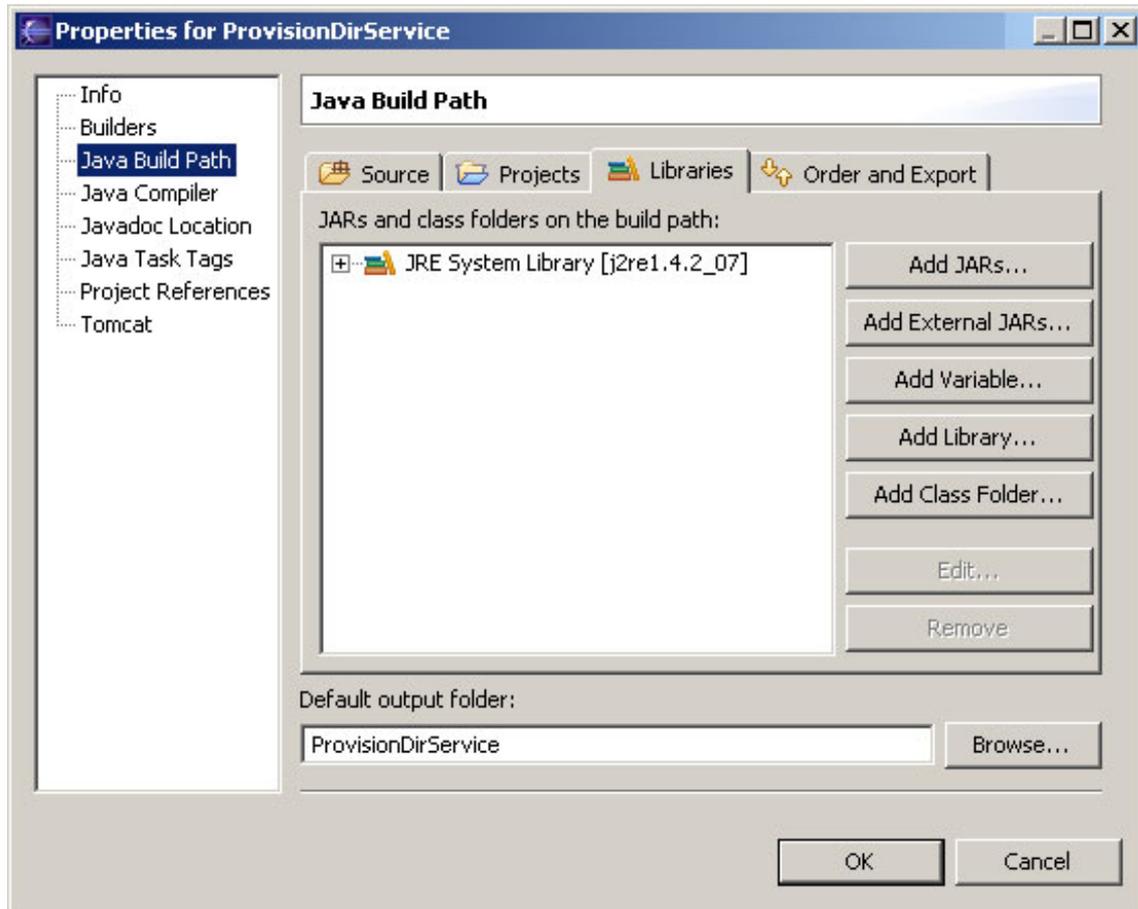
Figure 9. Source Path page



Click **OK**.

Include the WS-Core classes as the GT4 Library

The second thing we need to do is to update the Java Build Path Libraries to include the WS-Core JARs so the Eclipse editor will be able to find the WSRF classes our grid service sources will import. Do this by opening the project properties page (select Properties from the project's pop-up menu), selecting the Java Build Path page, and clicking the Libraries tab. As you can see in Figure 10, only the JRE System Library is currently imported.

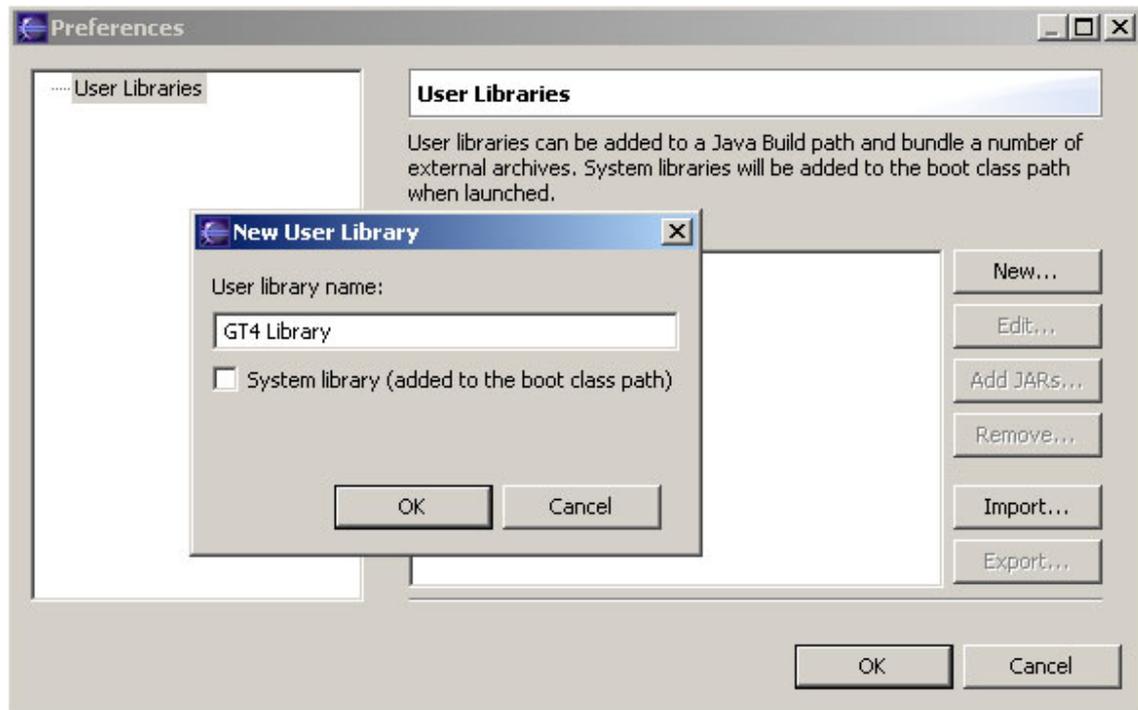
Figure 10. Updating the Java Build Path Libraries

Use the **Add Library...** button to add a user library and click **Next**.

Create the GT4 library

To create a user library from the GT4 library directory, use the **User Libraries...** button. Click **New...** in the User Libraries dialog (see Figure 11) and create a Library called GT4 Library.

Figure 11. Create a new user library

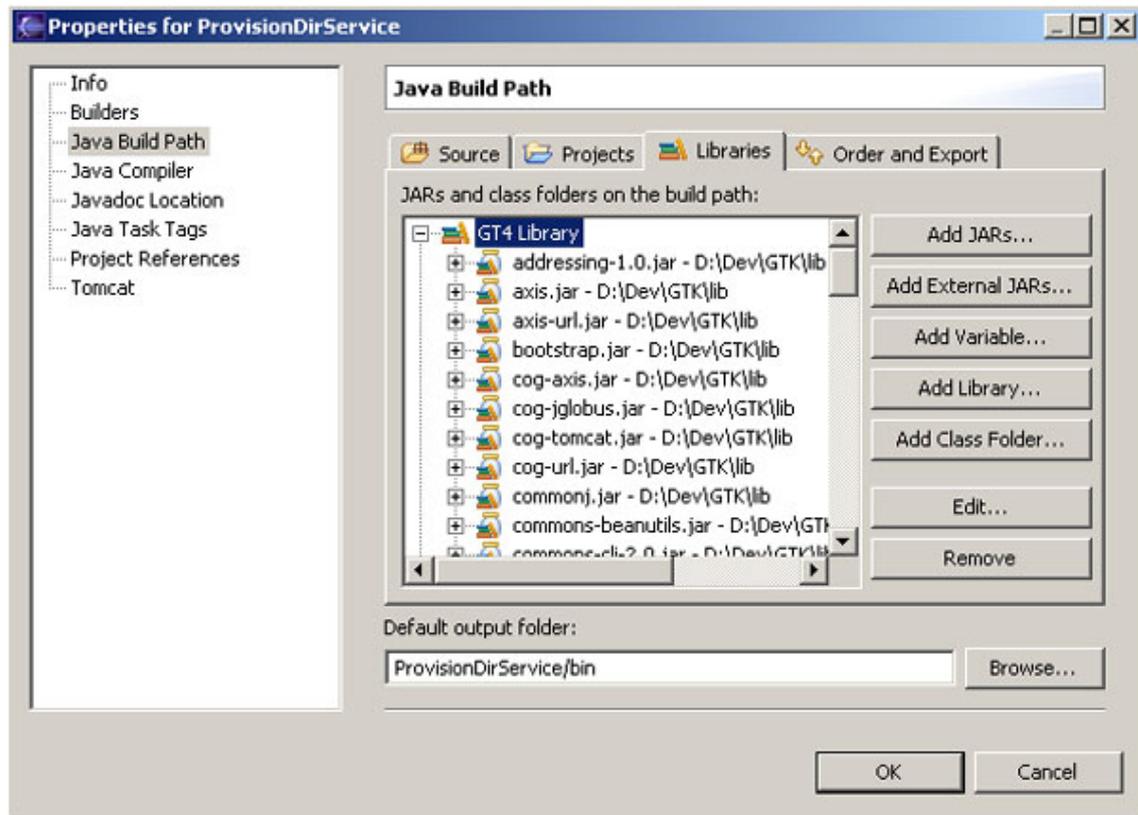


Click **Add JARs...** and select and add all of the JARs in `%GLOBUS_LOCATION%\lib` to the library. Selecting **OK** and **Finish** leaves us with the properly configured Libraries tab.

Finish the configuration

The Libraries tab should look like Figure 12.

Figure 12. Updated libraries tab



Click **OK** to finish the configuration.

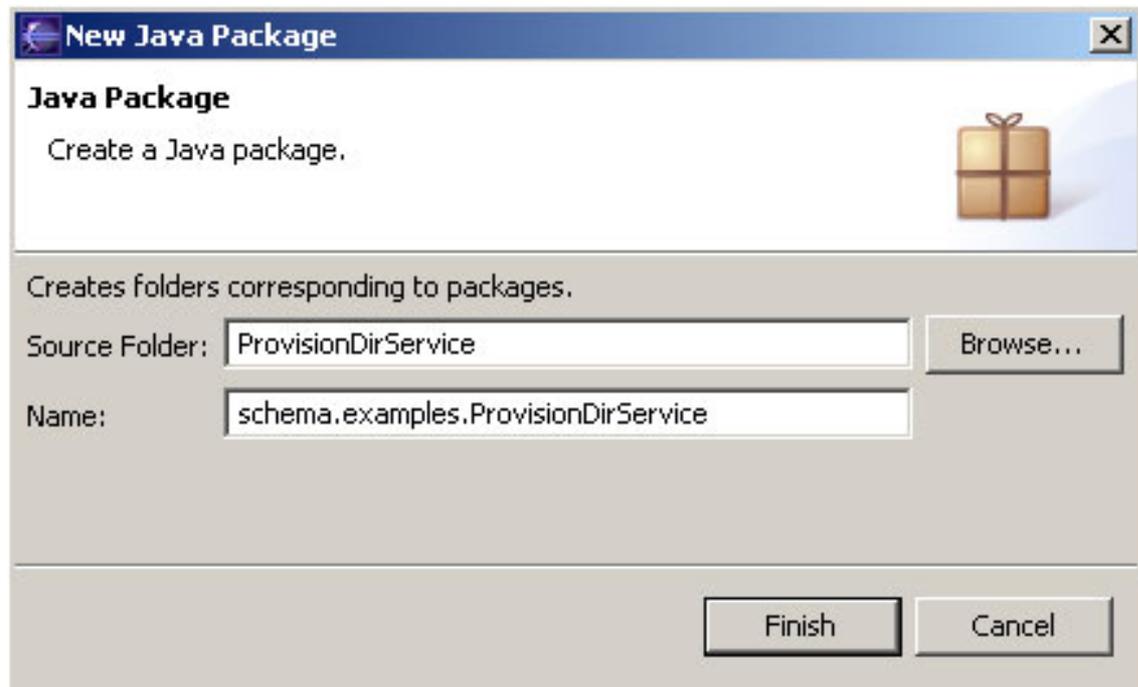
Adding folders and packages

Before adding the service sources, we will need to create the folder and package structure in which to put them.

