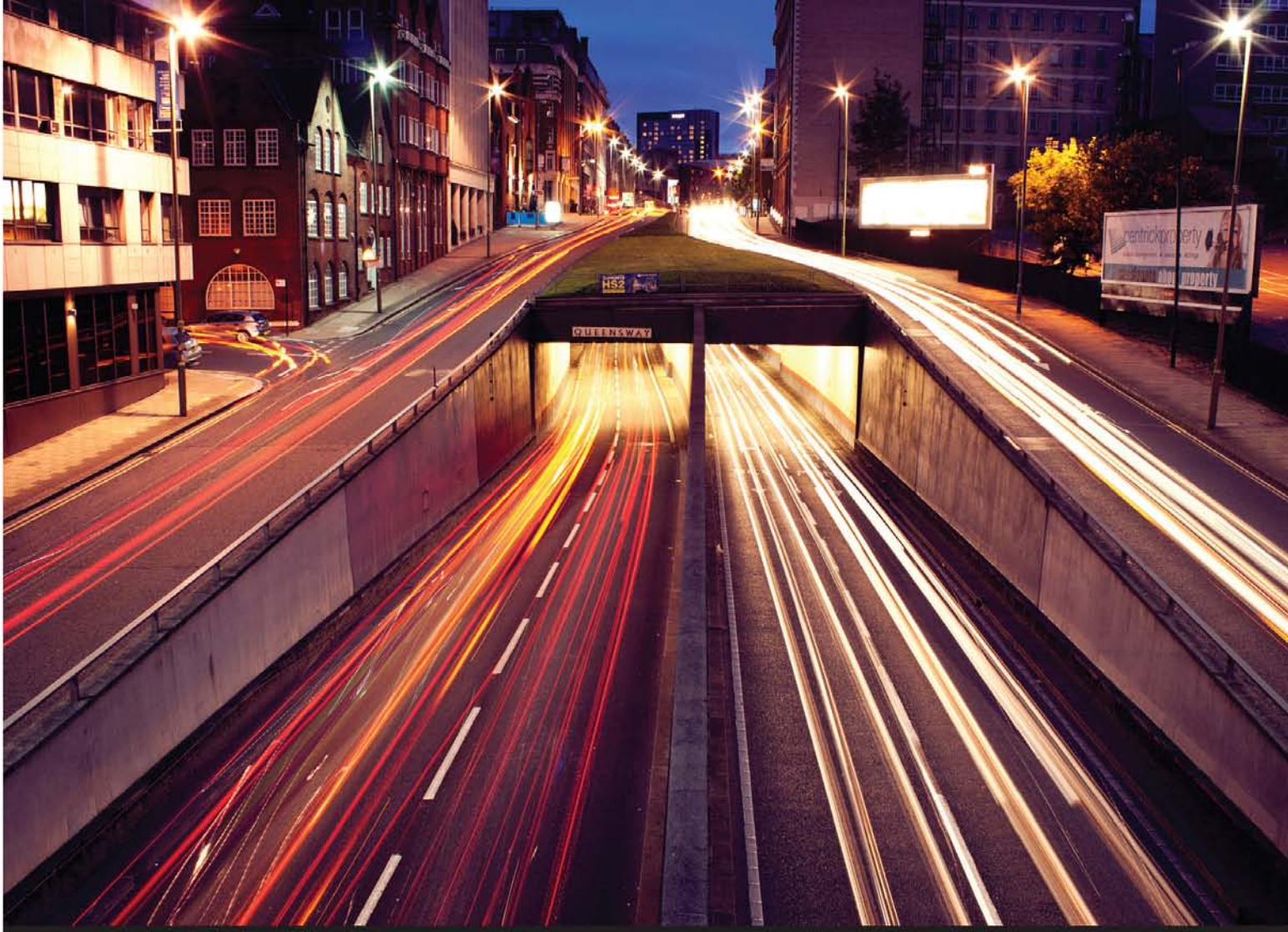


Professional Expertise Distilled

Java 7 JAX-WS Web Services

A practical, focused mini book for creating Web Services in Java 7

Deepak Vohra

[PACKT] PUBLISHING enterprise professional expertise distilled

Java 7 JAX-WS Web Services

A practical, focused mini book for creating Web Services in Java 7

Deepak Vohra

[PACKT] PUBLISHING enterprise professional expertise distilled

BIRMINGHAM - MUMBAI

Java 7 JAX-WS Web Services

First published: May 2012

Production Reference: 1140512

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84968-720-1

www.packtpub.com

Cover Image by Mark Holland (MJH767@bham.ac.uk)

Credits

Author
Deepak Vohra

Reviewer
Jobin Kuruvilla

Acquisition Editor
Andrew Duckworth

Lead Technical Editor
Susmita Panda

Technical Editors
Conrad Neil
Mehreen Shaikh

Project Coordinator
Leena Purkait

Proofreader
Joanna McMahon

Indexer
Rekha Nair

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Deepak Vohra is a consultant and a principal member of the NuBean.com software company. Deepak is a Sun Certified Java Programmer and a Web Component Developer. He has worked in the fields of XML and Java programming and J2EE for over five years. Deepak is the co-author of the book "*Pro XML Development with Java Technology*", *Apress*, and was the technical reviewer for the book "*WebLogic: The Definitive Guide*", *O'Reilly*. Deepak was also the technical reviewer for the book "*Ruby Programming for the Absolute Beginner*", *Course Technology PTR*, and the technical editor for the book "*Prototype and Scriptaculous in Action*", *Manning Publications*. Deepak is also the author of the books "*JDBC 4.0 and Oracle JDeveloper for J2EE Development*", *Packt Publishing*, "*Processing XML documents with Oracle JDeveloper 11g*", *Packt Publishing*, and "*EJB 3.0 Database Persistence with Oracle Fusion Middleware 11g*", *Packt Publishing*.

About the Reviewer

Jobin Kuruvilla is an Atlassian Consultant who is experienced in customizing Atlassian products and writing plugins for various customers. He is working with Go2group, a premier Atlassian partner, and is involved in managing Atlassian products for both big enterprises and small starter license installations.

Jobin had started his career as a Java/J2EE Developer in one of the biggest IT companies in India. After spending the initial years in the SOA world, he got hooked on to the Atlassian Toolset and was soon impressed with its power.

He is the author of "*JIRA Development Cookbook*", *Packt Publsihing*, which was published in November 2011.

Jobin also runs a website named *J Tricks – Little JIRA Tricks* (`http://www.j-tricks.com`). He has written numerous tutorials to help the developer community, who he thinks has contributed immensely to his personal development.

Dedicated to my daughter, Anna, my wife, Anu, my sister, Juby, and my parents and in-laws.

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following `@PacktEnterprise` on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface

Java 7 JAX-WS Web Services starts off with downloading and installing the Oracle GlassFish Server and NetBeans IDE. Although in this book we will be using GlassFish Server 3.1.1 and NetBeans IDE 7.0.1, later versions of GlassFish and NetBeans are also compatible. We then create a JAX-WS web service with Java 7. We shall also discuss the new `-clientjar` option in the `wsimport` tool or the `wsimport` Ant task which is used to generate JAX-WS portable artifacts from a service WSDL. Subsequently, we use the web service artifacts to invoke the web service from a web service client.

What this book covers

In *Chapter 1*, *Setting the Environment*, we set the environment for this book including installing the required software such as NetBeans IDE and the Oracle GlassFish Server.

In *Chapter 2*, *Developing a JAX-WS Web Service*, we discuss how the new `clientjar` option in the `wsimport` tool/Ant task in Java 7 is used for developing a JAX–WS web service. We use NetBeans IDE 7 and the Oracle GlassFish Server, which support Java 7.

Who this book is for

The target audience of the book is web service developers. Prior knowledge of JAX-WS web services including the `wsimport` tool and the Ant task is required. The book is also suitable for developers who want to learn about some of the new features in Java 7. If you use NetBeans-Glassfish for Java EE development you would be interested in how the new `wsimport clientjar` option may be leveraged to simplify web service development.

What you need for this book

The following software is required for the sample applications in the book:

1. Oracle GlassFish Server 3.1.1 or later
2. NetBeans IDE 7.0.1 or later

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Create a `sun-jaxws.xml` deployment descriptor in the `config` folder".

A block of code is set as follows:

```
<taskdef name="wsimport"
  classname="com.sun.tools.ws.ant.WsImport">
  <classpath refid="classpath"/>
</taskdef>
```