Add the Navigator view by selecting **Window > Show View > Navigator** from the menu.

To start with, we need to create the folder location in which to place the WSDL document that describes our service interface. We do this by creating a **New > Package** called `schema.examples.ProvisionDirService` so that a `schema/examples/ProvisionDirService/` path extends from the project root (see Figure 13).

Figure 13. New Java package dialog

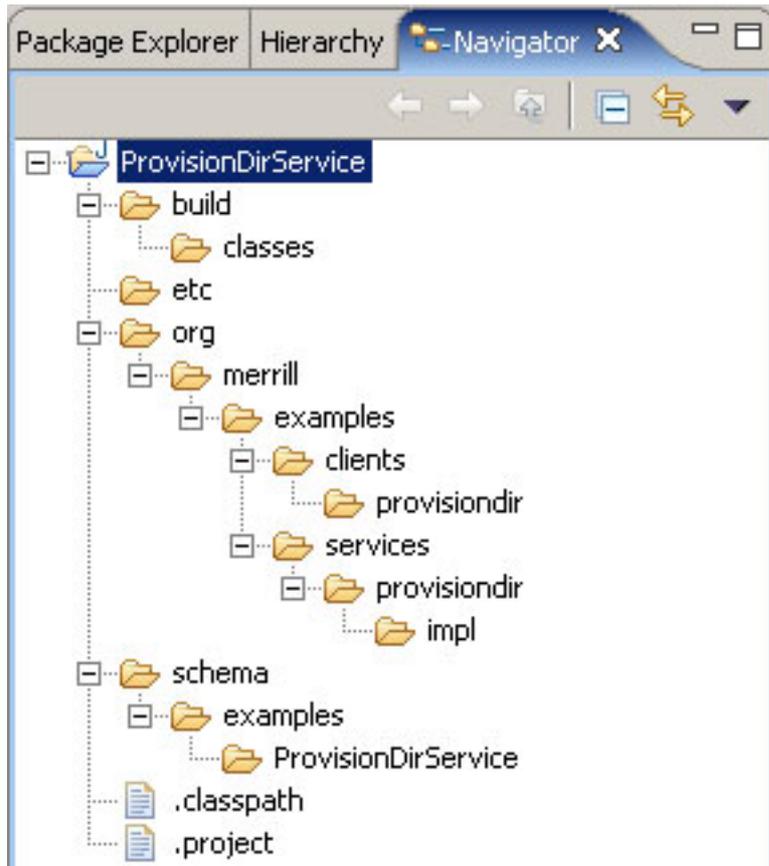


Go ahead and create the following additional four packages:

1. `etc`
2. `build.classes`
3. `org.merrill.examples.services.provisiondir.impl`
4. `org.merrill.examples.clients.provisiondir`

At this point, our Navigator view should resemble Figure 14.

Figure 14. Updated Navigator view



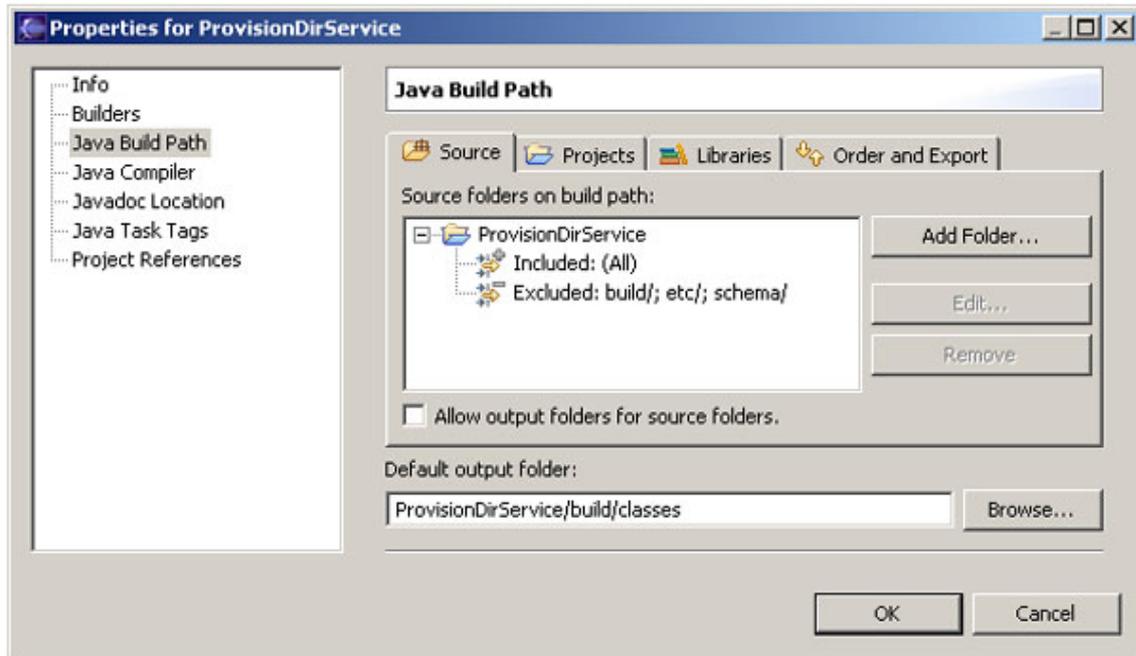
Setting the output destination

The last thing we need to do before adding source files to our project is configure the project's build output destination.

We want Eclipse to compile our Java sources and put the `.class` files in `build/classes`, so we need to go to the project Properties page again and configure the Java Build Path page. Change the Default output folder to `/ProvisionDirService/build/classes`.

However, we don't want Eclipse to copy nonsource files to the build output folder. Expand the `ProvisionDirService` source folder, select `Excluded` and click **Edit...** In the Inclusion and Exclusion Patterns dialog, click the **Add Multiple...** button and add the `build`, `etc`, and `schema` folders as exclusion patterns. The Java Build Path page's Source tab should now look like Figure 15.

Figure 15. Source tab for Java Build Path page



Click **OK**. Now we can add the actual files.

Section 4. Adding the project files

ProvisionDir.wsdl

The `ProvisionDirService` project is finally ready for its source files. There are five of them, and they are described in the following subsections. (Tip: It is usually easiest to use Eclipse's text editor to manipulate `.wsdl` or `.xml` files, instead of having it launch external applications.)

The `ProvisionDir.wsdl` file is the XML document that describes the service interface. It should be placed in the `schema/examples/ProvisionDirService` project folder. The service interface describes how the outside world can interact with our service, specifically the operations that can be performed on it. The service interface is often called the port type. The source for this WSDL document is:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ProvisionDirService"
  targetNamespace="http://examples.merrill.org/provisiondir/ProvisionDirService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://examples.merrill.org/provisiondir/ProvisionDirService"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsrflw="http://docs.oasis-open.org/wsrflw/2004/06/wsrflw-WS-Resource
Lifetime-1.2-draft-01.wsdl"
  xmlns:wsrp="http://docs.oasis-open.org/wsrflw/2004/06/wsrflw-WS-Resource
Properties-1.2-draft-01.xsd"
  xmlns:wsrpw="http://docs.oasis-open.org/wsrflw/2004/06/wsrflw-WS-Resource
Properties-1.2-draft-01.wsdl"
  xmlns:wslpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <wsdl:import
    namespace=
      "http://docs.oasis-open.org/wsrflw/2004/06/wsrflw-WS-ResourceProp
erties-1.2-draft-01.wsdl"
    location="../../wsrflw/properties/WS-ResourceProperties.wsdl" />

  <types>
  <xsd:schema targetNamespace="http://examples.merrill.org/provisiondir/Provision
DirService"
    xmlns:tns="http://examples.merrill.org/provisiondir/ProvisionDirService"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:import
      namespace="http://schemas.xmlsoap.org/ws/2004/03/addressing"
      schemaLocation="../../ws/addressing/WS-Addressing.xsd" />

    <!-- Requests and responses -->

    <xsd:element name="listDir">
      <xsd:complexType/>
    </xsd:element>
    <xsd:element name="listDirArrayResponse">
      <xsd:complexType>
```

```
        <xsd:sequence>
            <xsd:element name="a" type="xsd:string" minOccurs="0" maxOccurs="un
bounded" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="changeCwd" type="xsd:string" />
<xsd:element name="changeCwdResponse">
    <xsd:complexType/>
</xsd:element>

<!-- Resource properties -->

<xsd:element name="Cwd" type="xsd:string" />

<xsd:element name="ProvisionDirResourceProperties">
<xsd:complexType>
    <xsd:sequence>
        <xsd:element ref="tns:Cwd" minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
</types>

<message name="listDirInputMessage">
    <part name="parameters" element="tns:listDir" />
</message>
<message name="listDirOutputMessage">
    <part name="parameters" element="tns:listDirArrayResponse" />
</message>

<message name="changeCwdInputMessage">
    <part name="parameters" element="tns:changeCwd" />
</message>
<message name="changeCwdOutputMessage">
    <part name="parameters" element="tns:changeCwdResponse" />
</message>

<portType name="ProvisionDirPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty"
    wsrp:ResourceProperties="tns:ProvisionDirResourceProperties">

    <operation name="listDir">
        <input message="tns:listDirInputMessage" />
        <output message="tns:listDirOutputMessage" />
    </operation>

    <operation name="changeCwd">
        <input message="tns:changeCwdInputMessage" />
        <output message="tns:changeCwdOutputMessage" />
    </operation>

</portType>

</definitions>
```

Although WSDL documents are generally not meant for human consumption, there are several parts you should be aware of:

- The `<portType>` element describes our operations: `listDir` and `changeCwd`. Note that we support resource properties by deriving the port type from the Resource Properties WSDL. (Note also that while this does work with the software, WSRF officially does away with this older style of using extending port types; technically, we should include the full definition.) Both operations consist of an input and an output message, described above with `<message>` tags.
 - Because we are using the document/literal style of message exchange, the messages must have one message part (even if the logical method takes zero or multiple parameters or returns zero or multiple return values.)
 - These "wrapper" message parts are described in the `<!-- Requests and responses -->` section of the WSDL, which describes the schema format for the request and response message parts. The `listDir` element has no subfields because the logical `listDir()` method requires no parameters. The `listDirArrayResponse` element serves as a wrapper for the return value; an array of strings that denote the items in the current working directory. The `changeCwd` element is a string type and the `changeCwdResponse` element has no subfields because the method does not return anything.
 - Our resource properties (the current working directory) are described in the `<!-- Requests and responses -->` section.
-

ProvisionDirQName.java

The `ProvisionDirQName.java` file is a convenient interface class containing the QName URI/namespace constants relevant to our grid service. It should be placed in the `org/merrill/examples/services/provisiondir/impl` project folder. By having our service (and client) classes implement this interface, we can reference these constants without having to replicate them throughout the project. The source for this Java class is:

```
package org.merrill.examples.services.provisiondir.impl;

import javax.xml.namespace.QName;

public interface ProvisionDirQNames {
    public static final String NS = "http://examples.merrill.org/provisiondir/ProvisionDirService";

    public static final QName RESOURCE_PROPERTIES = new QName(NS,
        "ProvisionDirResourceProperties");

    public static final QName RESOURCE_REFERENCE = new QName(NS,
        "ProvisionDirResourceReference");

    /* Insert ResourceProperty Qnames here. */

    public static final QName RP_CWD = new QName(NS, "Cwd");
}
```

A change in state requires a change in one or more of these values, and vice versa.

ProvisionDirService.java

The `ProvisionDirService.java` file is the service implementation that provides the core functionality for exposing local directory information. It should be placed in the

`org/merrill/examples/services/provisiondir/impl` project folder.

The source for this Java class is:

```
package org.merrill.examples.services.provisiondir.impl;

import java.rmi.RemoteException;

import org.globus.wsrp.ResourceContext;
import org.globus.wsrp.Resource;
import org.globus.wsrp.ResourceProperties;
import org.globus.wsrp.ResourceProperty;
import org.globus.wsrp.ResourcePropertySet;
import org.globus.wsrp.impl.ReflectionResourceProperty;
import org.globus.wsrp.impl.SimpleResourcePropertySet;

import org.merrill.examples.provisiondir.ProvisionDirService.ListDirArrayResponse;
import org.merrill.examples.provisiondir.ProvisionDirService.ChangeCwdResponse;

public class ProvisionDirService implements Resource, ResourceProperties {

    /* Resource Property set */
    private ResourcePropertySet propSet;

    /* Insert resource properties here. */
    private String cwd;

    /* Constructor. Initializes RPs */
    public ProvisionDirService()
        throws RemoteException {

        this.propSet = new SimpleResourcePropertySet(
            ProvisionDirQNames.RESOURCE_PROPERTIES);

        try {
            /* Initialize Resource Properties here.*/
            ResourceProperty cwdRP = new ReflectionResourceProperty(
                ProvisionDirQNames.RP_CWD,
                "Cwd",
                this);
            this.propSet.add(cwdRP);
            setCwd("/");
        } catch (Exception e) {
            throw new RuntimeException(e.getMessage());
        }
    }
}
```

```
public ListDirArrayResponse listDir() throws RemoteException {

    java.io.File currentDir = new java.io.File(cwd);
    return new ListDirArrayResponse(currentDir.list());
}

public ChangeCwdResponse changeCwd(String newCwd) throws RemoteException {

    setCwd(newCwd);
    return new ChangeCwdResponse();
}

/* Insert get/setters for Resource Properties here.*/

public String getCwd() {
    return cwd;
}

public void setCwd(String cwd) {
    this.cwd = cwd;
}

/* Required by interface ResourceProperties */
public ResourcePropertySet getResourcePropertySet() {
    return this.propSet;
}
}
```

There are several parts of this class that you should be aware of:

- Resource properties (such as our current working directory) are maintained as private fields of the class (see the section under */* Insert resource properties here. */*).
- We need to initialize them and insert them into the set of resource properties in the constructor.
- We need to create simple getter/setter methods for our resource properties whose method names follow the `get<field name>` and `set<field name>` pattern.
- Our operation methods (in this case the `listDir()` and `changeCwd()` methods) will always have a single return type; name them after the input and return types that were defined in the WSDL. Don't worry if the editor shows red compilation errors -- we haven't generated the stub classes that represent these types yet.

For more information about using WSRF classes, see the Understanding WSRF tutorials under [Resources](#) on page 47 .

deploy-jndi-config.xml

The `deploy-jndi-config.xml` file is the JNDI deploy file that enables the GT4 WSRF implementation to locate the resource-home for this service. It should be placed in the `/org/merrill/examples/services/provisiondir` project folder. The source for this configuration file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<jndiConfig xmlns="http://wsrf.globus.org/jndi/config">

  <service name="examples/ProvisionDirService">
    <resource name="home" type="org.globus.wsrf.impl.ServiceResourceHome">
      <resourceParams>

        <parameter>
          <name>factory</name>
          <value>org.globus.wsrf.jndi.BeanFactory</value>
        </parameter>

      </resourceParams>

    </resource>
  </service>

</jndiConfig>
```

The file is pretty empty at the moment, but notice that two files must be imported for all of this to work. Typical versions of the `WS-ResourceProperties.wsdl` and `WS-Addressing.xsd` files reference directories you might not have created on your machine, so for the sake of simplicity, you can download simplified versions from the tutorial [Resources](#) on [page47](#) .

Now that we have the framework, let's start filling it out.

deploy-server.wsdd

The `deploy-server.wsdd` file is the WSDD configuration file that tells the Web Service container (Tomcat) how to publish the Web service. It should be placed in the `/org/merrill/examples/services/provisiondir` project folder. The source for this configuration file is:

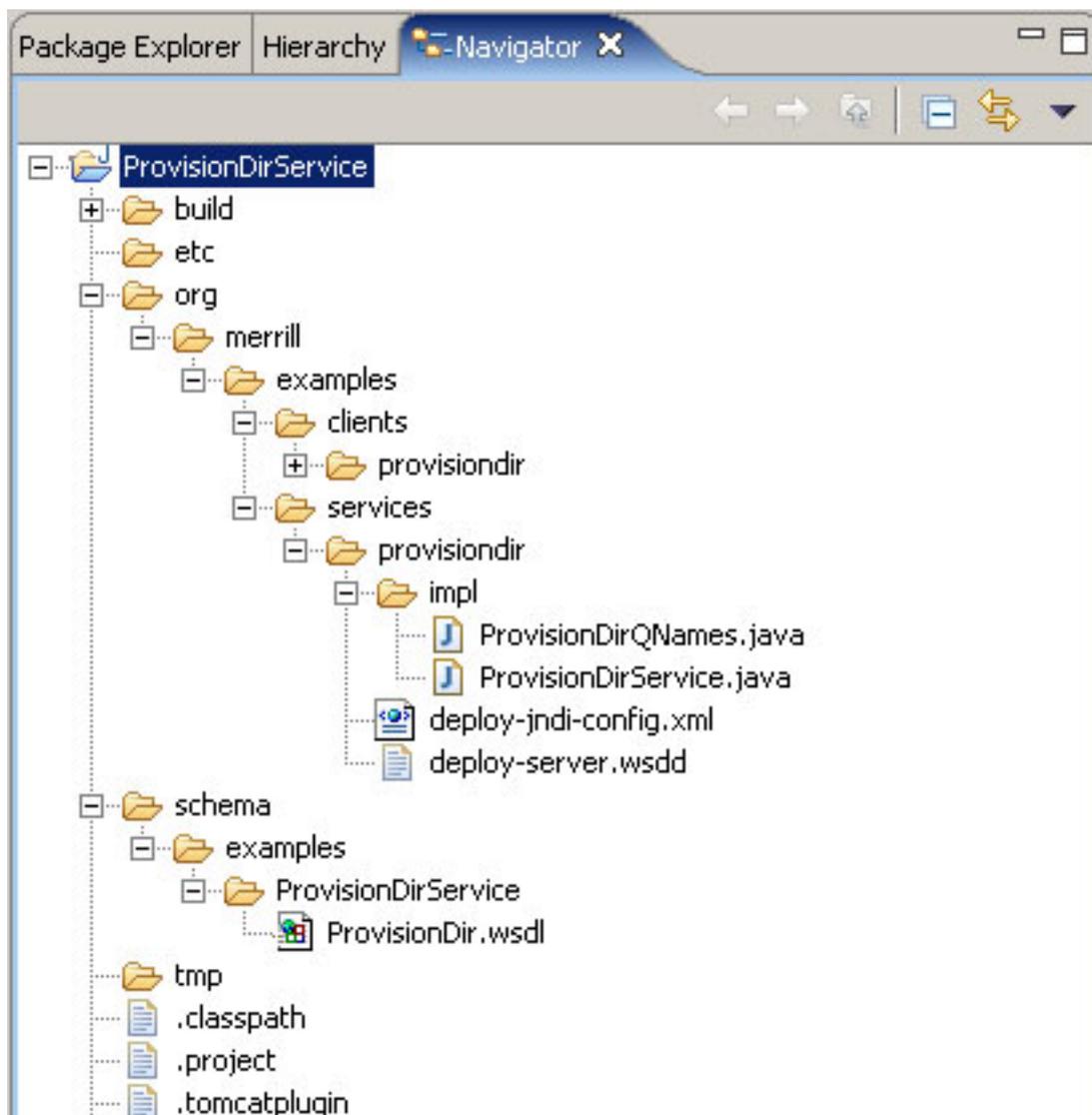
```
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultServerConfig"
  xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <service name="examples/ProvisionDirService" provider="Handler" use="literal"
    style="document">
    <parameter name="className" value="org.merrill.examples.services.provision
dir.impl.ProvisionDirService"/>
    <wsdlFile>share/schema/examples/ProvisionDirService/ProvisionDir_ser
vice.wsdl</wsdlFile>
    <parameter name="allowedMethods" value="*" />
  </service>
</deployment>
```

```
<parameter name="handlerClass" value="org.globus.axis.providers.RPCProv  
ider"/>  
<parameter name="scope" value="Application"/>  
<parameter name="providers" value="GetRPPProvider DestroyProvider"/>  
<parameter name="loadOnStartup" value="true"/>  
</service>  
  
</deployment>
```

At this point, the package explorer should resemble Figure 16.

Figure 16. Package explorer



Section 5. Working with a WS-Resource: Getting properties

Launch configuration overview

In this section, we'll cover the process of creating the Ant launch configuration necessary for building and deploying our grid service.

A launch configuration is a mechanism that Eclipse provides for executing one or more Ant buildfile targets. For the `ProvisionDirService` project, we want to create a launch configuration that, with the click of a mouse, will automatically:

- Generate the service stubs and parameter data structure classes
- Perform some tweaking to the service WSDL
- Assemble the service GAR, which contains all the files and information the Web Services container needs to deploy our service
- Deploy the GAR into the GT4 directory tree
- Deploy the GT4 WSRF into Tomcat

The `globus-build-service` and `WS-Core` distributions contain several Ant buildfiles that can cumulatively do all of these tasks, but we will need to create a "master" Ant buildfile to orchestrate them all from within one Launch Configuration.

buildservice.xml

The `buildservice.xml` Ant buildfile serves as the "master" buildfile that coordinates activities from the `globus-build-service` and `WS-Core` buildfiles. This file should be placed in the project root directory. The source for this buildfile is:

```
<?xml version="1.0"?>
<!--
-->

<project default="all" name="Grid Service Buildfile" basedir=". ">
  <description>
    Grid Service Buildfile
  </description>

  <property environment="env" />

  <target name="all">
    <ant antfile="${build.gar}" target="clean" />
    <ant antfile="${build.gar}" target="all" />
    <ant antfile="${build.packages}" target="deployGar" />
    <ant antfile="${build.tomcat}" target="deploySecureTomcat" dir="/Dev/GTK/sha
```

```
re/globus_wsrf_common/tomcat" />
  </target>

</project>
```

This "master" buildfile calls the following external buildfiles:

- The Globus-Build-Service build.xml Ant buildfile (which resides where it was unzipped in /dev/GTK/etc/)
- The WS-Core build-packages.xml Ant buildfile (/Dev/GTK/share/globus_wsrf_common/build-packages.xml)
- The WS-Core tomcat.xml Ant buildfile (/Dev/GTK/share/globus_wsrf_common/tomcat/tomcat.xml)

buildservice.properties

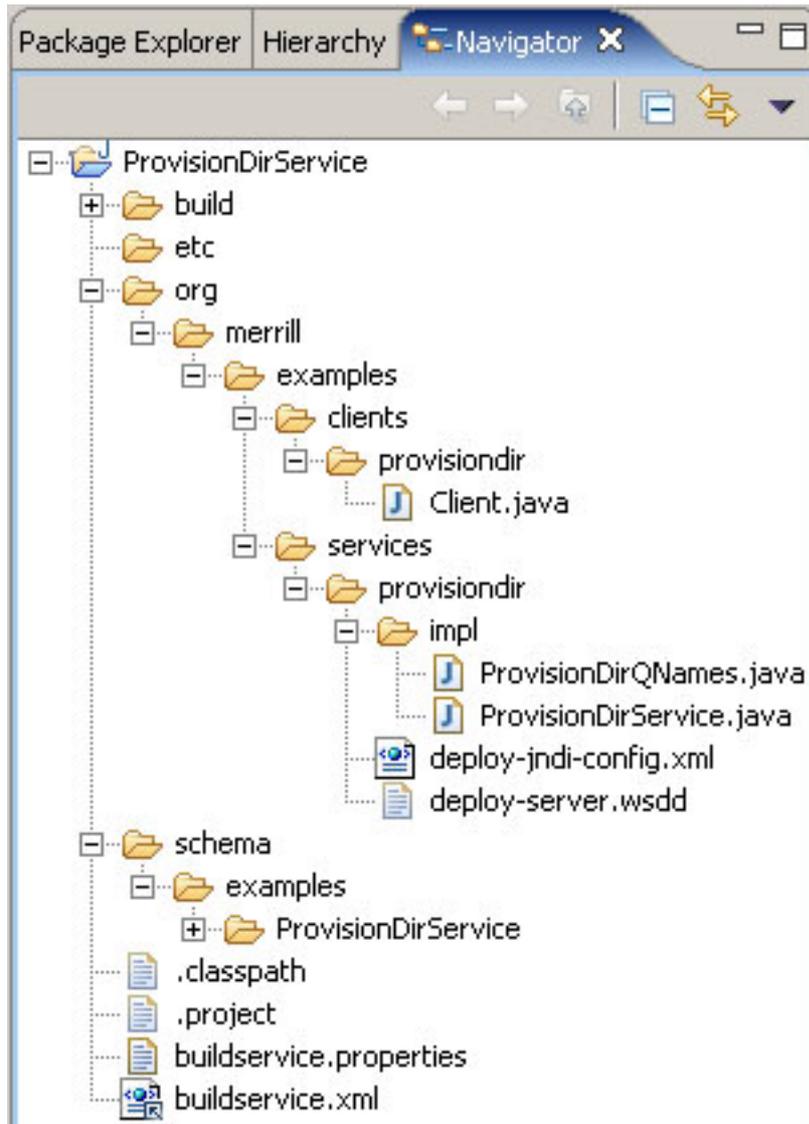
Before we can create a launch configuration for this `buildservice.xml` buildfile, we need to create a `buildservice.properties` file that contains the `name=value` properties specific to this service and needed to guide the various build tasks through the .GAR-creation/deployment process. This `buildservice.properties` file should be added to the project's root directory. The source for this file is:

```
package=com.merrill.examples.services.provisiondir
interface.name=ProvisionDir
package.dir=org/merrill/examples/services/provisiondir
schema.path=examples/ProvisionDirService
service.name=ProvisionDirService
gar.filename=org_merrill_examples_provisiondir

build.gar=/dev/GTK/etc/build.xml
build.packages=/Dev/GTK/share/globus_wsrf_common/build-packages.xml
build.tomcat=/Dev/GTK/share/globus_wsrf_common/tomcat/tomcat.xml
gar.name=/Dev/eclipse/workspace/ProvisionDirService/org_merrill_examples_provisiondir
tomcat.dir=/Dev/jakarta-tomcat-5.0.28
```

The package explorer should now resemble Figure 17:

Figure 17. Updated package explorer



Create the launch configuration

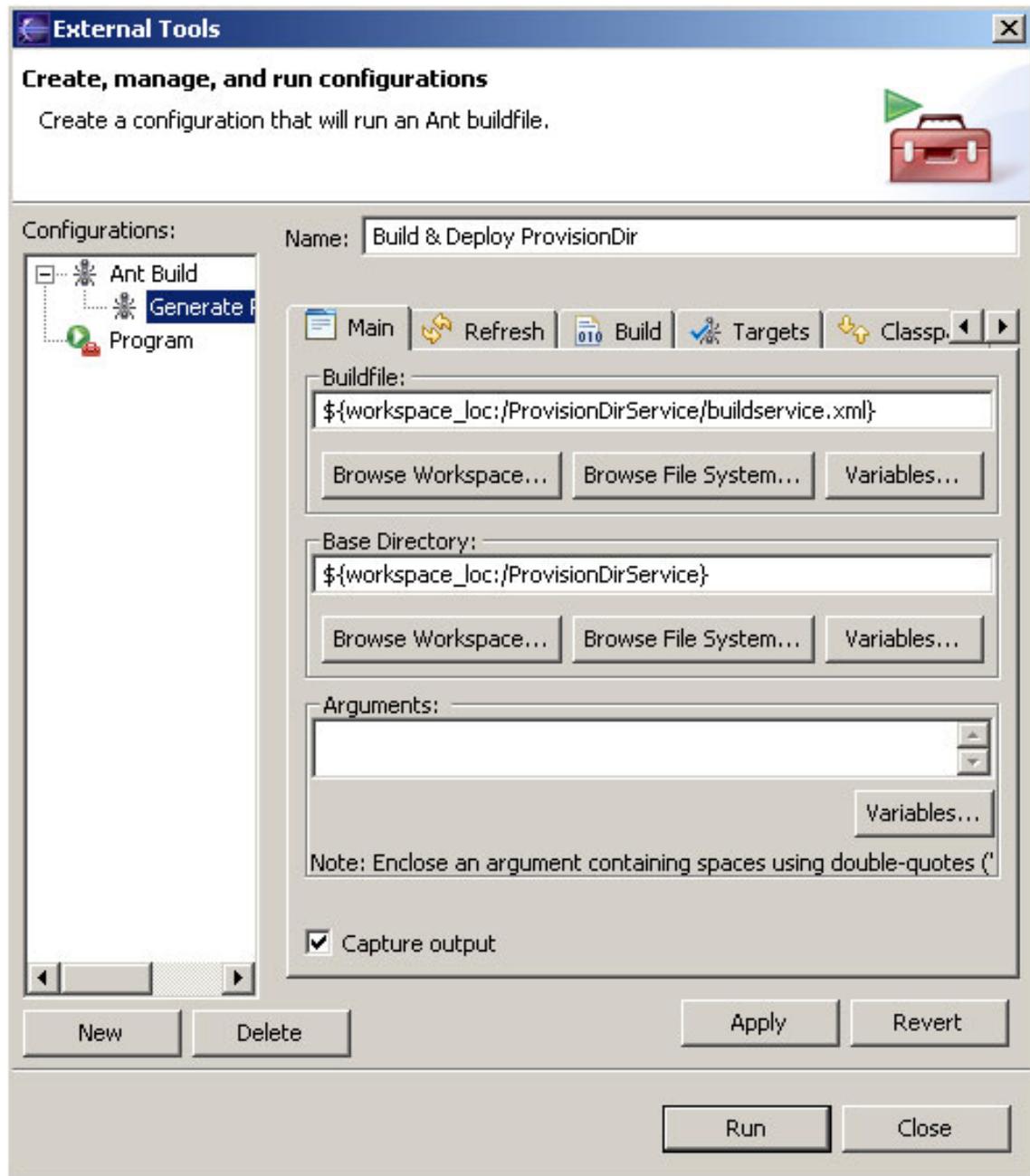
With this Ant buildfile now linked into the project and a corresponding `.properties` file created, we can now create a launch configuration that will create the `.GAR` for this service (as well as flesh out the other necessary artifacts). To do this, right-click on the `buildservice.xml` file and select **Run > External Tools...** as shown in Figure 18.

Figure 18. Run > External Tools option



If necessary, select "Ant Build" from the Configurations pane and click **New**. then select the ProvisionDirService buildservice.xml and rename it "Build and Deploy ProvisionDir" and enter the project root, `${workspace_loc:/ProvisionDirService}` as the Base Directory field in the Main tab, as shown in Figure 19.

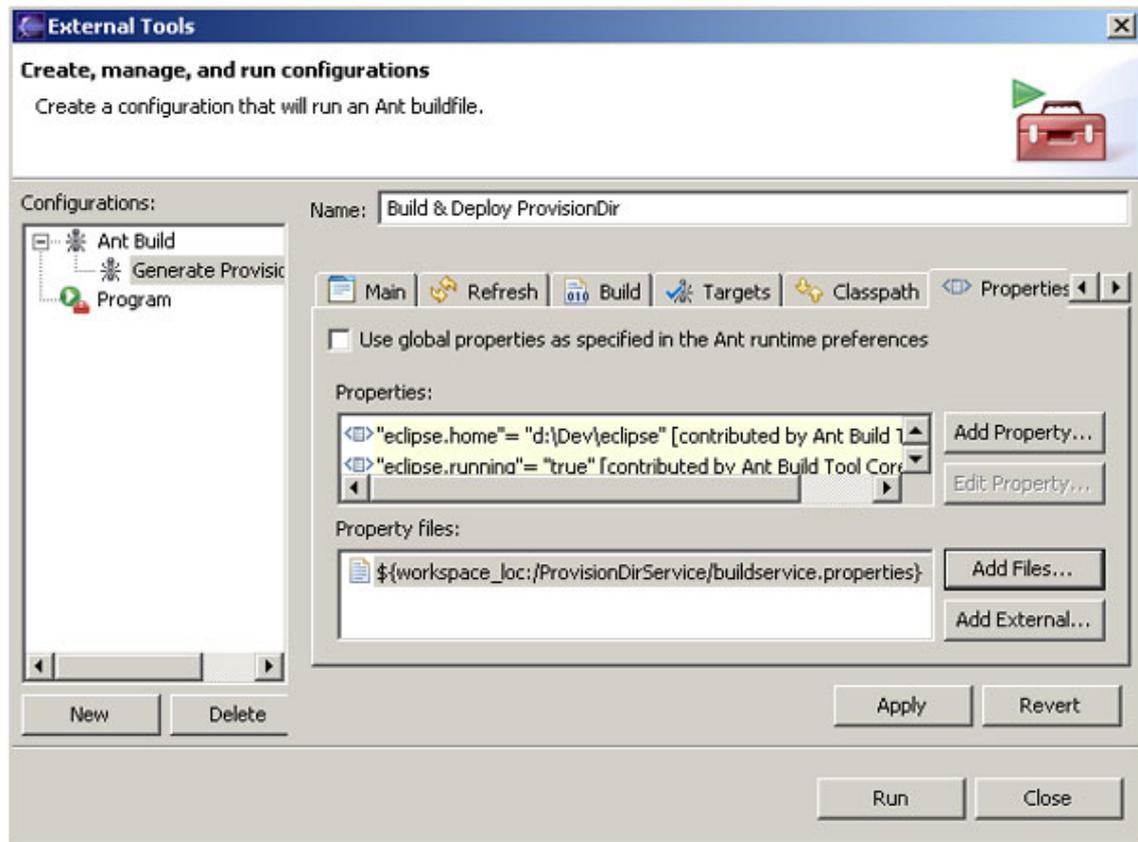
Figure 19. External Tools dialog



Launch configuration properties

In the Properties tab, deselect the "Use global properties..." checkbox and click **Add > Files...** to add the `buildservice.properties` file from the root of the project, as shown in Figure 20.