Any command-line input or output is written as follows:

```
asadmin start-domain domain1
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "In **Project Properties** the **Java Platform** should be set to **Java 7**".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Setting the Environment

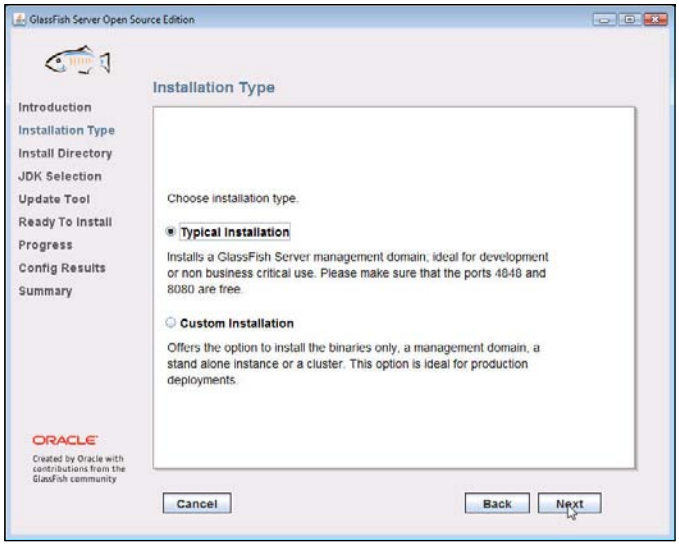
For developing a JAX-WS web service with Java 7, we shall be using NetBeans IDE 7 and Oracle GlassFish Server 3.1.1, both of which support Java 7. In this chapter we shall install the software required for this book. The Windows version of the software (`.exe` application) is used in this book. If using another OS, install the corresponding version of the software if available.

Installing the Oracle GlassFish Server

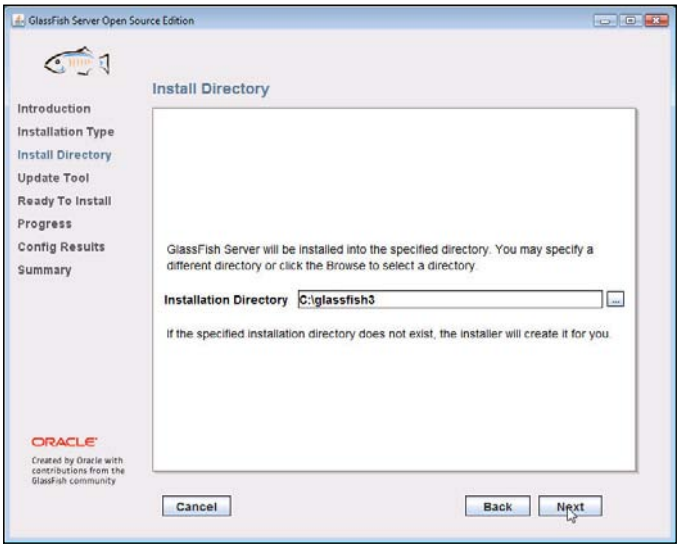
Download the Oracle GlassFish Server 3.1.1 or the latest version Open Source Edition `.exe` file from `http://www.oracle.com/technetwork/java/javaee/downloads/ogs-3-1-1-downloads-439803.html`. Double-click on the `.exe` file to start the installation. The installation gets initialized, as shown in the following screenshot:

In the **Introduction** window, click on **Next**.

In the **Installation Type** window, select **Typical Installation**. Ports **4848** and **8080** must be available. For example, if the Oracle database XE is installed and running, the database needs to be stopped. Click on **Next**:

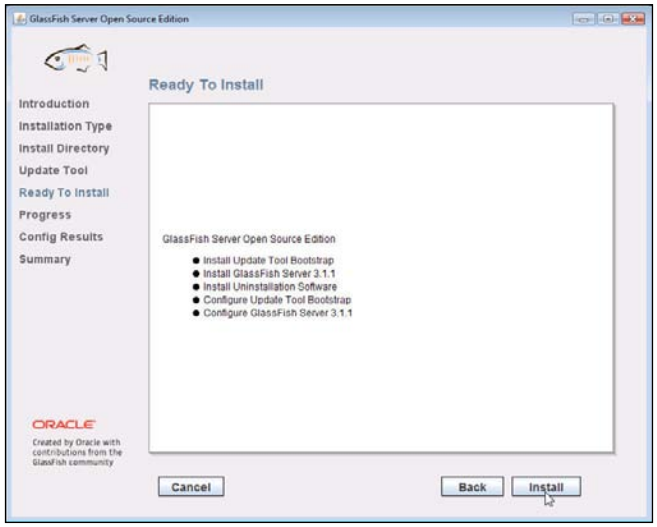


In the **Install Directory** window, specify an **Installation Directory** (**C:/glassfish3**) and click on **Next**. The directory does not need to be created prior to specifying the Installation Directory:

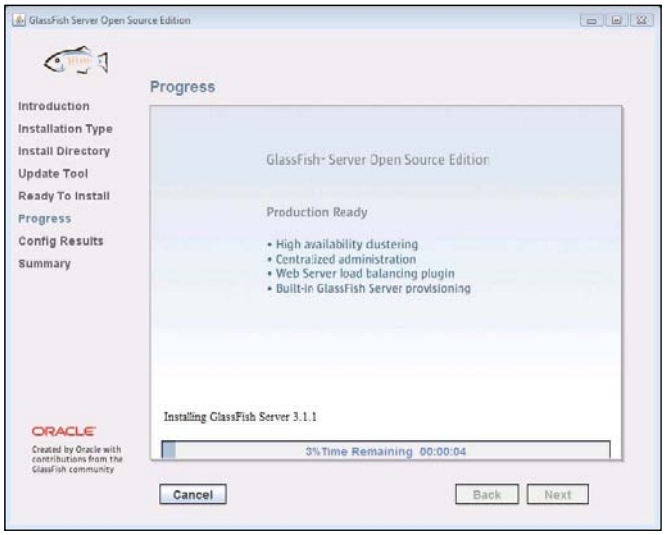


If GlassFish Server updates need to be installed, select the **Install Update Tool** checkbox. Select the **Enable Update Tool** to enable the update tool. As we won't be using the **Update Tool** for the sample application in the book, in **Update Tool** deselect **Install Update Tool** and click on **Next**.

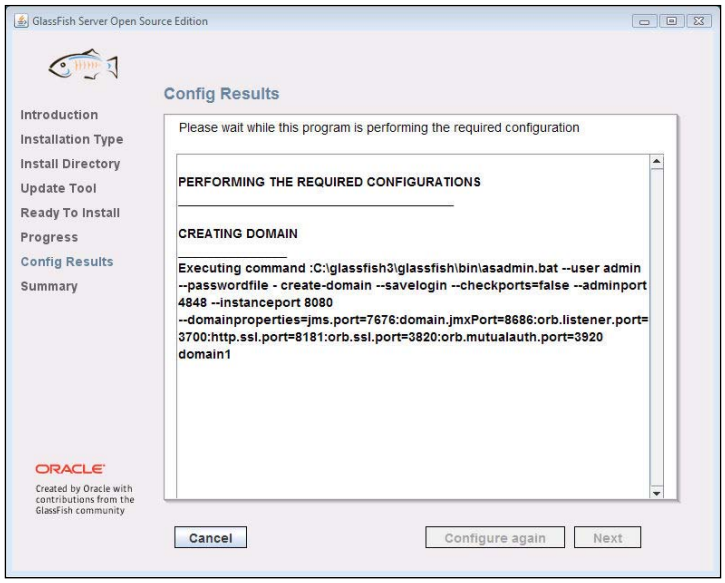
In the **Ready To Install** window, click on **Install** to install and configure the Oracle GlassFish Server Open Source Edition:



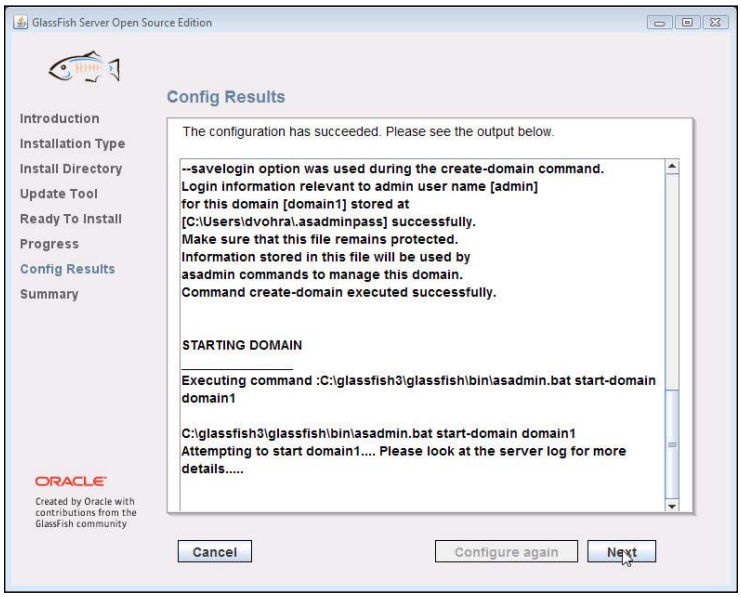
The installation of GlassFish Server 3.1.1 starts and the **Progress** bar indicates the percentage installed:



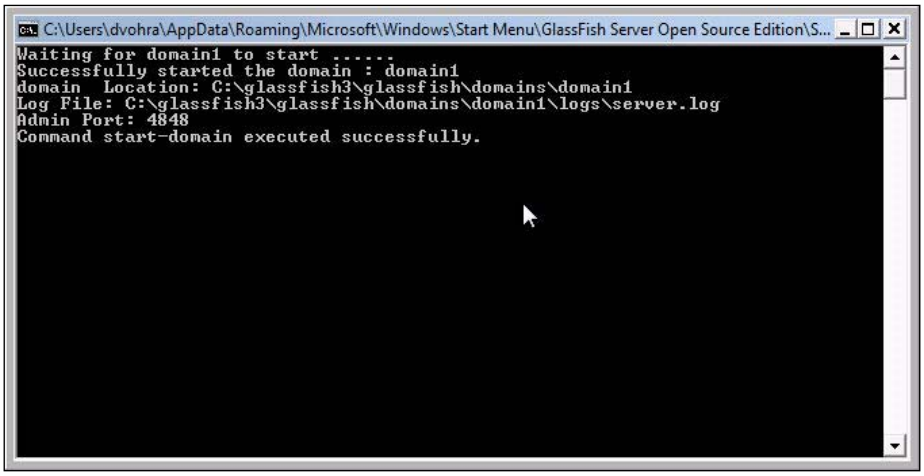
The **Config Results** window shows whether the required configuration domain (**domain1**) is created or not. If the domain creation fails with the error: "The system cannot find the path specified", running the installer with the `-j JAVA_HOME` option fixes this issue.



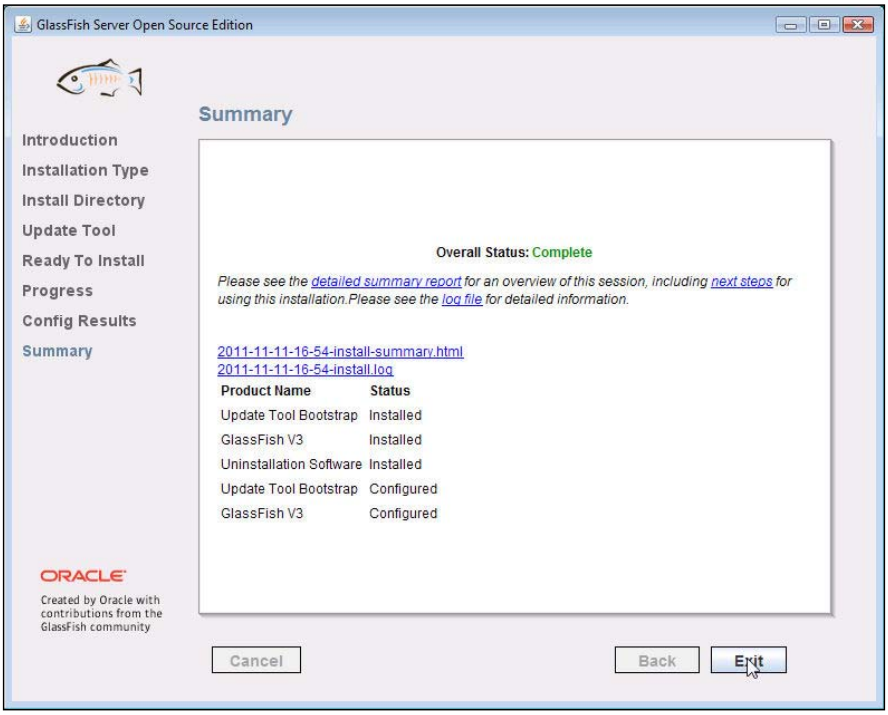
The **start-domain** command gets invoked:



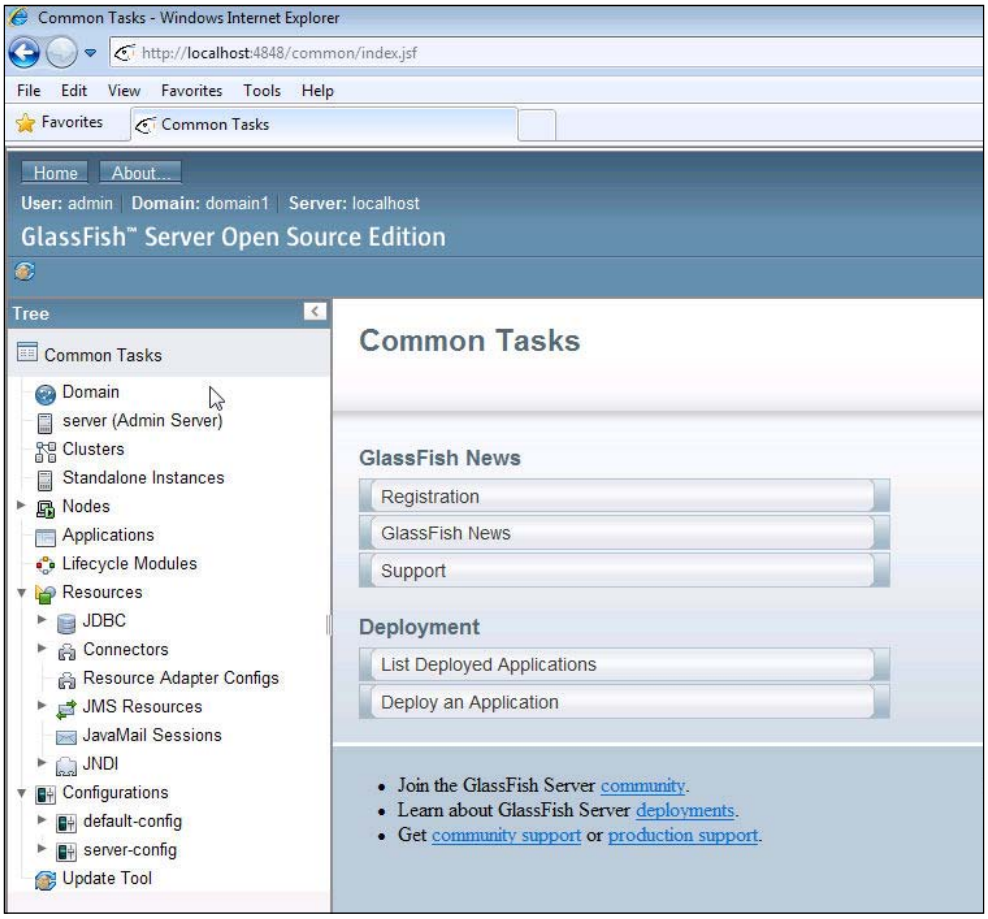
As shown in the following screenshot, the domain **domain1** starts successfully. The **Admin Port** is **4848**:



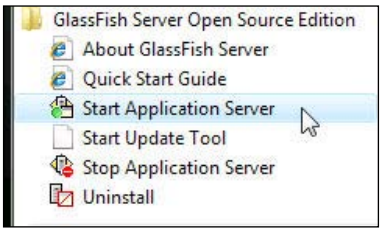
The Oracle GlassFish Server is now installed and configured. Click on **Exit**:



The GlassFish Server Administration server console may be accessed with the URL `http://localhost:4848/common/index.jsf`:



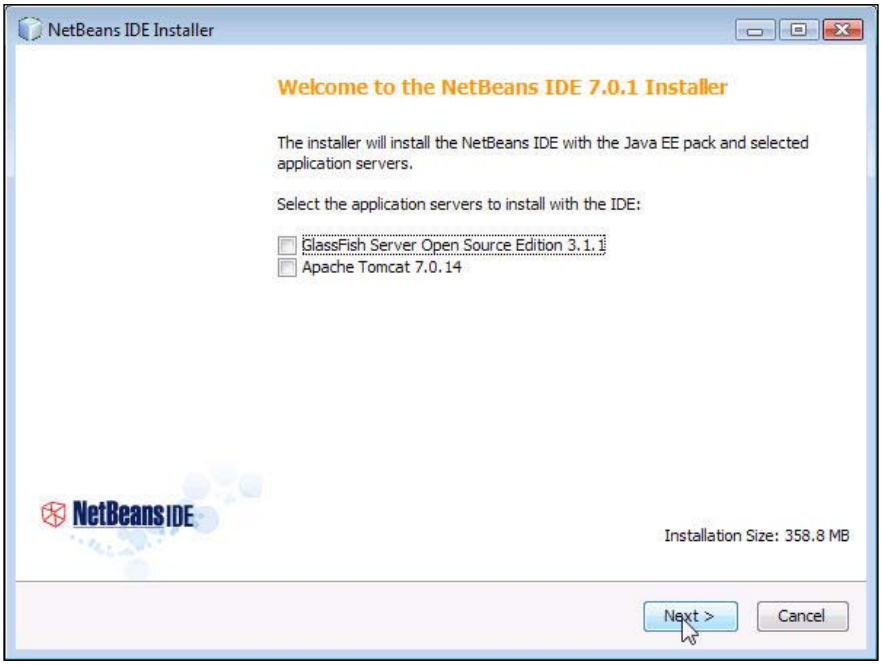
The application server may also be started/stopped from the **Windows** | **Start** program menu:



The server can be accessed at `http://localhost:8080`.

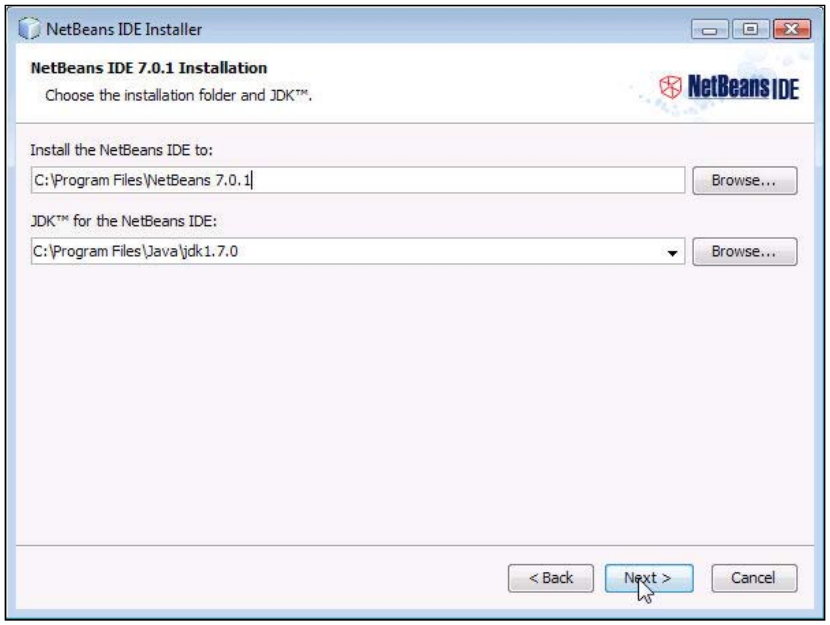
Installing NetBeans IDE 7