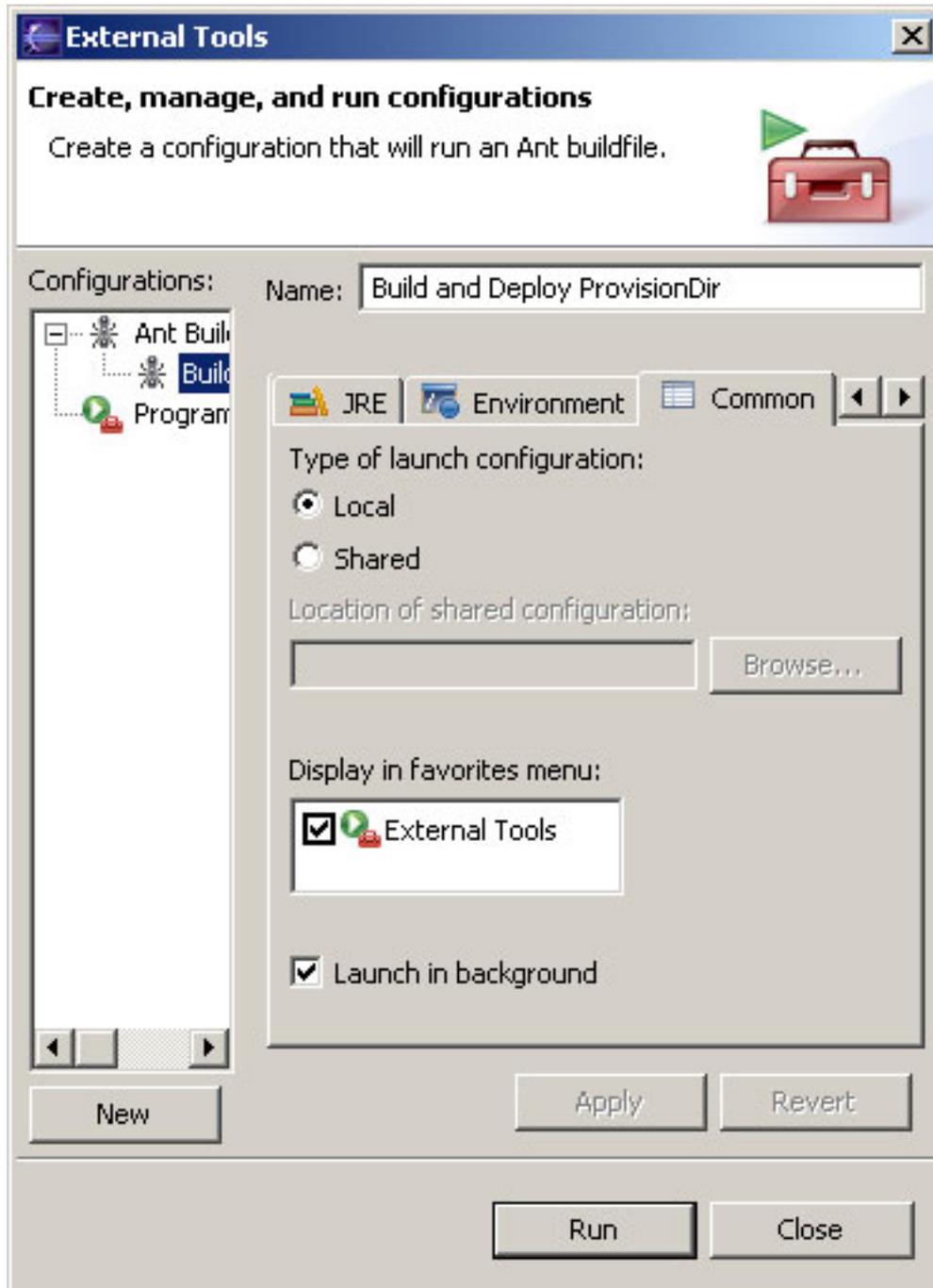
Figure 20. Properties tab of the External Tools dialog



Adding the launch configuration to External Tools toolbar button

Finally, in the Common tab, select the External Tools checkbox to display this configuration in the favorites drop-down menu when clicking on the External Tools icon in the toolbar (see Figure 21). Close (and save) the launch configuration.

Figure 21. Adding the Launch Configuration to External Tools toolbar button



Click **Close**.

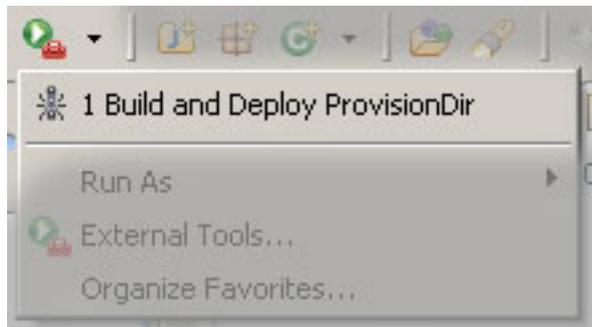
Section 6. Generate stubs and deploy the grid service

Building and deploying

Now that we've completed the project setup, we can begin the iterative process of building/running/debugging/editing.

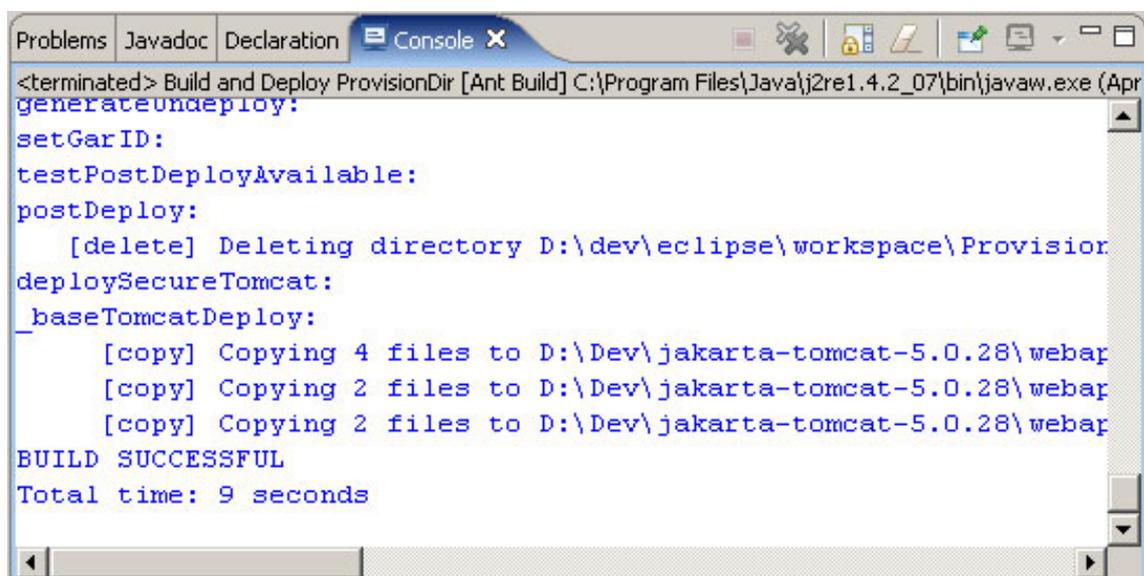
We can build and deploy our grid service with one click of the mouse by clicking the **External Tools icon > Build and Deploy ProvisionDir** launch configuration from the main toolbar (see Figure 22). This will kick off our Ant task and generate all of the remaining artifacts, create the service GAR, deploy the GAR into WSRF, and deploy WSRF into Tomcat.

Figure 22. Clicking the Build and Deploy ProvisionDir launch configuration from the main toolbar



You should see the output from the build process in the Console View. The Console View will show you any compilation or assembly errors (if any) or a "Build Successful," as shown in Figure 23.

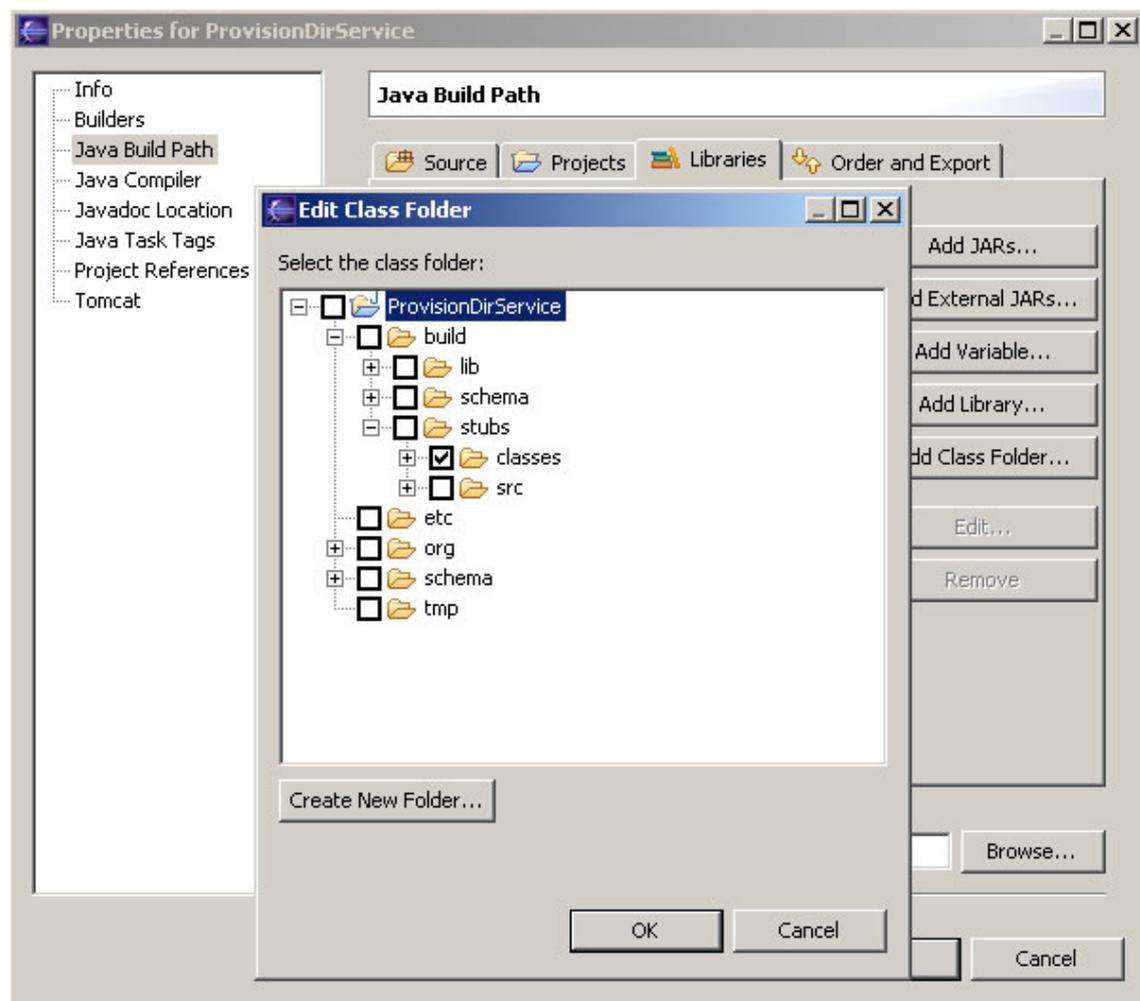
Figure 23. Build Successful message



Updating the Build Path to Include Stub Classes

Once we've generated our stubs, we can configure the Libraries tab of the Java Build Path page in the project Properties to put the stub class folder on the build path. Doing this will silence the Editor View's red compile errors. Go to the project Properties, select the Java Build Path page, and click the **Add Class Folder...** button as shown in Figure 24.