Download the NetBeans 7.0.1 or the latest version of Java EE Installer for Windows `.exe` file from `http://netbeans.org/downloads/index.html`. Double-click on the `.exe` application. The **NetBeans IDE 7.0.1 Installer** initializes. The NetBeans IDE Java EE version comes embedded with the GlassFish Server and Tomcat. The GlassFish Server may be installed with NetBeans IDE or separately. As we already installed the standalone version of the Oracle GlassFish Server, deselect the checkbox for the **GlassFish** Sever and also deselect the checkbox for the **Apache Tomcat** server. Click on **Next**:

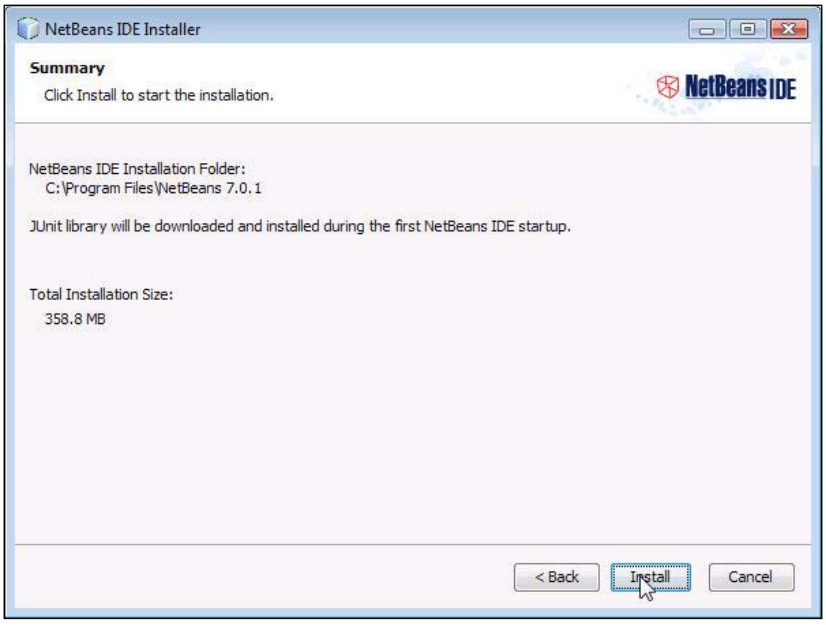


In the **License Agreement** window, select the checkbox to accept the agreement and click on **Next**. Accept the **JUnit License Agreement** and click on **Next**.

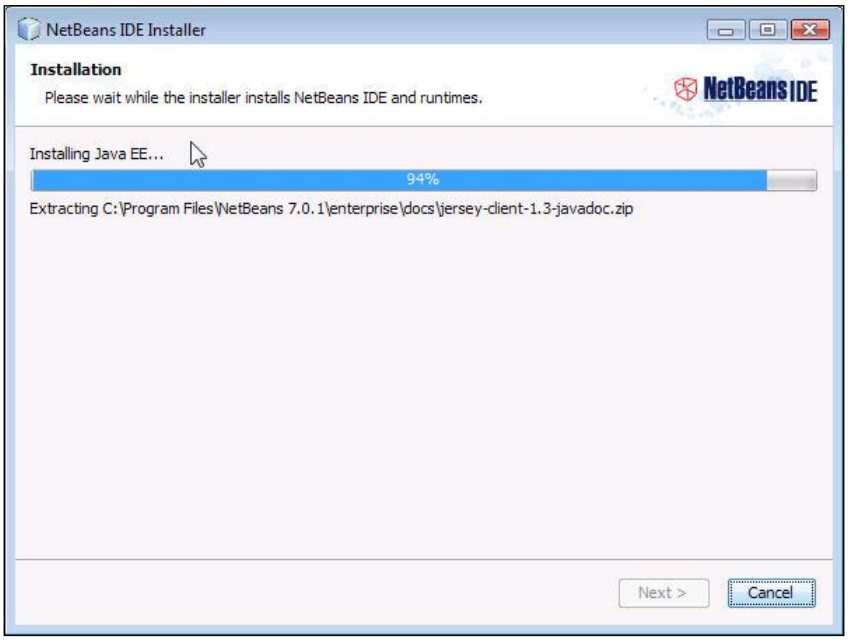
Choose the installation folder for **NetBeans IDE 7.0.1** and **JDK 1.7.0**, and click on **Next**:



The **Summary** of the installation is then displayed. Click on **Install**:



The installation of the NetBeans IDE and Java EE runtime environment begins:



When the installation completes, click on **Finish**.

Summary

In this chapter we downloaded and installed the Oracle GlassFish Server 3.1.1 and NetBeans IDE 7.0.1.

In the next chapter, we will develop a JAX-WS web service with the Java 7 `wsimport` task.

2
Developing a JAX-WS Web Service

Java 7 supports **Java API for XML Web Services** (**JAX-WS**) 2.2.4 or later. In this chapter, we shall discuss the procedure for using the new `-clientjar` option in the `wsimport` task/tool in Java 7. We require a Java IDE that supports Java 7 and a web container that also supports Java 7. We need to install the following JDK and software as discussed in the previous chapter:

- Java SE 7
- NetBeans IDE 7.0.1
- Oracle GlassFish Server 3.1.1

The sample NetBeans project is available in the downloadable sample ZIP file. This chapter has the following sections:

- What is new in Java 7 wsimport?
- Creating a NetBeans project
- Creating the implementation class
- Creating the WSDL
- Creating the deployment descriptors
- Creating a client class
- Creating the deployment targets
- Creating an Apache Ant `build` file
    - Building and deploying the service
    - Building the client
    - Running the client

- Testing the web service
- Using the `wsimport` tool from the command line

The `wsimport` tool or the `wsimport` Ant task is used to generate JAX-WS portable artifacts, which are used for invoking a web service from a web service client, from a service **Web Services Description Language** (**WSDL**). Specifically, `wsimport` generates the following artifacts:

- **Service Endpoint Interface** (**SEI**)
- Service class
- If a `wsdl:fault` is present in the WSDL, an `Exception` class
- Java classes mapped from schema types
- If a `wsdl:message` is present, asynchronous response beans

Java 6 wsimport limitation

The problem with Java 6 `wsimport` is that the JAX-WS runtime needs to fetch the WSDLs from the endpoint each time a service instance is created, which could incur a network overhead. The WSDL location is saved in the generated artifacts and the JAX-WS runtime fetches the metadata, which is useful if the endpoint policy or the service definition has changed. In the absence of the runtime fetch of the metadata, the clients would need to be regenerated if the endpoint policy or the service definition have changed. JAX-WS runtime may have access to local WSDLs using various methods such as a Service API, a `jax-ws-catalog.xml` file, or making the WSDL available at a relative local location and using the `–wsdlLocation` option when running the `wsimport` tool.

What is new in Java 7 wsimport?

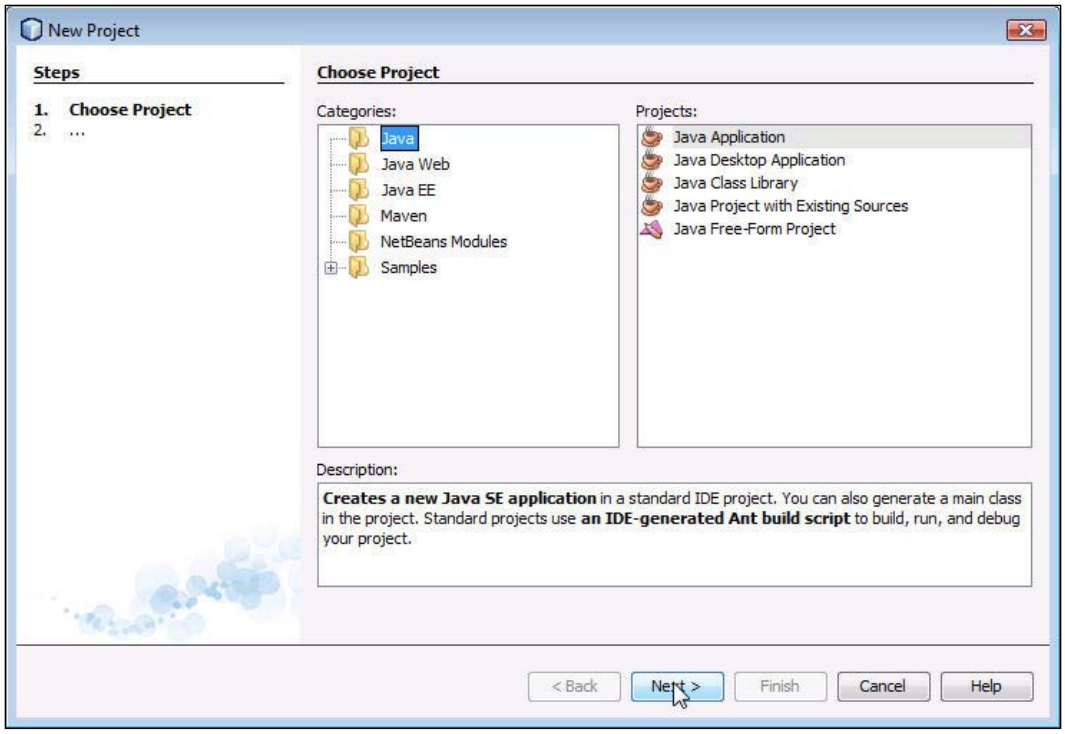
Java 7 supports Java API for XML Web Services (JAX-WS) 2.2.4, which has introduced a new (since JAX-WS 2.2.2) `wsimport` option called `-clientjar` as shown in the following sample command:

```
wsimport -clientjar wsclient.jar
  http://example.com/service/hello?WSDL
```

The `-clientjar` option fetches the WSDLs and the schemas and packages them with the generated client-side artifacts into a JAR file. By including the generated JAR file in the classpath of the web service client, there is no need to fetch the WSDLs from the endpoint each time a service instance is created, thus saving on network overhead.

Creating a NetBeans project

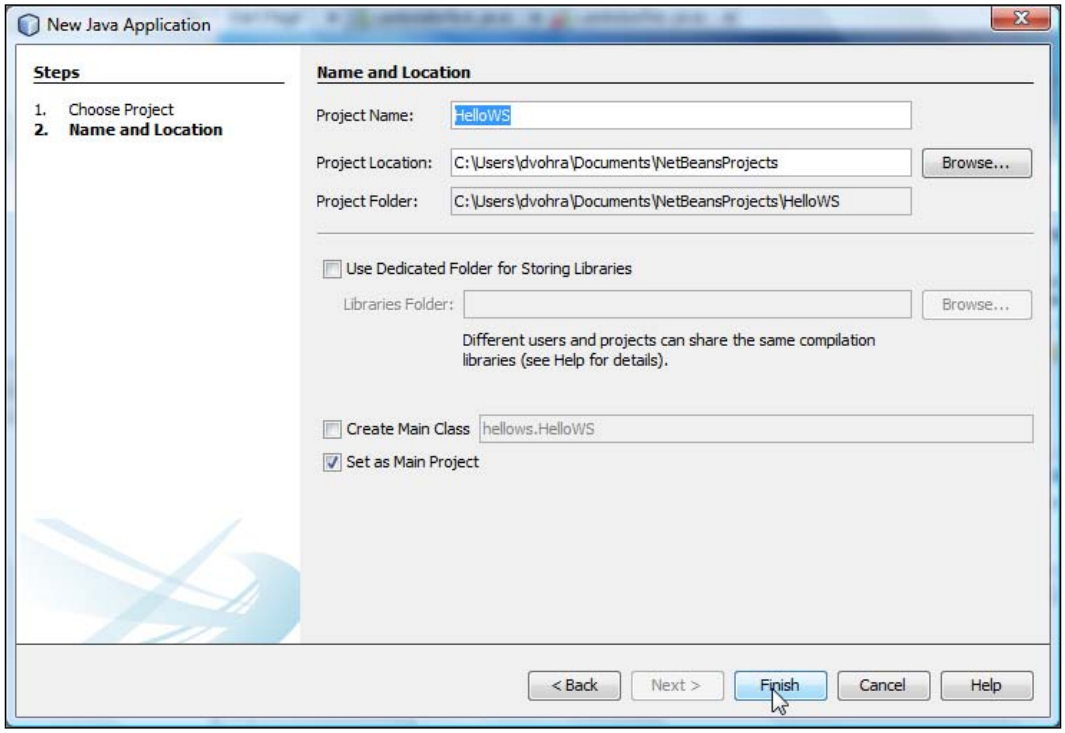
A JAX-WS web service essentially consists of a Java class annotated with the `javax.jws.WebService` annotation—the web service endpoint. A web service may optionally consist of a **Service Endpoint Interface** (**SEI**) that is implemented by the service endpoint implementation class. A web service implementation class must not be `abstract` or `final`. Business methods of the implementation class that are to be exposed as operations to a web service client must be `public`, must not be `static` or `final`, and must be annotated with the `@WebMethod` annotation. In NetBeans IDE select **File | New Project** to create a new project. In the **New Project** wizard, select **Java** in **Categories** and **Java Application** in **Projects**, and click on **Next**:



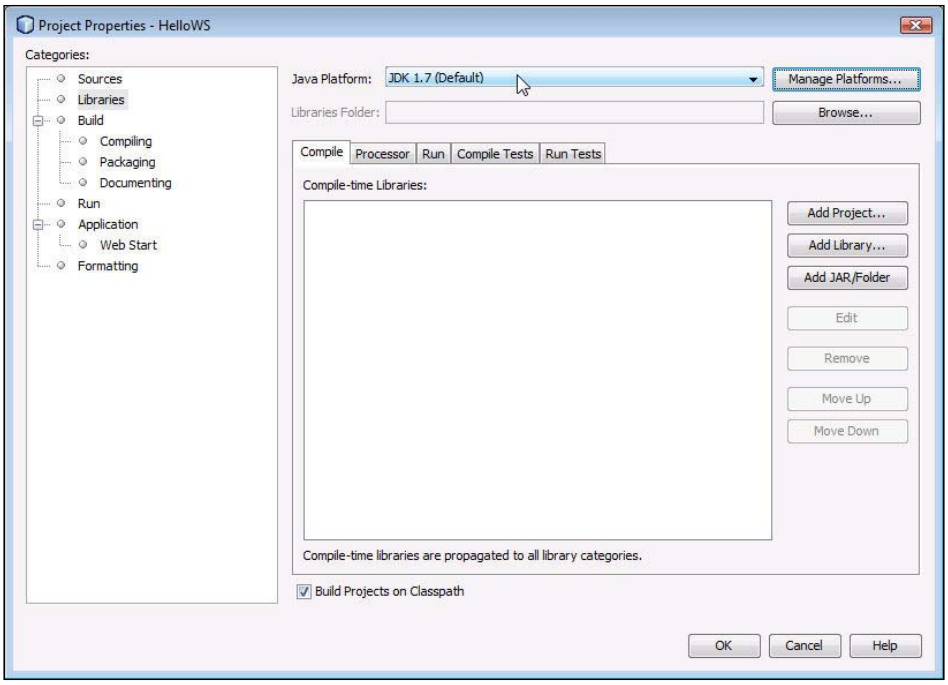
Downloading the example code

You can download the example code fles for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the fles e-mailed directly to you.

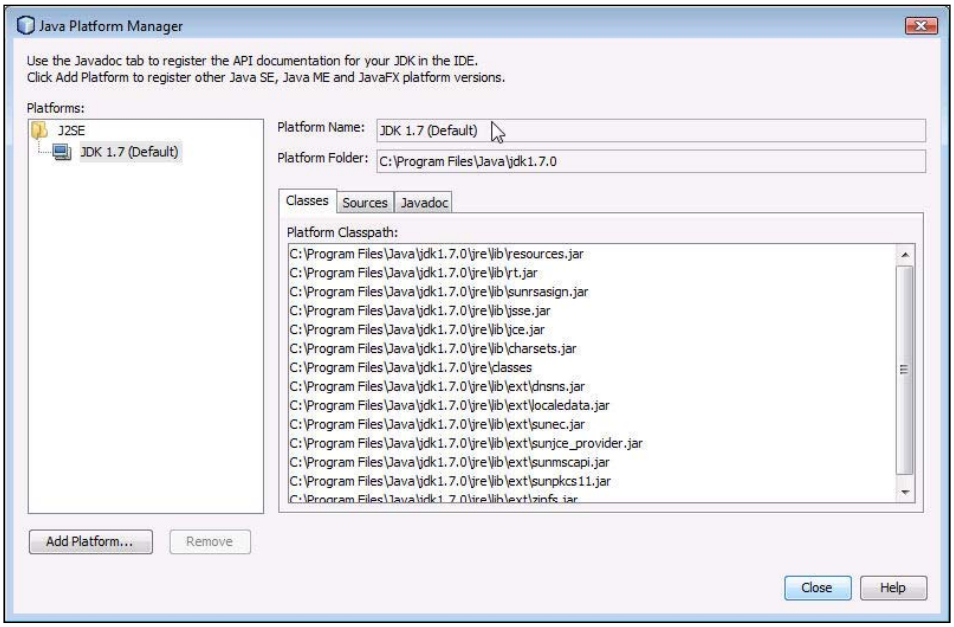
In **New Java Application** specify the **Project Name (HelloWS)** and choose the default **Project Location** and **Project Folder**. Uncheck the checkbox **Create Main Class**, and click on **Finish**:



The Java platform for the project should be set to **Java 7**. To do this, right-click on the project node and select **Properties**. In **Project Properties** the **Java Platform** should be set to **Java 7**.

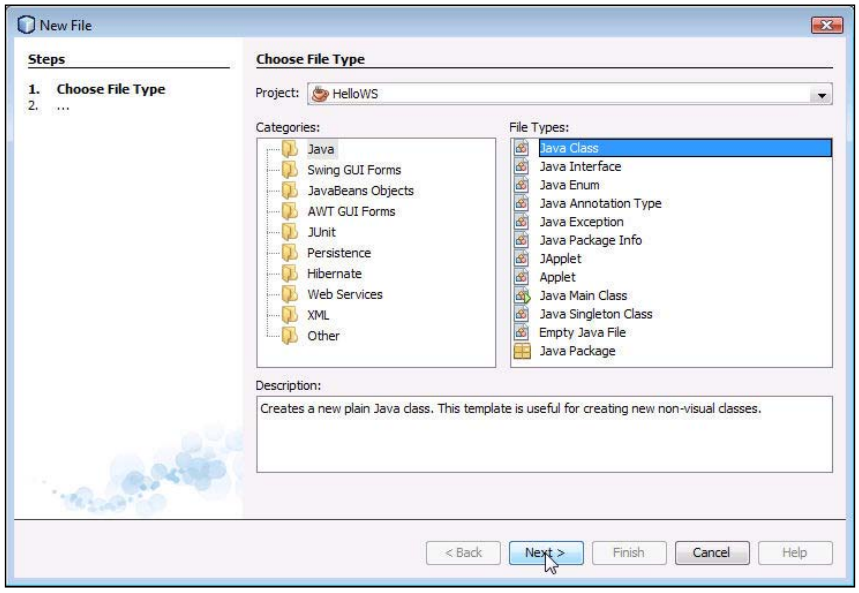


Click on **Manage Platforms**. The **JDK 1.7 Platform Classpath** shows the Java 7 JAR files in the classpath:

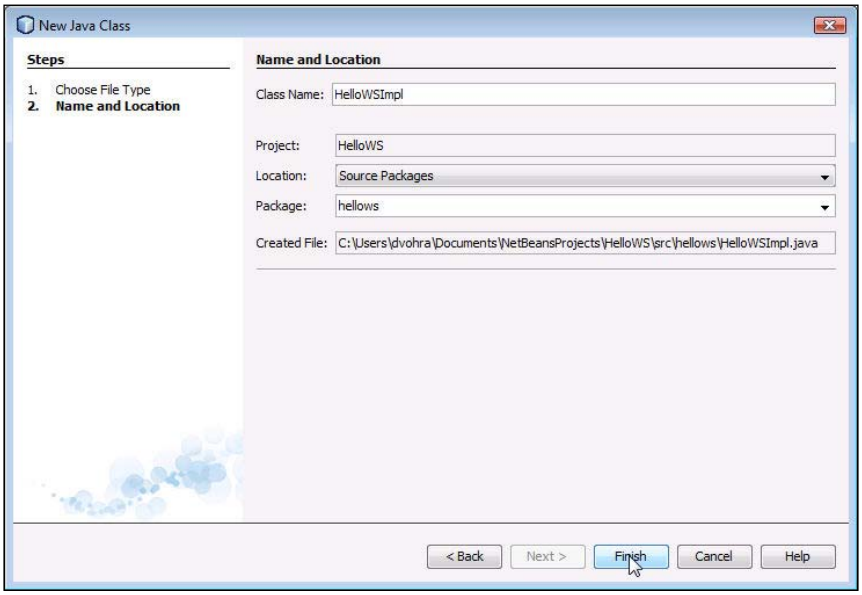


Creating the implementation class

Next, create the implementation class. Select **File | New File**. In **New File**, select Java in **Categories** and **Java Class** in **File Types**, and click on **Next**:



Specify the **Class Name** (**HelloWSImpl**), **Package** (**hellows**), and click on **Finish**:



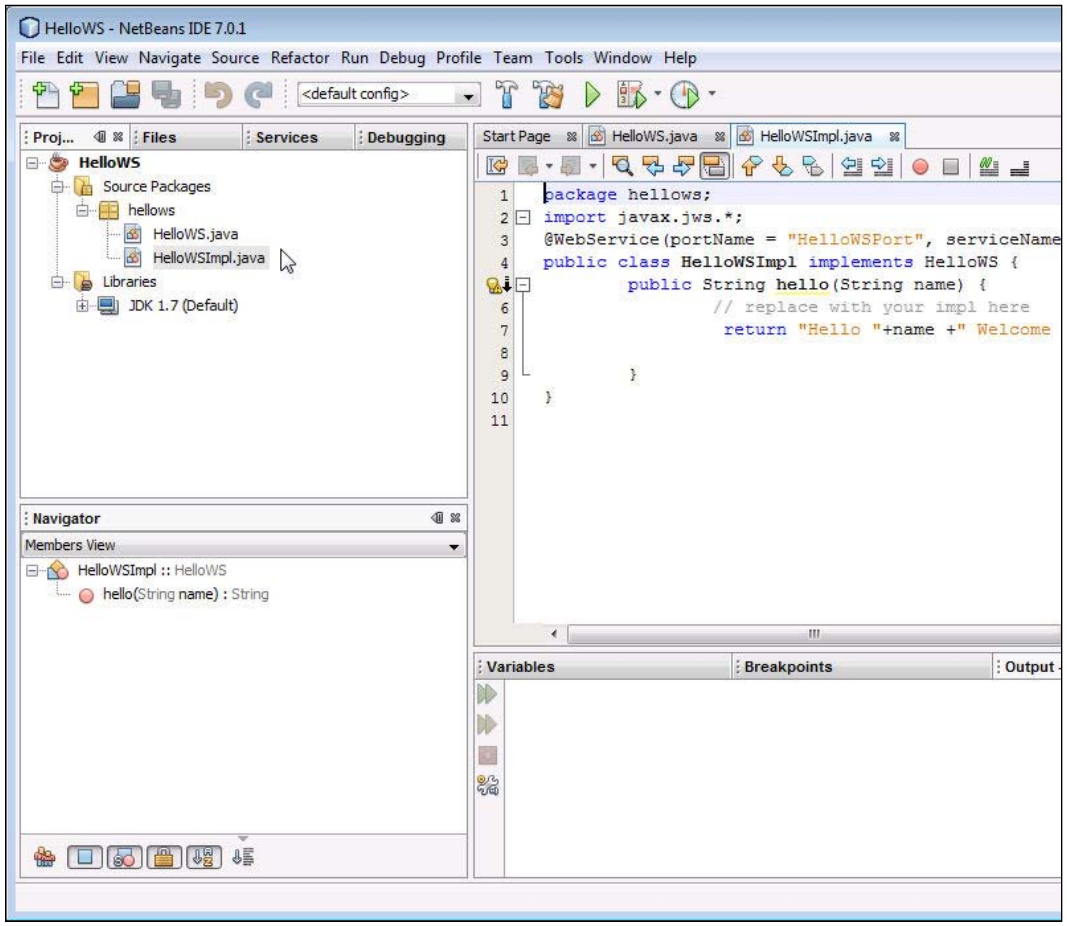
The web service implementation class `HelloWSImpl` is annotated with the `@WebService` annotation and implements the `HelloWS` interface. The implementation class contains a method `hello` that takes a `String` parameter for name and returns a Hello message containing the name. The implementation class is listed as follows:

```
package hellows;
import javax.jws.*;
@WebService(portName = "HelloWSPort", serviceName = "HelloWSService",
  targetNamespace = "http://hellows/",
  endpointInterface = "hellows.HelloWS")
public class HelloWSImpl implements HelloWS {
  public String hello(String name) {
// replace with your impl here
     return "Hello "+name +" Welcome to Web Services!";
  }
}
```

Similarly, add a Java interface for a SEI. The SEI declares `public` methods that clients may invoke on the service. The SEI is optional, as a web service implementation class implicitly defines a SEI. As we specified, an explicit SEI with an `endpointInterface` element in the `@WebService` annotation in the implementation class, we must include a SEI with `public` methods made available on the service. The service endpoint interface `HelloWS` contains the `hello` method annotated with the `@WebMethod` annotation:

```
package hellows;

import javax.jws.WebMethod;
import javax.jws.WebService;
@WebService(name = "HelloWS", targetNamespace = "http://hellows/")
  public interface HelloWS {
  @WebMethod(operationName = "hello")
  public String hello(String name);
}
```

The implementation class and the endpoint interface are shown in the following NetBeans project:



Creating the WSDL

A WSDL describes a set of business operations. The WSDL `HelloWSService.wsdl` for the sample web service defines an operation "hello" that takes an input parameter and returns a response. The `HelloWSService.wsdl` used in this chapter has elements that are discussed in the following table:

Element	Description
message	Describes input and output parameters and return values.
types	Describes the schema (`HelloWSService_metadata1.xsd`) for the XML types used in the messages.
portType	Describes the operations and associated messages. Defines abstract operations. `portType` is `HelloWS` and operation(s) are `hello`.
binding	Binding `HelloWSPortBinding` describes the protocol used to access a `portType`. Also describes the data formats for the messages defined by the `portType` element.
service	Service `HelloWSService` describes the web service and a list of ports.
port	Port `HelloWSPort` describes the location of the web service `http://localhost:8080/clientjar/hellows?wsdl` and the binding used for service access.

To learn more about WSDL refer to `http://www.w3.org/TR/wsdl`.
The `HelloWSService.wsdl` is listed as follows:

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://
hellows/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
name="HelloWSService" targetNamespace="http://hellows/">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://hellows/"
        schemaLocation="HelloWSService_metadata1.xsd">
      </xsd:import>
    </xsd:schema>
  </types>
  <message name="hello">
    <part element="tns:hello" name="parameters"></part>
  </message>
  <message name="helloResponse">
    <part element="tns:helloResponse" name="parameters"></part>
  </message>
  <portType name="HelloWS">
    <operation name="hello">
      <input message="tns:hello"></input>
      <output message="tns:helloResponse"></output>
    </operation>
  </portType>
  <binding name="HelloWSPortBinding" type="tns:HelloWS">
```

```
      <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http">
      </soap:binding>
      <operation name="hello">
        <soap:operation soapAction=""></soap:operation>
        <input>
          <soap:body use="literal"></soap:body>
        </input>
        <output>
          <soap:body use="literal"></soap:body>
        </output>
      </operation>
    </binding>
    <service name="HelloWSService">
      <port binding="tns:HelloWSPortBinding" name="HelloWSPort">
        <soap:address
          location="http://localhost:8080/clientjar/hellows?wsdl">
        </soap:address>
      </port>
    </service>
  </definitions>
```

The XML types used in the input and output messages are described in the schema for the web service, `HelloWSService_metadata1.xsd`. The `tns` namespace prefix defines the `http://hellows/` namespace, which is the `targetNamespace` of the schema. The schema has elements `hello` and `helloResponse`. The `hello` element is of type `tns:hello`, which defines an element `arg0` for the input message parameter(s). The `helloReponse` element is of type `tns:helloResponse`, which defines the return value. The schema for the element is listed as follows:

```
<xs:schema xmlns:tns="http://hellows/" xmlns:xs="http://www.
w3.org/2001/XMLSchema" targetNamespace="http://hellows/"
version="1.0">
  <xs:element name="hello" type="tns:hello"></xs:element>
  <xs:element name="helloResponse"
    type="tns:helloResponse"></xs:element>
  <xs:complexType name="hello">
    <xs:sequence>
      <xs:element minOccurs="0" name="arg0"
        type="xs:string"></xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="helloResponse">
    <xs:sequence>
      <xs:element minOccurs="0" name="return"
        type="xs:string"></xs:element>
```

```
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
```

Create a directory `config` in the NetBeans project root directory `HelloWS` and copy the WSDL and schema to the directory. When packaging the web service, the WSDL and the schema should be packaged in the `WEB-INF/wsdl` folder.

Creating the deployment descriptors

The `web.xml` delegates requests for the web service to the JAX-WS runtime using a web service listener and a web service servlet. The following listener and servlet classes are configured in `web.xml`:

```
Listener class com.sun.xml.ws.transport.
  http.servlet.WSServletContextListener
Servlet class com.sun.xml.ws.transport.
  http.servlet.WSServlet
```

The `HelloWSPort` web service servlet is mapped to URL pattern `/hellows` and has the `load-on-startup` set to `1`. The `web.xml` file is listed as follows; copy the `web.xml` file to the `config` folder:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.
com/xml/ns/javaee/web-app_2_5.xsd" xsi:schemaLocation="http://java.
sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.
xsd" id="WebApp_ID" version="2.5">
  <display-name>HelloWebService</display-name>

    <listener>
        <listener-class>com.sun.xml.ws.transport.
          http.servlet.WSServletContextListener
        </listener-class>
    </listener>
    <servlet>

        <display-name>clientjar</display-name>
        <servlet-name>HelloWSPort</servlet-name>
        <servlet-class>com.sun.xml.ws.transport.
          http.servlet.WSServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloWSPort</servlet-name>
```

```
            <url-pattern>/hellows</url-pattern>
        </servlet-mapping>
        <session-config>
            <session-timeout>60</session-timeout>
        </session-config>
    </web-app>
```

Create a `sun-jaxws.xml` deployment descriptor in the `config` folder. The `sun-jaxws.xml` file specifies the endpoints and contains implementation or container-specific information about the endpoints. The `sun-jaxws.xml` file is used by the runtime to determine which class to use in order to process incoming requests. The deployment descriptor has root element `endpoints` and one or more `endpoint` elements, each of which represents a port in the WSDL. The sample `sun-jaxws.xml` has one endpoint for `HelloWSPort` and has attributes that are discussed in the following table:

Attribute	Value	Description
name	`fromwsdl`	The endpoint name.
interface	`hellows.HelloWS`	Specifies the Service Endpoint Interface (SEI).
implementation	`hellows.HelloWSImpl`	Specifies the implementation class.
wsdl	`WEB-INF/wsdl/HelloWSService.wsdl`	Specifies the WSDL file location in the WAR file. Required element.
service	`{http://hellows/}HelloWSService`	Specifies the qualified name (QName) of the WSDL service. Required element.
port	`{http://hellows/}HelloWSPort`	Specifies the qualified name of the WSDL port. Required element.
url-pattern	`/hellows`	Specifies the URL pattern used to access the endpoint. Should be the same as the `url-pattern` in `web.xml`.

To learn more about elements in a `sun-jaxws.xml` refer to the `sun-jaxws.xsd` schema: `http://docs.oracle.com/cd/E17802_01/webservices/webservices/docs/2.0/jaxws/sun-jaxws.xsd`. Copy the `sun-jaxws.xml` listed below to the `config` folder:

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints
  xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
  version="2.0">
  <endpoint
    name="fromwsdl"
    interface="hellows.HelloWS"
    implementation="hellows.HelloWSImpl"
    wsdl="WEB-INF/wsdl/HelloWSService.wsdl"
    service="{http://hellows/}HelloWSService"
    port="{http://hellows/}HelloWSPort"
    url-pattern="/hellows" />
</endpoints>
```

Creating a client class

Create a client Java class `hellowsclient.HelloWSClient`. In the Java client application, create an instance of the `HelloWSService` service:

```
hellows.HelloWSService service=new HelloWSService();
```

The Service class will be created during the build, which is explained later. Obtain a proxy to the service from the service using the `getHelloWSPort()` method:

```
hellows.HelloWS port = service.getHelloWSPort();
```

Invoke the `hello(String)` method of the service using the service proxy:

```
String result = port.hello("John Smith.");
```

Output the result of the web service method invocation. The Java client class is listed as follows:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package hellowsclient;
/**
 *
 * @author dvohra
```

```
 */
import hellows.*;
public class HelloWSClient {
  /**
   * @param args
   */
  public static void main(String[] args) {
    hellows.HelloWSService service = new hellows.HelloWSService();
    hellows.HelloWS port = service.getHelloWSPort();
    String result = port.hello("John Smith.");
    System.out.println(result);
  }
}
```

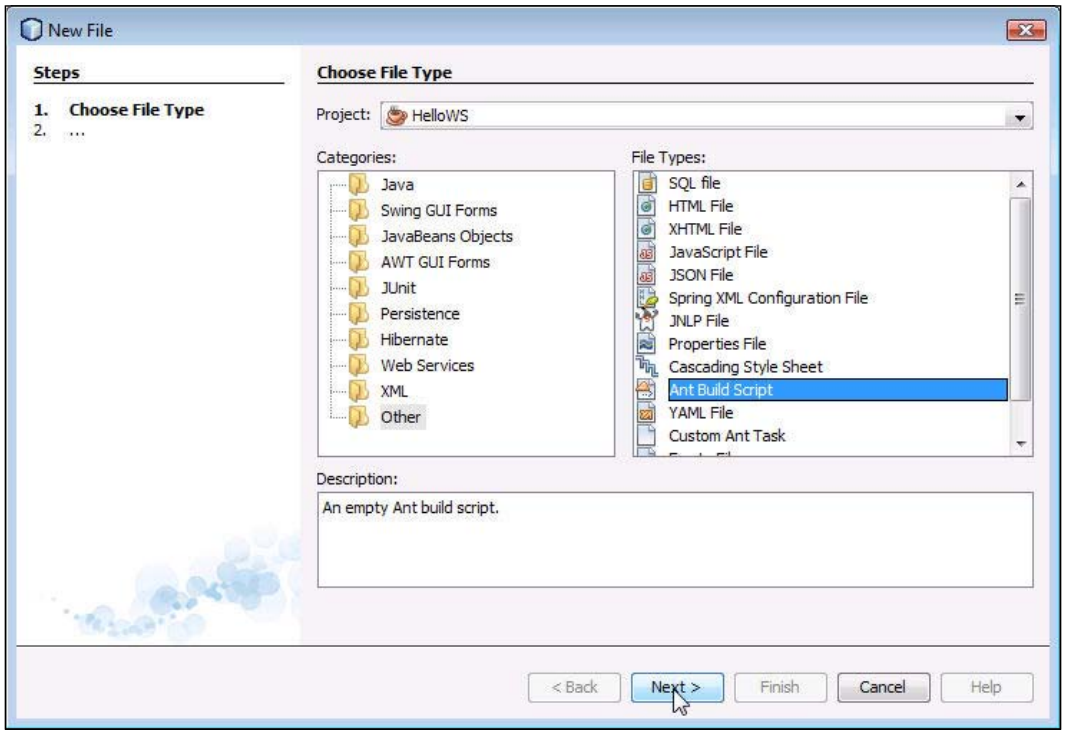
Creating the deployment targets

We shall be deploying the web service to the Oracle GlassFish Server. A build file fragment may be used for specifying deployment targets. Create a `deploy-targets.xml` file in the `config` directory. Specify properties for the GlassFish Server directory, the `build` directory, the `.war` file directory, the build classes directory, and the GlassFish domain name. Create a "deploy" target to deploy the web service WAR file to the `autodeploy` directory of the GlassFish Server domain. Deploying to the GlassFish Server is essentially copying to the `autodeploy` directory. The `deploy-targets.xml` file is listed as follows:

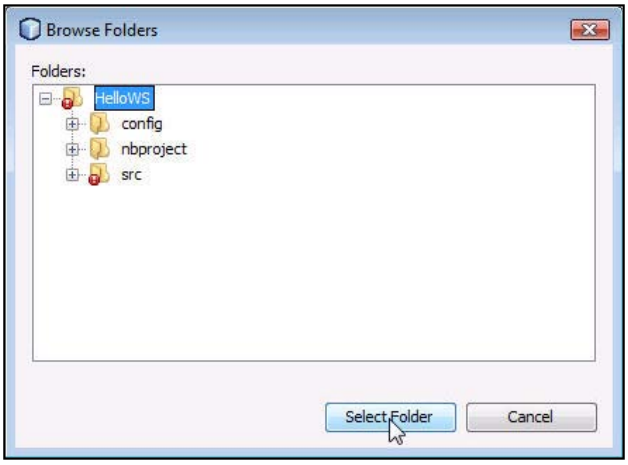
```
<?xml version="1.0" encoding="UTF-8"?>
<project>

  <property name="as.home" value="C:/glassfish3/glassfish"/>
  <property name="build.home" value="${basedir}/build"/>
  <property name="build.war.home" value="${build.home}/war"/>
  <property name="build.classes.home"
    value="${build.home}/classes"/>
  <property name="domain" value="domain1"/>

  <target name="deploy">
  <copy file="${build.war.home}/${ant.project.name}.war"
    todir="${as.home}/domains/${domain}/autodeploy"/>
  </target>

</project>
```

Creating an Apache Ant build file

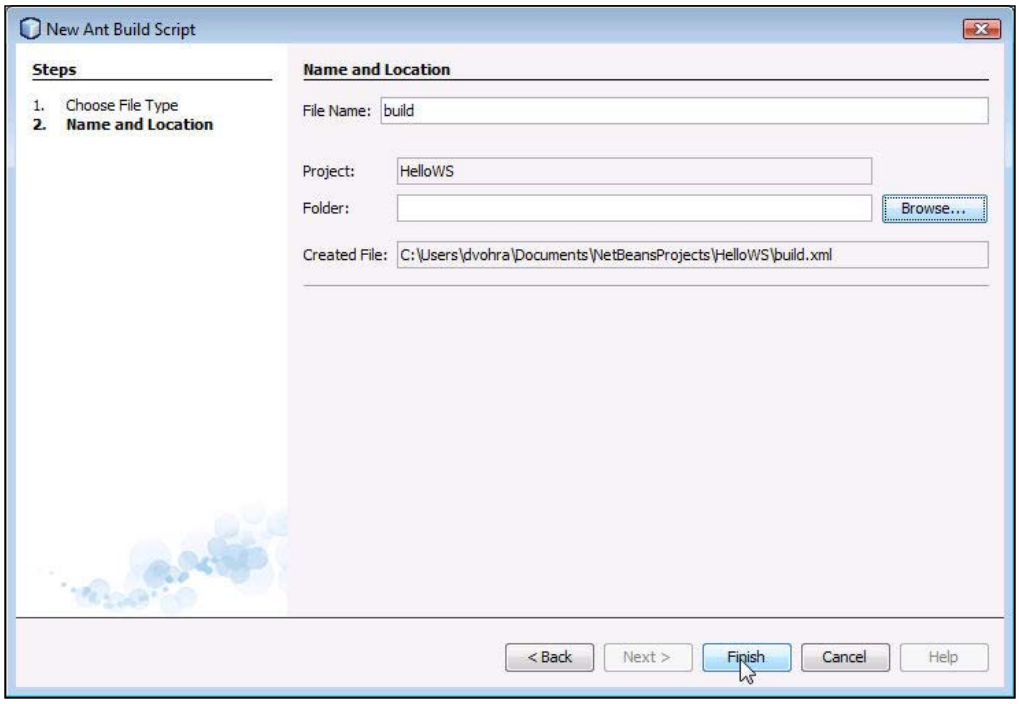
We shall build the web service using an Apache Ant `build.xml` file. To learn more about Apache Ant and creating build files refer to the URL `http://ant.apache.org/`. Create a `build.xml` file in the project root directory. Select **File | New File**. In **New File** select **Categories** as **Other** and **File Types** as **Ant Build Script**, and click on **Next**:



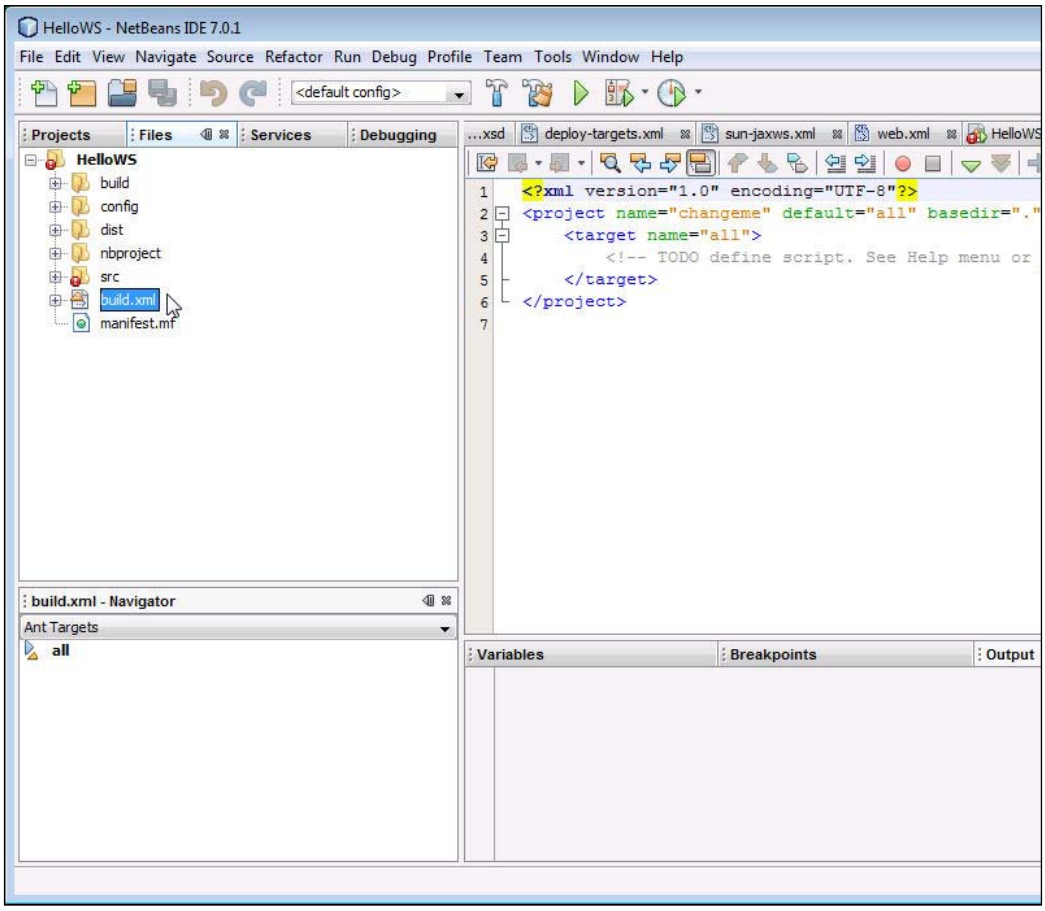
In the **Name and Location** window, click on the **Browse** button for **Folder**. In the **Browse Folders** window, select the project root folder **HelloWS**, and click on **Select Folder**:



In **Name and Location**, specify the **File Name** as **build**, **Project** as **HelloWS**, and click on **Finish**:



A **build.xml** file gets added to the **HelloWS** folder:



The `build.xml` file is used to compile, package, and deploy the web service to the GlassFish Server, use the `-clientjar` option to generate a JAR file for the web service portable artifacts and WSDLs, compile the client class, and run the client. In the `build.xml` file, add **property** elements for the properties listed in the following table, which also includes properties from the `deploy-targets.xml` build fragment:

Property	Value
`java.home`	`C:/Program Files/Java/jdk1.7.0`
`modules.home`	`C:/glassfish3/glassfish/modules`
`as.home`	`C:/glassfish3/glassfish`
`build.home`	`${basedir}/build`
`build.war.home`	`${build.home}/war`

Property	Value
build.classes.home	${build.home}/classes
domain	domain1

Create a `path` element to specify the classpath, which includes the `tools.jar` file from Java 7 and the JAR files from the GlassFish `modules` directory:

```
<path id="classpath">
  <pathelement location="${java.home}/lib/tools.jar"/>
  <fileset dir="${modules.home}">
    <include name="*.jar"/>
  </fileset>
</path>
```

In `build.xml`, we shall specify targets for the following:

1. Build the web service WSDL to generate web service artifacts using the `wsimport` task.
2. Compile the web service implementation class.
3. Create a WAR file from the web service classes and deployment descriptors.
4. Generate and package web service portable artifacts including the WSDLs into a JAR file using the new `-clientjar` option of `wsimport`.
5. Compile the client class `HelloWSClient.java`.
6. Run the web service client.

Specify the targets in `build.xml` listed in the following table, which also lists the targets from the `deploy-targets.xml` file:

Target	Description
setup	Build the required directories.
build-server-wsdl	Compiles the web service classes in the `hellows` directory and runs the `wsimport` task to generate web service artifacts.
create-war	Creates a WAR file from the compiled classes and includes the `sun-jaxws.xml` file and the WSDL.
generate-client	Generates the client JAR using the wsimport's `-clientjar` option.
client	Compiles the client class.
run	Runs the client class with the client JAR generated using the `-clientjar` option in the classpath.
deploy	Copies the web service WAR file to the GlassFish `autodeploy` directory.

Target	Description
server	Invokes the `clean, build-server-wsdl, create-war`, and `deploy` targets in the specified order.
clean	Deletes the `build` directory and its sub-directories.

The `build.xml` file is used to perform the following tasks:

- Building and deploying the service
- Building the client
- Running the client

Building and deploying the service

Add a task definition for the `com.sun.tools.ws.ant.WsImport` class, which defines the `wsimport` task:

```
<taskdef name="wsimport"
  classname="com.sun.tools.ws.ant.WsImport">
  <classpath refid="classpath"/>
</taskdef>
```

The new `-clientjar` option is not needed for the web service to get deployed on the GlassFish Server. In the `build-server-wsdl` target, run the `wsimport` task to generate the web service artifacts from the web service WSDL. Also, compile the web service implementation class:

```
<target name="build-server-wsdl" depends="setup">
  <wsimport
    debug="true"
    verbose="${verbose}"
    keep="false"
    destdir="${build.classes.home}"
    package="hellows"
    wsdl="${basedir}/config/HelloWSService.wsdl">
  </wsimport>
  <javac
    fork="true"
    srcdir="${basedir}/src"
    destdir="${build.classes.home}"
    includes="**/hellows/**">
    <classpath refid="classpath"/>
  </javac>
</target>
```

In the `create-war` target, package the web service classes including the deployment descriptors into a WAR file:

```
<target name="create-war">
  <war warfile="${build.war.home}/${ant.project.name}.war"
    webxml="config/web.xml">
  <webinf dir="${basedir}/config" includes="sun-jaxws.xml"/>
  <zipfileset
    dir="${basedir}/config"
    includes="*.wsdl, *.xsd"
    prefix="WEB-INF/wsdl"/>
    <classes dir="${build.classes.home}"
      includes="**/*.class"/>
  </war>
</target>
```

In the `server` target invoke `clean`, `build-server-wsdl`, `create-war`, and the `deploy` target from the `deploy-targets.xml` file to compile, package, and deploy the web service:

```
<target name="server" depends="setup">
  <antcall target="clean"/>
  <antcall target="build-server-wsdl"/>
  <antcall target="create-war"/>
  <antcall target="deploy"/>
</target>
```

Building the client

Next, we demonstrate the new `–clientjar` option in `wsimport` to generate a JAR file for the web service portable artifacts and WSDLs to be made available to the JAX-WS runtime when the web service is invoked from a client. In the `generate-client` target, run the `wsimport` task and specify the JAR file to package the generated artifacts and WSDLs using the `clientjar` attribute. Specify the WSDL URL as `http://localhost:8080/clientjar/hellows?wsdl` with the `wsdl` attribute. In the `wsdl` URL, `clientjar` is the WAR file name for the deployed web service and `hellows` is the servlet pattern to invoke the web service servlet as specified in `web.xml`. As the JAR file is to be made available to the client at runtime, specify the package name the same as the package for the client class:

```
<target name="generate-client">
  <wsimport
    debug="true"
    verbose="${verbose}"
    destdir="${build.classes.home}"
```

```
      package="hellowsclient"
      clientjar="HelloWSServiceClient.jar"
        wsdl="http://localhost:8080/clientjar/hellows?wsdl">
    </wsimport>
  </target>
```

Next, compile the web service client class `hellowsclient.HelloWSClient.java`. In the `client` target compile the client class using the `javac` task with the JAR file generated using the `-clientjar` option in the classpath:

```
  <target name="client" depends="generate-client">
    <javac
      fork="true"
      srcdir="${basedir}/src"
      destdir="${build.classes.home}"
      includes="/hellowsclient/**">
      <classpath>
        <path refid="classpath"/>
        <pathelement location="${build.classes.home}
          /HelloWSServiceClient.jar"/>
      </classpath>
    </javac>
  </target>
```

To learn more about the `Wsimport Ant` task in Java 7 refer to the URL `http://jax-ws.java.net/nonav/2.2.5/docs/wsimportant.html`.

Running the client

Next, we run the client class in the `run` target using the `java` task with the JAR file generated using the `-clientjar` option in the classpath:

```
  <target name="run">
    <java fork="true"  classname="hellowsclient.HelloWSClient">
      <classpath>
        <path refid="classpath"/>
        <pathelement location="${build.classes.home}"/>
        <pathelement location="${build.classes.home}
          /HelloWSServiceClient.jar"/>
        <pathelement location="${basedir}/config"/>
      </classpath>
      <jvmarg value="-Dcom.sun.xml.ws.transport.
        http.client.HttpTransportPipe.dump=${log}"/>
    </java>
  </target>
```

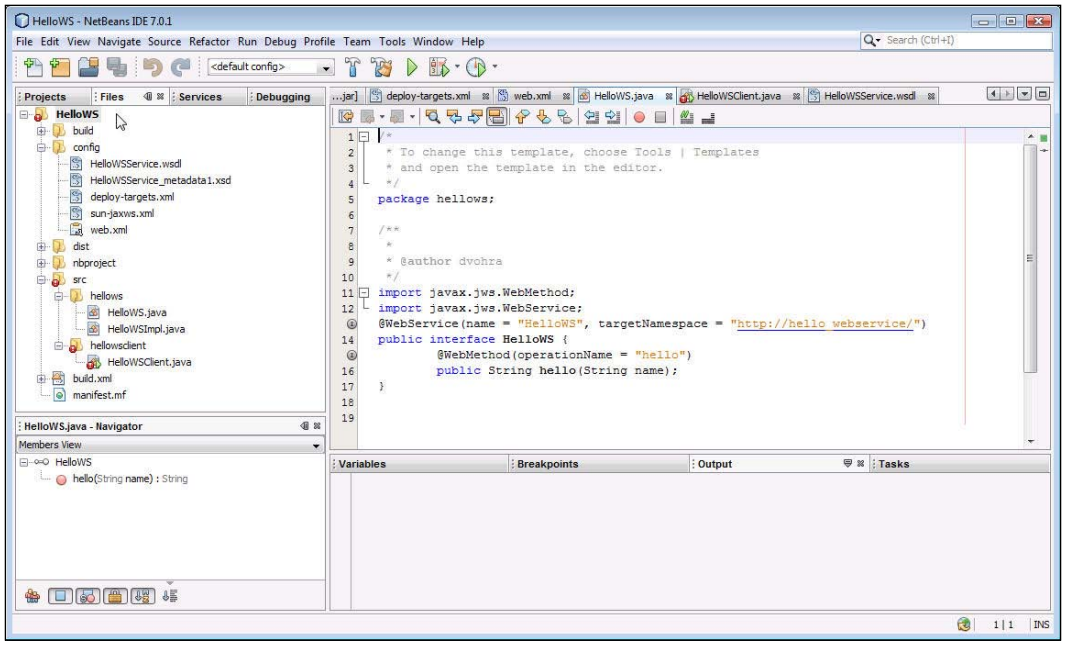
The `build.xml` is listed as follows:

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


<project basedir="." default="help" name="clientjar">
<property name="java.home"
  value="C:/Program Files/Java/jdk1.7.0" />
<property name="modules.home"
  value="C:/glassfish3/glassfish/modules" />
  <import file="config/deploy-targets.xml"/>

  <path id="classpath">
    <pathelement location="${java.home}/lib/tools.jar"/>
    <fileset dir="${modules.home}">
      <include name="*.jar"/>
    </fileset>
  </path>

  <taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport">
    <classpath refid="classpath"/>
  </taskdef>

  <target name="setup">
    <mkdir dir="${build.home}"/>
    <mkdir dir="${build.classes.home}"/>
    <mkdir dir="${build.war.home}"/>
  </target>

  <target name="clean">
    <delete dir="${build.home}" includeEmptyDirs="true"/>
  </target>

  <target name="build-server-wsdl" depends="setup">
    <wsimport
      debug="true"
      verbose="${verbose}"
      keep="false"
      destdir="${build.classes.home}"
        package="hellows"
        wsdl="${basedir}/config/HelloWSService.wsdl">
    </wsimport>
    <javac
      fork="true"
      srcdir="${basedir}/src"
      destdir="${build.classes.home}"
        includes="**/hellows/**">
        <classpath refid="classpath"/>
    </javac>
```

```
</target>

<target name="create-war">
  <war warfile="${build.war.home}
    /${ant.project.name}.war" webxml="config/web.xml">
  <webinf dir="${basedir}/config" includes="sun-jaxws.xml"/>
  <zipfileset
    dir="${basedir}/config"
    includes="*.wsdl, *.xsd"
    prefix="WEB-INF/wsdl"/>
  <classes dir="${build.classes.home}" includes="**/*.class"/>
  </war>
</target>

<target name="generate-client">
  <wsimport
    debug="true"
    verbose="${verbose}"
    destdir="${build.classes.home}"
    package="hellowsclient"
    clientjar="HelloWSServiceClient.jar"
      wsdl="http://localhost:8080/clientjar/hellows?wsdl">
  </wsimport>
</target>

<target name="client" depends="generate-client">
  <javac
    fork="true"
    srcdir="${basedir}/src"
    destdir="${build.classes.home}"
      includes="/hellowsclient/**">
    <classpath>
      <path refid="classpath"/>
      <pathelement location="${build.classes.home}
        /HelloWSServiceClient.jar"/>
    </classpath>
  </javac>
</target>

<target name="run">

  <java fork="true"  classname="hellowsclient.HelloWSClient">
    <classpath>
      <path refid="classpath"/>
      <pathelement location="${build.classes.home}"/>
      <pathelement location="${build.classes.home}
        /HelloWSServiceClient.jar"/>
      <pathelement location="${basedir}/config"/>
    </classpath>
```

```
        <jvmarg value="-Dcom.sun.xml.ws.transport.
          http.client.HttpTransportPipe.dump=${log}"/>
      </java>
    </target>


    <target name="server" depends="setup">
      <antcall target="clean"/>
      <antcall target="build-server-wsdl"/>
      <antcall target="create-war"/>
      <antcall target="deploy"/>
    </target>

  </project>
```