Figure 24. Edit Class Folder dialog



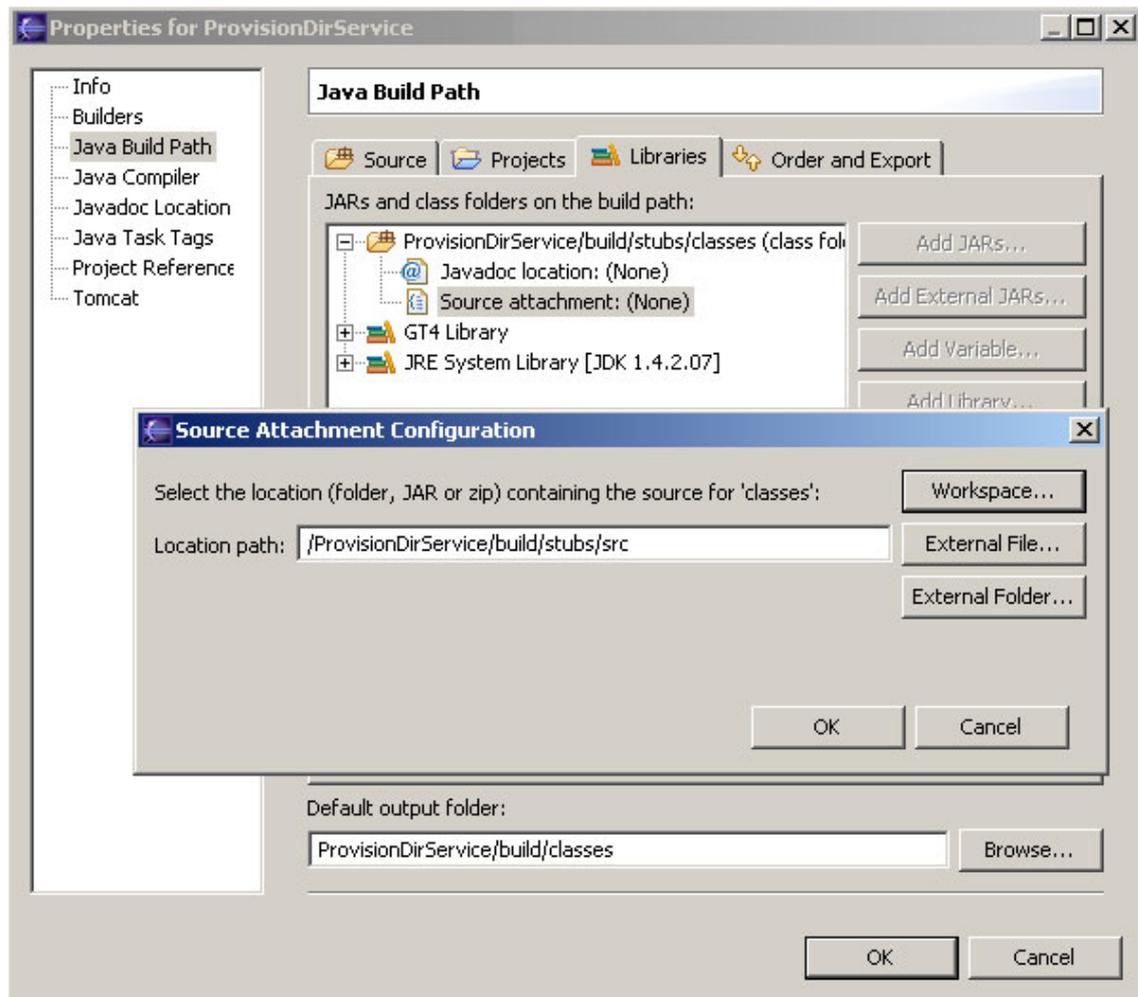
Click **OK**.

Attach the source to the stub classes

We can now attach the source to the stub classes by right-clicking the "Source Attachment" item for the classes folder and specifying the

/ProvisionDir/build/stubs/src folder in the dialog shown in Figure 25.

Figure 25. Source attachment configuration dialog



Click **OK**, then click **OK** again to finish the configuration.

Section 7. Running and Debugging using the Tomcat container

Running the Web Services container

Running (and stopping) the grid service is as easy as starting the Tomcat container by clicking the Start/Stop/Restart Tomcat toolbar buttons.

Figure 26. Tomcat toolbar buttons



This fires up Tomcat (you can monitor its output in the Console View), which runs the GTK WSRF Web application and, therefore, exposes our active service to the outside world.

In the next panel, we create a (very simple) client to test and debug our grid service.