The directory structure of the web service project is shown in the following screenshot:

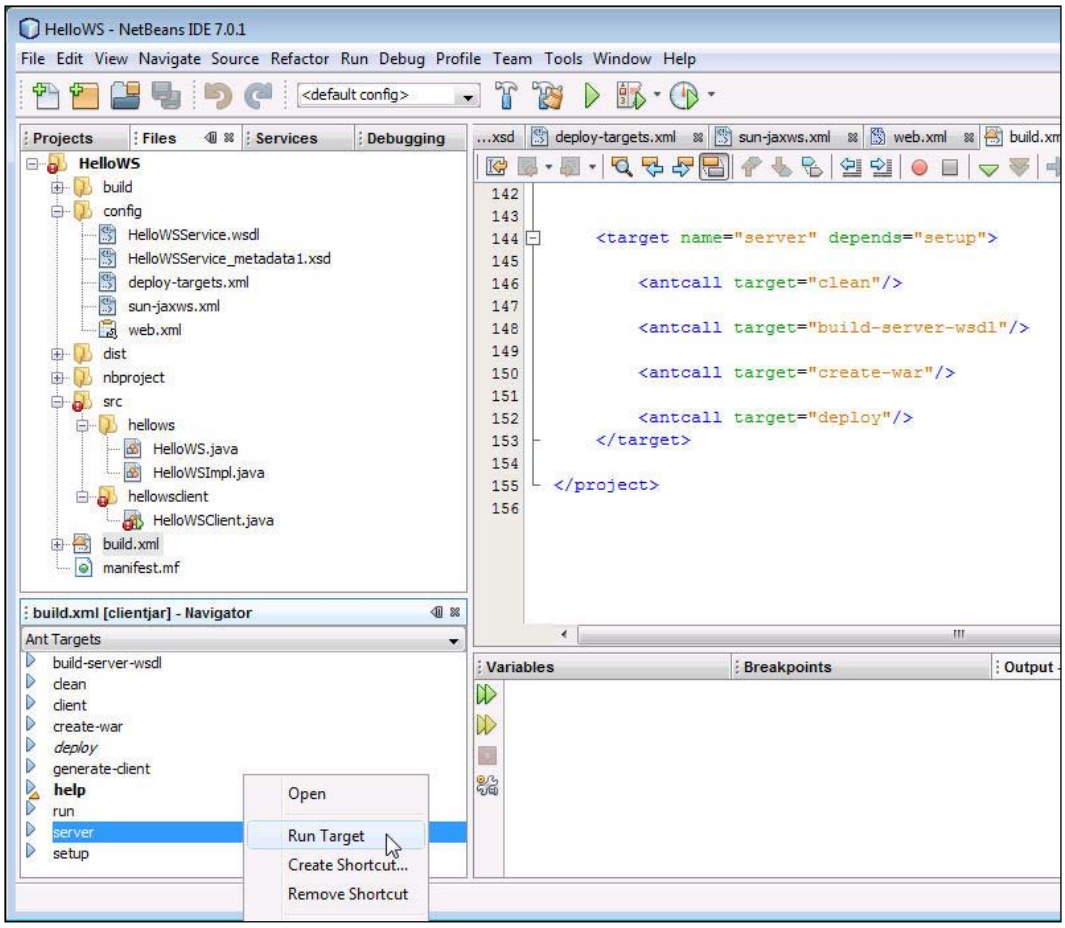


Testing the web service

Having generated the web service artifacts, packaged and deployed the web service, generated the runtime JAR file for the web service client, and compiled the client class, next we shall test the web service client. We shall run the `build.xml` targets in the following order:

- `server` target: To generate and deploy the web service WAR file to the GlassFish Server
- `client` target: To generate the JAR file using the `-clientjar` option of `wsimport` and compile the client class
- `run` target: To run the client class

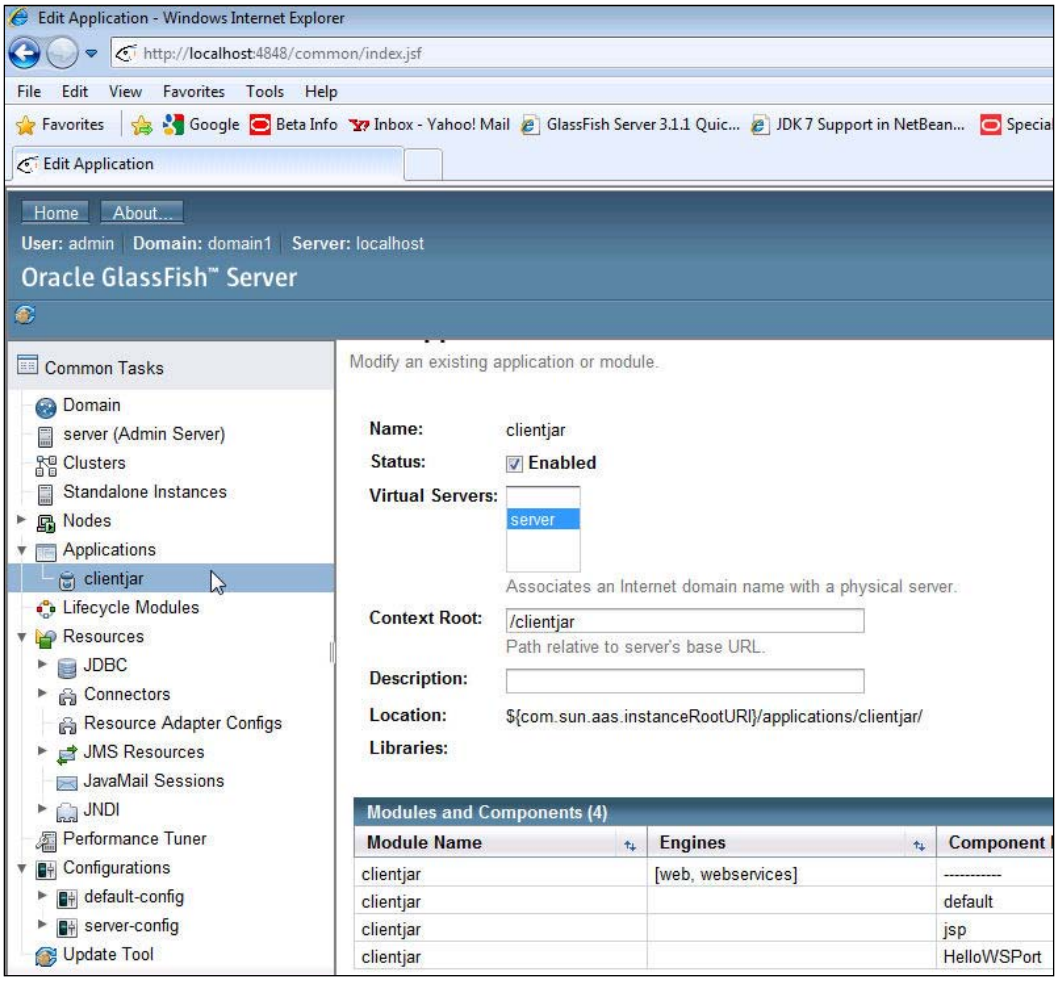
Right-click on the `server` target and select **Run Target**:



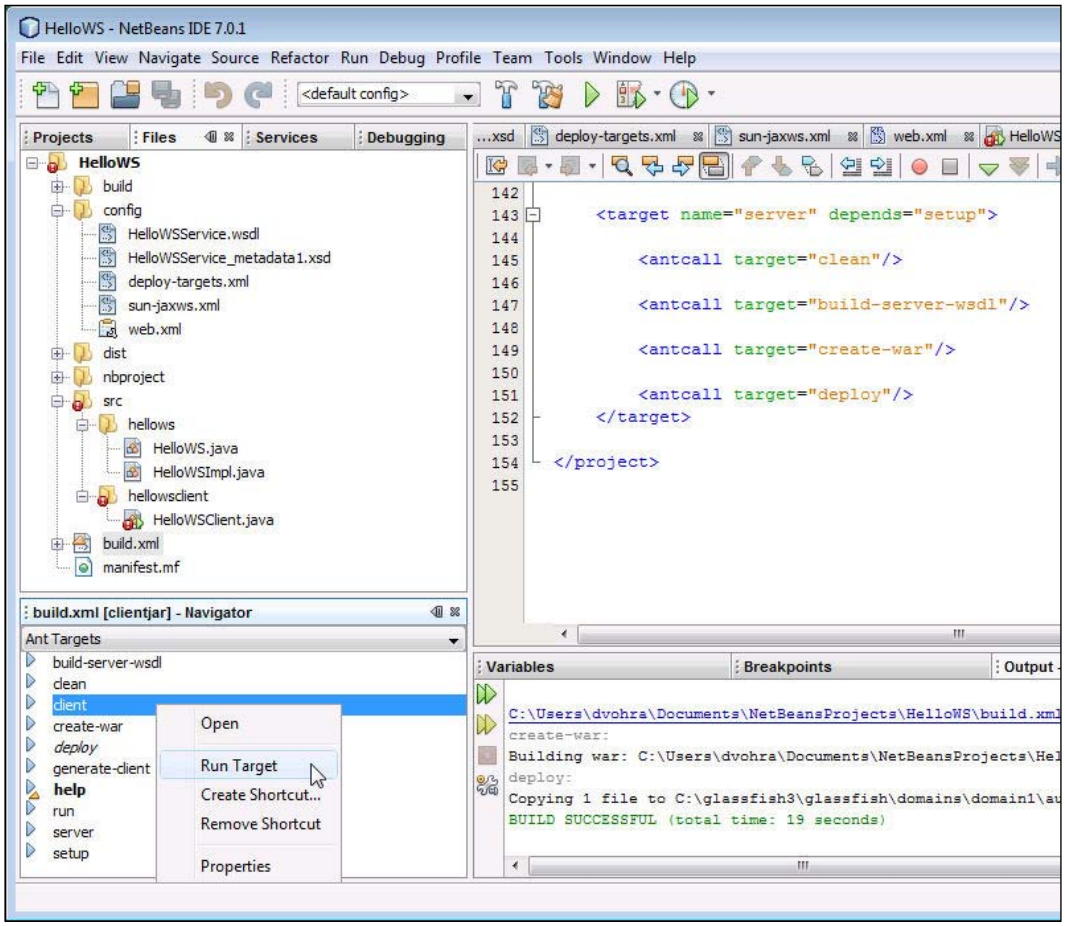
The output from the `server` target indicates that the `clean`, `setup`, `build-server-wsdl`, `create-war`, and `deploy` targets are run to deploy the `clientjar.war` file to the GlassFish Server:

```
clean:
Deleting directory C:\Users\dvohra\Documents\NetBeansProjects\HelloWS\
build
setup:
Created dir: C:\Users\dvohra\Documents\NetBeansProjects\HelloWS\build
Created dir: C:\Users\dvohra\Documents\NetBeansProjects\HelloWS\build\
classes
Created dir: C:\Users\dvohra\Documents\NetBeansProjects\HelloWS\build\
war
build-server-wsdl:
parsing WSDL...
Generating code...
Compiling code...

Compiling 1 source file to C:\Users\dvohra\Documents\NetBeansProjects\
HelloWS\build\classes
create-war:
Building war: C:\Users\dvohra\Documents\NetBeansProjects\HelloWS\
build\war\clientjar.war
deploy:
Copying 1 file to C:\glassfish3\glassfish\domains\domain1\autodeploy
BUILD SUCCESSFUL (total time: 21 seconds)
```

The `clientjar` WAR file is deployed on the Oracle GlassFish Server, as shown in the following screenshot:



Next, run the **client** target to generate the runtime JAR file and compile the client class. Right-click on the `client` target and select **Run Target**:

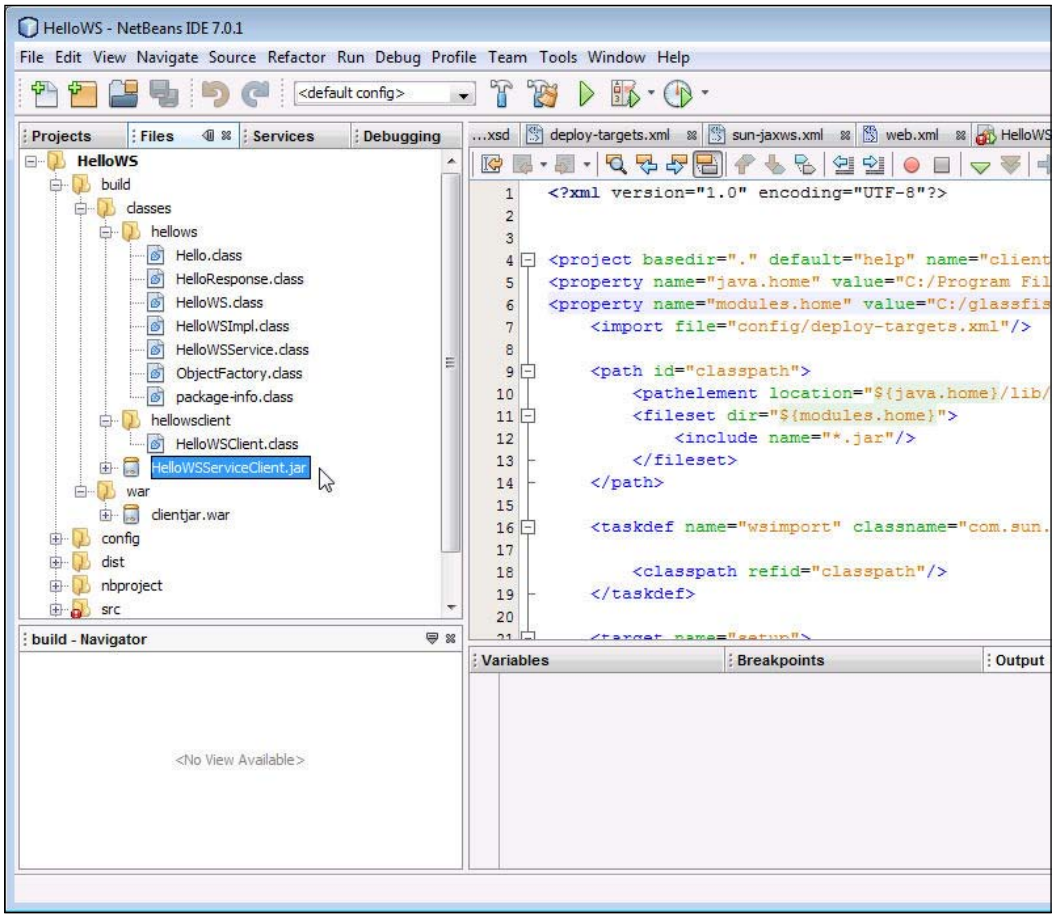


The `generate-client` and `client` targets start running and the runtime JAR file `HelloWSServiceClient.jar` is generated:

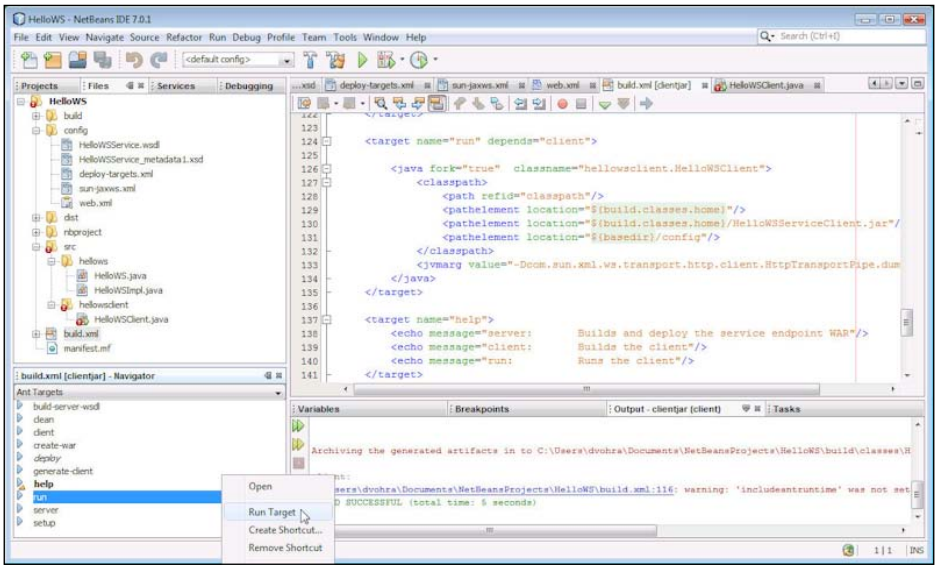
```
generate-client:
Consider using <depends>/<produces> so that wsimport won't do
unnecessary compilation
parsing WSDL...
Downloading the WSDL and associated metadata
Generating code...
Compiling code...
Archiving the generated artifacts in to C:\Users\dvohra\Documents\
NetBeansProjects\HelloWS\build\classes\HelloWSServiceClient.jar.
```

```
client:
Compiling 1 source file to C:\Users\dvohra\Documents\NetBeansProjects\
HelloWS\build\classes
BUILD SUCCESSFUL (total time: 9 seconds)
```

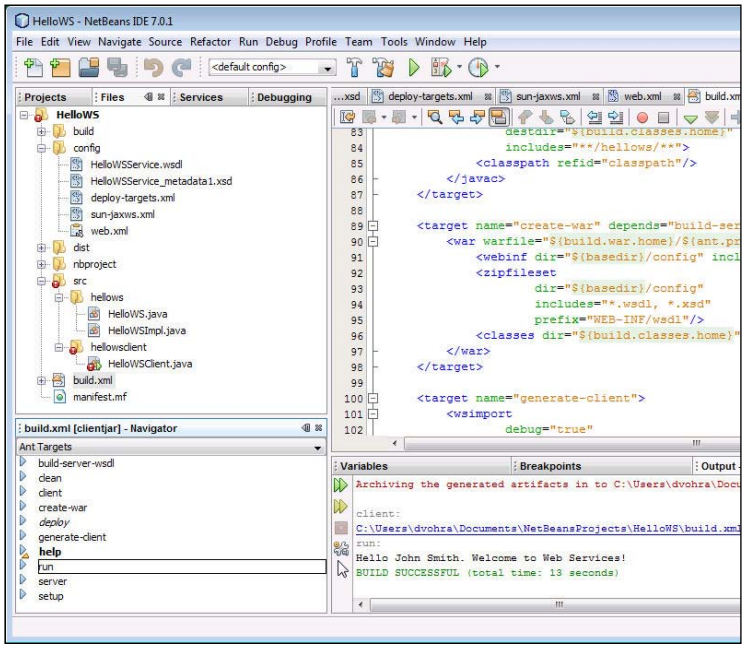
The build classes generated by the `build-server-wsdl` and `client` Ant tasks, and the client JAR generated by the `generate-client` task are shown in the following screenshot:



Next, test the client class by running the `run` target. Right-click on the `run` target and select **Run Target**:



The client class runs and the web service gets invoked to generate a Hello message as shown in the following screenshot:



The output from the `run` target is as follows:

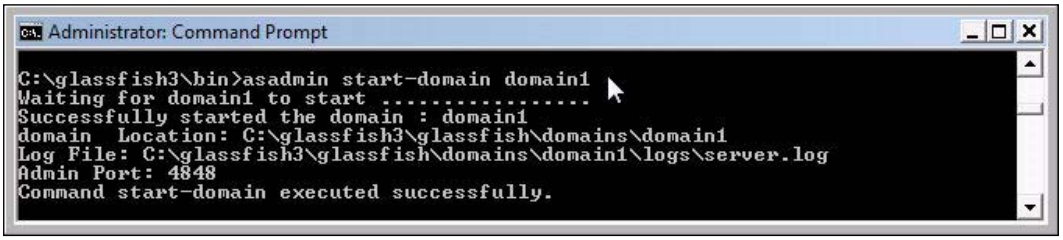
```
run:
Hello John Smith. Welcome to Web Services!
BUILD SUCCESSFUL (total time: 10 seconds)
```

Using the wsimport tool from the command line

We used the Ant task `wsimport` to generate the client JAR file in the `generate-client` target. We may also use the `wsimport` tool to generate the client JAR from the command line. First, we need to start the GlassFish Server using the following command:

```
asadmin start-domain domain1
```

`Domain1` starts successfully as shown in the command output:



Next, generate the client JAR file using the `wsimport` tool with the following options:

Option	Value	Description
`-p`	`hellowsclient`	Package name of the client-side classes
`-d`	`C:\Users\dvohra\Documents\NetBeansProjects\HelloWS\build\classes`	Destination directory of the client JAR
`-verbose`		Output messages about the compiler
`-clientjar`	`HelloWSServiceClient.jar`	The client JAR file name

For other command line options refer to the URL `http://jax-ws.java.net/nonav/2.2.3/docs/wsimport.html`. Run the `wsimport` command with the options and the WSDL URL:

```
wsimport -p hellowsclient -verbose -d C:\Users\dvohra\Documents\
NetBeansProjects\HelloWS\build\classes -clientjar HelloWSServiceClient.
jar http://localhost:8080/clientjar/hellows?wsdl
```

The output from the `wsimport` command shows that the client JAR `HelloWSServiceClient.jar` is generated in the `build/classes` folder:

Summary

In this chapter we discussed the procedure to develop a JAX-WS web service and test the web service with a client. We used the `wsimport` task/tool to generate client-side web service artifacts and used the new `-clientjar` option in `wsimport` of Java 7 to package the artifacts and WSDL/s into a JAR file. Making a JAR file containing web service artifacts and WSDLs in the runtime of the client precludes the requirement to access the WSDLs over the network at runtime, thus saving on network overhead.

Index

Symbols

A

B

C

D

E

H

I

J

M

N

O

P

R

S

T

U

W

Thank you for buying

Java 7 JAX-WS Web Services

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] enterprise
professional expertise distilled
PUBLISHING

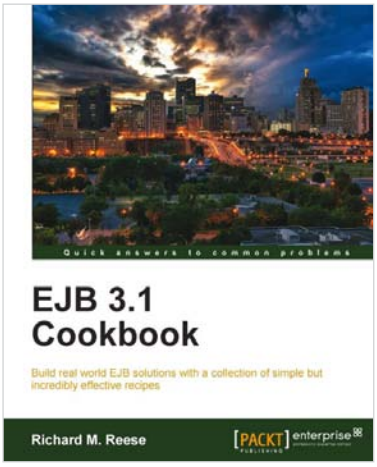
Java 7 New Features Cookbook

ISBN: 978-1-84968-562-7 Paperback: 384 pages

Over 100 comprehensive recipes to get you up-to-speed with all the exciting new features of Java 7

1. Comprehensive coverage of the new features of Java 7 organized around easy-to-follow recipes
2. Covers exciting features such as the try-with-resources block, the monitoring of directory events, asynchronous IO and new GUI enhancements, and more
3. A learn-by-example based approach that focuses on key concepts to provide the foundation to solve real world problems

---



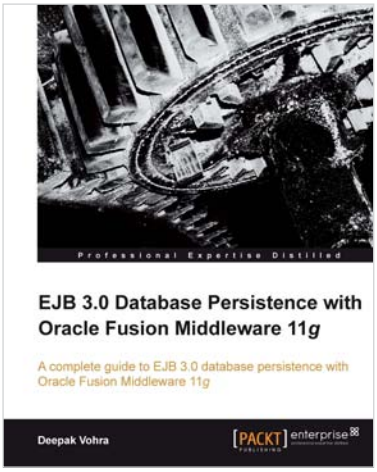
EJB 3.1 Cookbook

ISBN: 978-1-84968-238-1 Paperback: 436 pages

Build real world EJB solutions with a collection of simple but incredibly effective recipes

1. Build real world solutions and address many common tasks found in the development of EJB applications
2. Manage transactions and secure your EJB applications
3. Master EJB Web Services
4. Part of Packt's Cookbook series: Comprehensive step-by-step recipes illustrate the use of Java to incorporate EJB 3.1 technologies

[PACKT] enterprise
professional expertise distilled
PUBLISHING



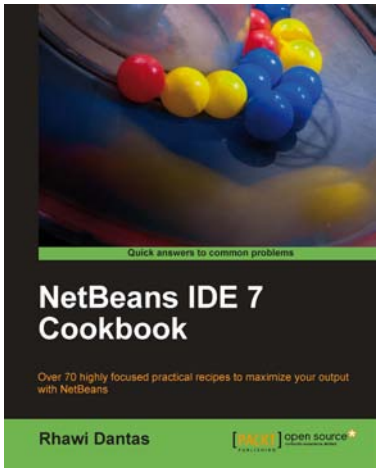
EJB 3.0 Database Persistence with Oracle Fusion Middleware 11g

ISBN: 978-1-84968-156-8 Paperback: 448 pages

A complete guide to EJB 3.0 database persistence with Oracle Fusion Middleware 11*g*

1. Integrate EJB 3.0 database persistence with Oracle Fusion Middleware tools: WebLogic Server, JDeveloper, and Enterprise Pack for Eclipse
2. Automatically create EJB 3.0 entity beans from database tables
3. Learn to wrap entity beans with session beans and create EJB 3.0 relationships

---



NetBeans IDE 7 Cookbook

ISBN: 978-1-84951-250-3 Paperback: 308 pages

Over 70 highly focused practical recipes to maximize your output with NetBeans

1. Covers the full spectrum of features offered by the NetBeans IDE
2. Discover ready-to-implement solutions for developing desktop and web applications
3. Learn how to deploy, debug, and test your software using NetBeans IDE
4. Another title in Packt's Cookbook series giving clear, real-world solutions to common practical problems