A test client

The `Client.java` file is a simple command-line class that performs a couple of simple operations on our grid service. This file is not necessary for the development of the service; any WSRF-based client (perhaps one written in Perl using `WSRF::Lite`) can be used to invoke our service. `Client.java` should be placed in the `org/merrill/examples/clients/provisiondir` project folder. The source for this configuration file is:

```
package org.merrill.examples.clients.provisiondir;

import org.apache.axis.message.addressing.Address;
import org.apache.axis.message.addressing.EndpointReferenceType;
import org.globus.axis.util.Util;

import org.merrill.examples.provisiondir.ProvisionDirService.ListDir;
import org.merrill.examples.provisiondir.ProvisionDirService.ListDirArrayResponse;
import org.merrill.examples.provisiondir.ProvisionDirService.ChangeCwdResponse;
import org.merrill.examples.provisiondir.ProvisionDirService.ProvisionDirPortType;
import org.merrill.examples.provisiondir.ProvisionDirService.service.ProvisionDirServiceAddressingLocator;
import org.merrill.examples.services.provisiondir.impl.*;

import org.oasis.wsr.properties.GetResourcePropertyResponse;

public class Client implements ProvisionDirQNames {
    static {
        Util.registerTransport();
    }
}
```

```

    }

    public static void main(String[] args) {
        ProvisionDirServiceAddressingLocator locator =
            new ProvisionDirServiceAddressingLocator();

        try {
            String serviceURI=args[0];

            // Create endpoint reference to singleton service
            EndpointReferenceType endpoint = new EndpointReferenceType();
            endpoint.setAddress(new Address(serviceURI));
            ProvisionDirPortType provisionDir =
                locator.getProvisionDirPortTypePort(endpoint);

            // Get PortType
            provisionDir = locator.getProvisionDirPortTypePort(endpoint);

            // Access resource properties (get CWD)
            GetResourcePropertyResponse cwd = provisionDir.getResourcePro
perty(RP_CWD);

            System.out.println("\nCurrent Working Directory (Cwd RP): " +
                cwd.get_any()[0].getValue());

            // Perform a listing
            ListDirArrayResponse listings = provisionDir.listDir(new Lis
tDir());

            String[] array = listings.getA();
            System.out.println("\nListing:");
            for (int i = 0; i < array.length; i++) {
                System.out.println("\t" + array[i]);
            }

            // change the current working directory
            provisionDir.changeCwd("/temp");

            // Access resource properties (get CWD)
            cwd = provisionDir.getResourceProperty(RP_CWD);
            System.out.println("\nCurrent Working Directory (Cwd RP): " +
                cwd.get_any()[0].getValue());

            // Perform a listing
            listings = provisionDir.listDir(new ListDir());
            array = listings.getA();
            System.out.println("\nListing:");
            for (int i = 0; i < array.length; i++) {
                System.out.println("\t" + array[i]);
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

The operation of the client is pretty straightforward:

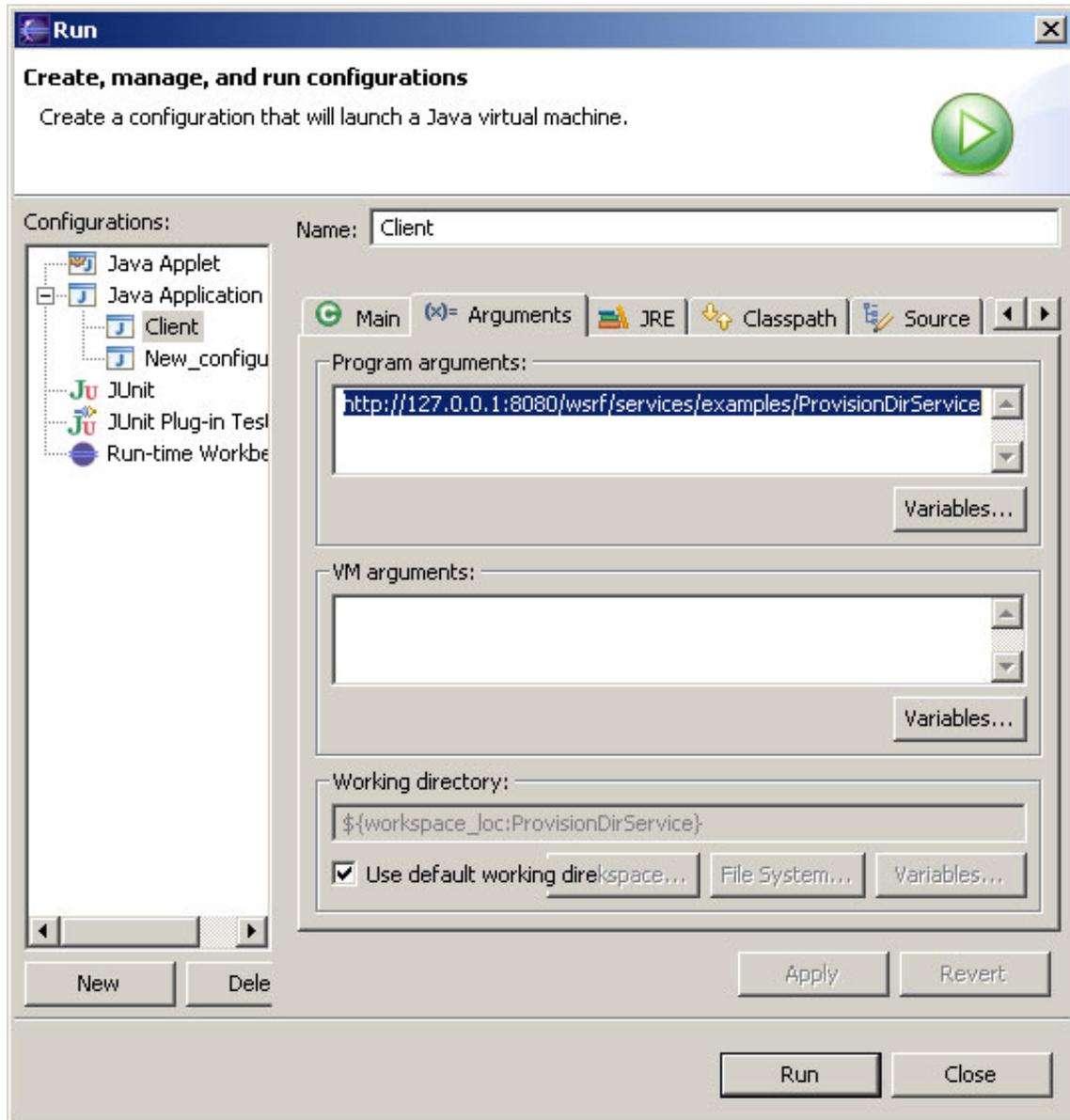
- Using the generated locator class, the client creates a proxy stub to the remote ProvisionDirService.
- The client retrieves the Cwd resource property and displays it.

- The client invokes the `listDir()` operation and retrieves the wrapper datastructure that contains the list of strings denoting the contents of the current working directory.
 - The client invokes the `changeCwd()` operation to change the resource's current directory.
 - The client retrieves the new `Cwd` resource property and displays it again.
 - The client invokes the `listDir()` again and displays the contents of the new current working directory.
-

Executing the test client

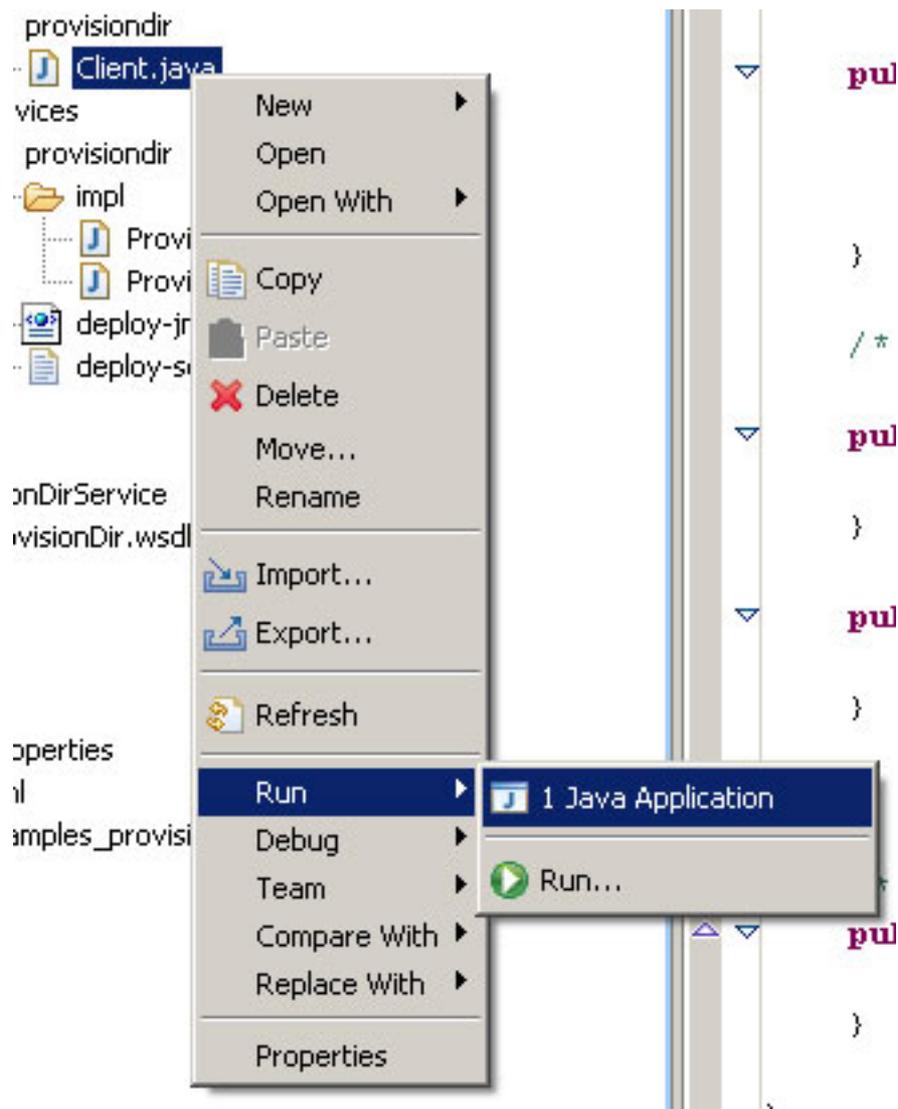
To test the client, simply right-click the `Client.java` file and select **Run > Run...** from the pop-up menu (See Figure 27). In the Run dialog that is displayed, select the Arguments tab and enter `http://127.0.0.1:8080/wsrf/services/examples/ProvisionDirService` in the Program Arguments: textbox.

Figure 27. Run dialog



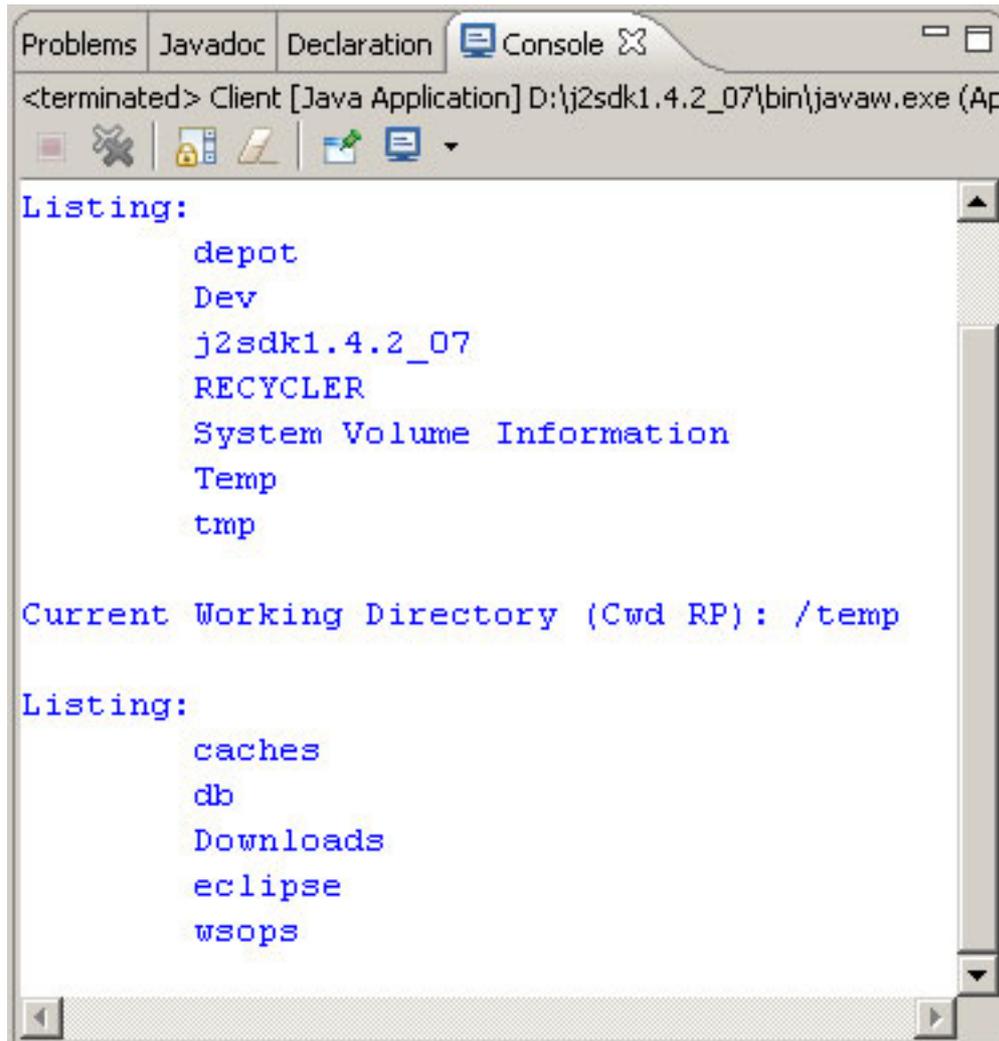
Click Close to save the run configuration for this file. Once this is done, we can repeatedly run the client application by simply right-clicking the `Client.java` file and selecting **Run > Java Application**:

Figure 28. Run Java Application



The output from the client application shows up in the Console view (see Figure 29).

Figure 29. Console view



The screenshot shows the Eclipse IDE's Console window. The title bar includes tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The console output shows the following text:

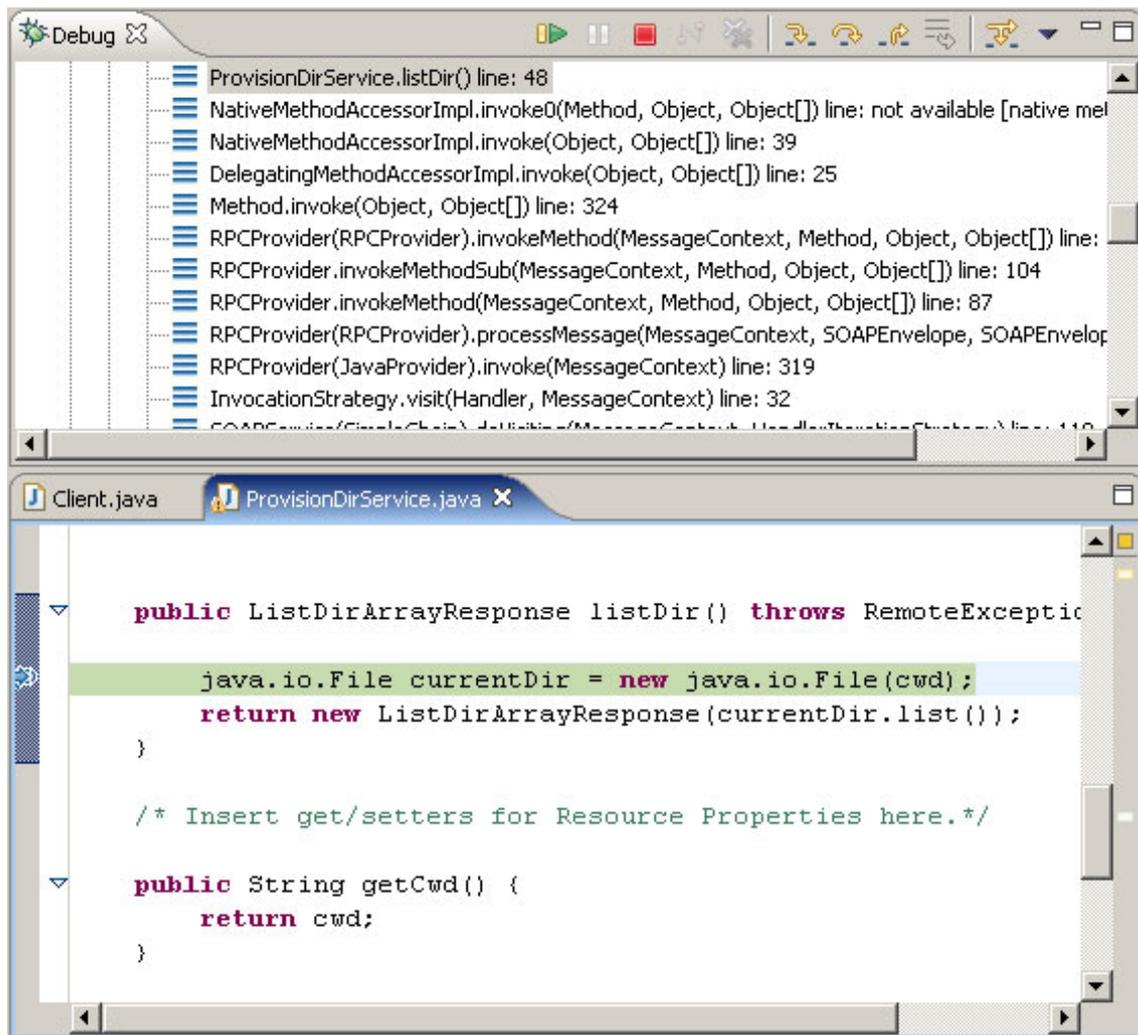
```
<terminated> Client [Java Application] D:\j2sdk1.4.2_07\bin\javaw.exe (Ap  
Listing:  
    depot  
    Dev  
    j2sdk1.4.2_07  
    RECYCLER  
    System Volume Information  
    Temp  
    tmp  
  
Current Working Directory (Cwd RP): /temp  
  
Listing:  
    caches  
    db  
    Downloads  
    eclipse  
    wsops
```

(Note: Every time we rebuild/deploy the grid service using the Launch Configuration, right-click the project and select Refresh from the pop-up menu to re-compile the client `.class` bytecode because it gets cleaned up during the rebuild/deploy process.)

Debugging in Tomcat

Go ahead and set a breakpoint in `ProvisionDirService.listdir()` and rerun the same client program. The Eclipse debugger traps the breakpoint and switches into the Debug Perspective.

Figure 30. Eclipse Debugger



Making changes

Now to change our server code, we just update our code and save the file. And that's it. It isn't necessary to build or deploy the GAR or restart Tomcat.

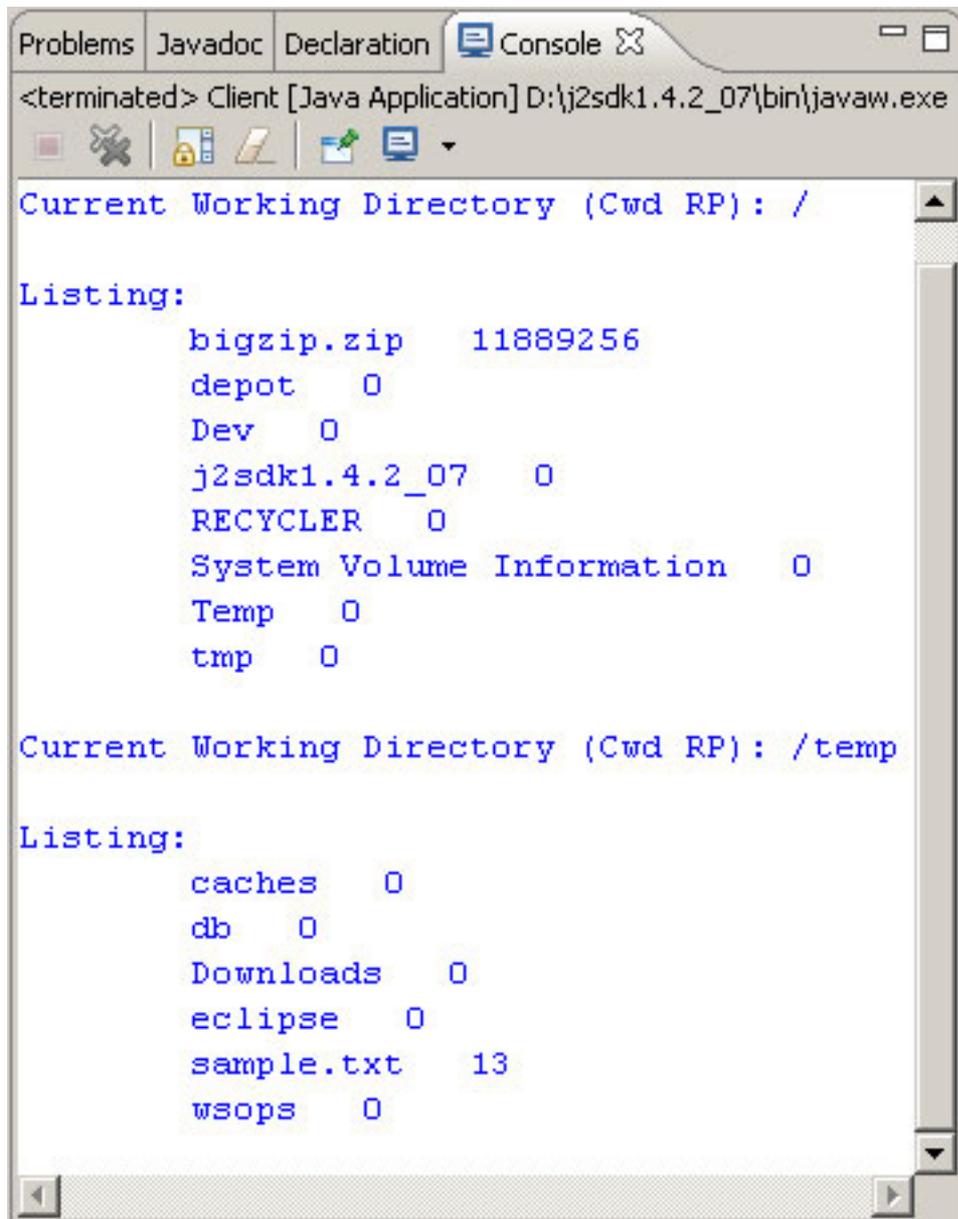
For example, change the `listDir()` method to read:

```
public ListDirArrayResponse listDir() throws RemoteException {

    java.io.File currentDir = new java.io.File(cwd);
    String[] listings = currentDir.list();
    for (int i = 0; i < listings.length; i++) {
        java.io.File listing = new java.io.File(currentDir, list
ings[i]);
        listings[i] = listings[i] + " " + listing.length();
    }
    return new ListDirArrayResponse(listings);
}
```

Save the file. Tomcat will already be running the newly updated grid service logic. (No need to even re-run the launch configuration or restart Tomcat.) To test it, re-run the client with a click of the mouse, and you'll see output with file sizes now (See Figure 31).

Figure 31. Updated console



```
<terminated> Client [Java Application] D:\j2sdk1.4.2_07\bin\javaw.exe
Current Working Directory (Cwd RP): /
Listing:
    bigzip.zip    11889256
    depot        0
    Dev          0
    j2sdk1.4.2_07  0
    RECYCLER     0
    System Volume Information  0
    Temp         0
    tmp          0

Current Working Directory (Cwd RP): /temp
Listing:
    caches      0
    db          0
    Downloads   0
    eclipse     0
    sample.txt  13
    wsops       0
```

Note: If any modifications change the service interface (say, if the WSDL is updated), rebuild using the "Build and Deploy ProvisionDir" launch configuration and restart Tomcat. (Two additional mouse clicks.)

Section 8. Summary

Summary

This tutorial showed how the development of a GT4 grid service can be vastly simplified by using the Eclipse Java IDE to coordinate the iterative process of writing, deploying, running, and debugging. We gave a step-by-step walkthrough of how to create, configure, and use the `ProvisionDirService` Java Project, focusing on the following areas:

- Setting up Eclipse, GT4, Tomcat, and the other necessary plug-ins and tools
- Creating and configuring the Eclipse project in preparation for the source files
- Adding the source files (and reviewing their major features)
- Creating the build/deploy Launch Configuration that orchestrates the automatic generation of the remaining artifacts, assembling the GAR, and deploying the grid service into the Web services container
- Using the Launch Configuration to generate and deploy the grid service
- Running and debugging the grid service in the Tomcat container

The significant thing to keep in mind is that, once the project is configured correctly and the service is running inside Tomcat, any changes made to the service implementation will be immediately reflected in the running grid service. That convenience -- in addition to the fact that you never have to leave the Eclipse interface -- makes for a winning combination among Eclipse, Tomcat, and GT4.

Resources

This tutorial draws upon a variety of different tools and technologies. Here are some resources to get started with.

Source files used for the `ProvisionDirService`:

- `ProvisionDir.wsdl` -- The `ProvisionDir` service interface
- `ProvisionDirQNames.java` -- A convenient interface class containing the `QName` URI/namespace constants that our service (and client) classes implement
- `ProvisionDirService.java` -- The service implementation that provides the core functionality for exposing local directory information
- `deploy-server.wsdd` -- The WSDD configuration file that tells the Web service container (Tomcat) how to publish the Web service
- `deploy-jndi-config.xml` -- The JNDI deploy file that enables the GT4 WSRF implementation to locate the resource-home for this service

- buildservice.xml -- The "master" buildfile that builds and deploys the grid service into the Tomcat container
- Client.java -- A simple client test program

◦ Download the [gr-eclipsesrc.zip](#) for this tutorial.

Some good starting places (with plenty of further jumping-off points):

WSRF and related specifications:

- The [WSRF specification working drafts](#) maintained by OASIS
- The four-part " [Understanding WSRF](#)" developerWorks tutorial
- The Globus Alliance [overview of WSRF](http://www.globus.org/wsrfl/) (<http://www.globus.org/wsrfl/>)

[Globus Toolkit V4](http://www-unix.globus.org/toolkit/) (<http://www-unix.globus.org/toolkit/>) and [The Globus Alliance](http://www.globus.org/) (<http://www.globus.org/>) :

- [Globus Toolkit 4 Programmer's Tutorial](http://gdp.globus.org/gt4-tutorial/) (<http://gdp.globus.org/gt4-tutorial/>)
- developerWorks " [Globus Toolkit 4 Early Access: WSRF](#)"
- [GT4 documents and manuals](http://www-unix.globus.org/toolkit/docs/4.0/) (<http://www-unix.globus.org/toolkit/docs/4.0/>)
- [GT4 development tools](http://gsbt.sourceforge.net/) (<http://gsbt.sourceforge.net/>) , such as the up-and-coming Eclipse GT4 IDE plug-in and the globus-build-service tools

The [Eclipse Platform](http://www.eclipse.org/) (<http://www.eclipse.org/>) :

- [Eclipse project FAQ](#)
- developerWorks' " [Introducing Eclipse](#)" article

[Apache Tomcat](http://jakarta.apache.org/tomcat/) (<http://jakarta.apache.org/tomcat/>) and the Sysdeo Tomcat Plug-in:

- [Sysdeo Tomcat Plug-in](http://www.sysdeo.com/eclipse/tomcatPlugin.html) (<http://www.sysdeo.com/eclipse/tomcatPlugin.html>)
- [The Power of Three -- Eclipse, Tomcat, and Struts](http://javaboutique.internet.com/tutorials/three/) (<http://javaboutique.internet.com/tutorials/three/>)

You can also find more helpful resources in the IBM developerWorks [Grid computing zone](http://www.ibm.com/developerworks/grid/) (<http://www.ibm.com/developerworks/grid/>) .

Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like *developerWorks* to cover.

For questions about the content of this tutorial, contact the author, Duane Merrill, at .

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .