opentext™

OpenText™ Documentum™ Content Management

Composer BOF Development Tutorial

Understand business object framework class loading and build a business object framework module.

EDCPC250400-PTL-EN-01

OpenText™ Documentum™ Content Management
Composer BOF Development Tutorial
EDCPC250400-PTL-EN-01
Rev.: 2025-Oct-20

This documentation has been created for OpenText™ Documentum™ Content Management CE 25.4.
It is also valid for subsequent software releases unless OpenText has made newer documentation available with the product, on an OpenText website, or by any other means.

Open Text Corporation

275 Frank Tompa Drive, Waterloo, Ontario, Canada, N2L 0A1

Tel: +1-519-888-7111
Toll Free Canada/USA: 1-800-499-6544 International: +800-4996-5440
Fax: +1-519-888-0677
Support: https://support.opentext.com
For more information, visit https://www.opentext.com

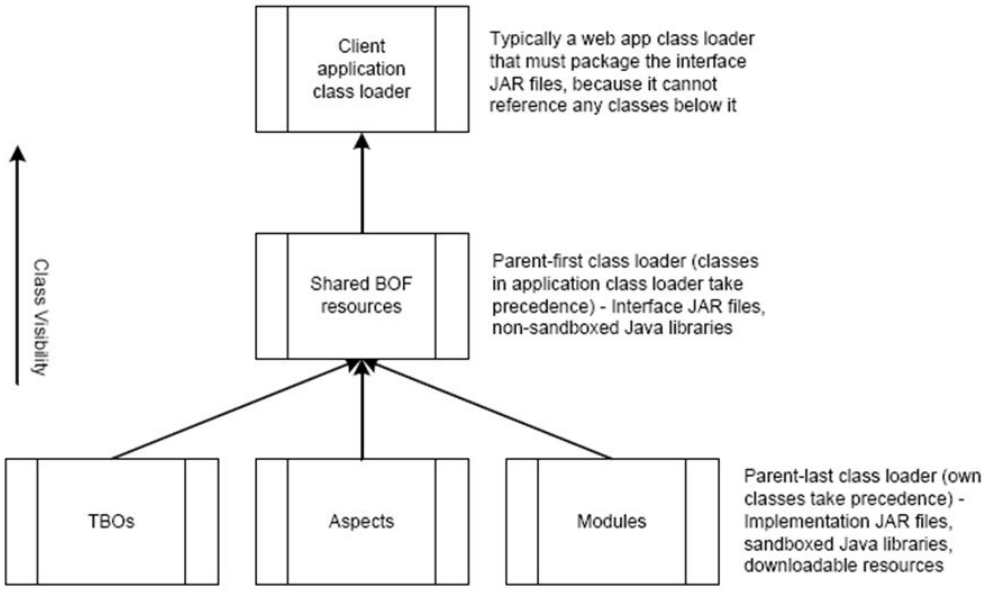
Table of Contents

Chapter 1

Understanding BOF class loading

1.1 BOF class loader hierarchy

Documentum Composer categorizes JAR files into three categories: Implementation, Interface, and Mixed. A different class loader is used to load the classes depending on the type of JAR file. The following diagram shows the class loader hierarchy:



Implementation JAR files are loaded into module-specific class loaders. Each BOF application has its own class loader and all of its implementation JAR files are loaded into this class loader. Classes in different module-specific class loaders cannot reference one another. This class loader is parent-last, meaning classes within the class loader take precedence over the same classes in its parent class loaders (the shared BOF class loader and the application class loader).

Interface JAR files are loaded in a class loader that is shared across a Foundation Java API instance. All interface JAR files for all BOF applications are loaded into the shared BOF class loader. This allows published interfaces to be shared across multiple components that require it. This class loader is parent-first, meaning classes that are loaded by its parent class loader (the application class loader) take precedence over its own classes. Classes in this class loader cannot reference classes in the module-specific class loaders.

Mixed JAR files are deprecated and should not be used.

The application class loader is typically where your client or web application is loaded. This class loader cannot reference any of the classes that are loaded by the shared BOF class loader or the module-specific class loaders. You must package any interface JAR files that are needed by your client application with the client application, so it is aware of your BOF application's API.

1.2 What to package in JAR files

The different types of JAR files and their class loading behavior in a repository are more complex than in your development environment. Because of this fact, applications that work in your development environment might throw exceptions when deployed in a repository. It is important to understand what to package in implementation and interface JAR files before developing your BOF application. As a general rule, you should not use mixed JAR files.

Implementation JAR files typically contain the logic of your application and a primary class that serves as the entry point to your BOF module. Implementation JAR files are loaded into a module-specific class loader and classes in different BOF modules cannot reference one another.

Interface JAR files do not need to include all interfaces, but all published interfaces that are needed by your clients should be packaged. Any exception classes or factory classes that appear in your interfaces should also be packaged in interface JAR files. Ensure that your interfaces do not transitively expose classes that are not packaged in the interface JAR file, which would lead to a NoClassDefFoundException.

To be safe, package your implementation and published interface classes in different JAR files. Implementation JAR files can contain non-published interfaces (the interface is not needed by the client or any other BOF module). It is recommended that you also separate your implementation and published interface source files into separate directories so that the packaging process is less error prone.

1.3 Common exceptions caused by incorrect JAR packaging

If you do not package your JAR files correctly, the different layers of class loaders can cause exceptions to occur. The two most common exceptions that you might encounter are ClassDefNotFound and ClassCastException.

The ClassCastException occurs when you try to cast an object to an interface or class that it does not implement. In most cases, you will be sure that the object you are casting implements the interface that you are trying to cast it to, so there is another point to consider when encountering this error. Java considers a class to be the same only when it has the same fully qualified name (package and class name) and if it is loaded by the same class loader. If you accidentally package a published interface within an implementation JAR file, the exception occurs if you try to cast an object to that interface in your client application.

For instance, say you created a BOF module that implements an interface and package the interface in an implementation JAR file:

- The interface that is packaged in the implementation JAR resolves to the module-specific class loader because it is parent last.
- Your client application instantiates the BOF module and tries to cast it to the interface. It uses the interface that is loaded by the client application class loader, because there is no way for your application to reference the interface in the module class loader (the parent class loader cannot see children class loaders).
- Java throws ClassCastException, because it expects the interface that was loaded by the module-specific class loader to be used to cast the BOF module, but you are using the one that was loaded by your application class loader. Alternatively, if you correctly package the interface inside an interface JAR file, it is loaded by the shared BOF class loader, which is parent-first. The interface resolves to its parent class loader first (your application's class loader), and no exception is thrown.

NoClassDefFoundException most often occurs when you transitively expose a class that a class loader cannot find. A common example is when you accidentally package an implementation class inside an interface JAR file, and that implementation class references another class in an implementation JAR file. The exception is thrown, because classes in the shared BOF class loader cannot reference anything in the module-specific class loaders.

For instance, say you created a BOF module and accidentally packaged an implementation class inside an interface JAR file:

- You call a method that references the implementation class that you accidentally packaged inside the interface JAR file.
- The method runs and in turns references another class inside the implementation JAR file.
- Java throws the NoClassDefFoundException, because the classes that are loaded in the shared BOF class loader cannot see any classes that are loaded by the module-specific class loaders (classes in parent class loaders cannot see classes in child class loaders). This problem can manifest itself in other scenarios, but the basic problem involves referencing classes that do not exist (either through not packaging the class at all or packaging the class at a level that is hidden from the referencing class).

Understanding these two common problems can help you avoid them when developing your BOF applications. Because of these problems, it is recommended that you separate your implementation source files and published interface source files into different folders to begin with, so that packaging the JAR files is a simpler process.

Chapter 2

Building the Hello World BOF module

Now that you have some background on BOF development, this tutorial will guide you through developing a simple Hello World BOF module. You will create the following items when developing the Hello World BOF module:

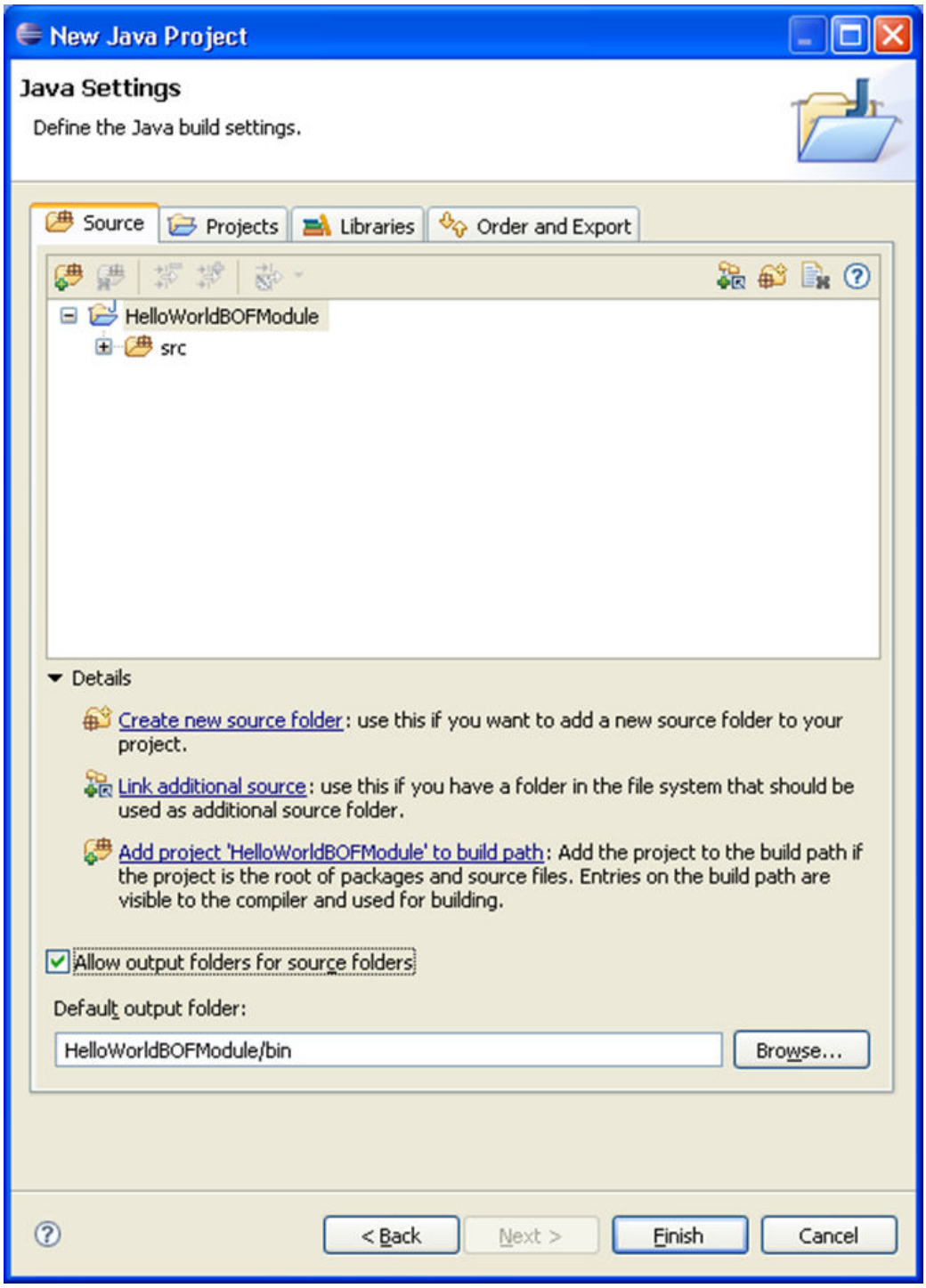
- An interface for the Hello World BOF module.
- An implementation class for the Hello World BOF module.
- An Ant script that builds the interface and implementation classes into separate JAR files.
- A JAR Definition artifact that defines the implementation and interface JAR files as a Documentum artifact.
- A Module artifact that defines the BOF module.
- An Ant script that automatically updates the JAR Definition with the most recent version of the implementation JAR file and builds and installs the project.

2.1 Creating the Hello World Java project

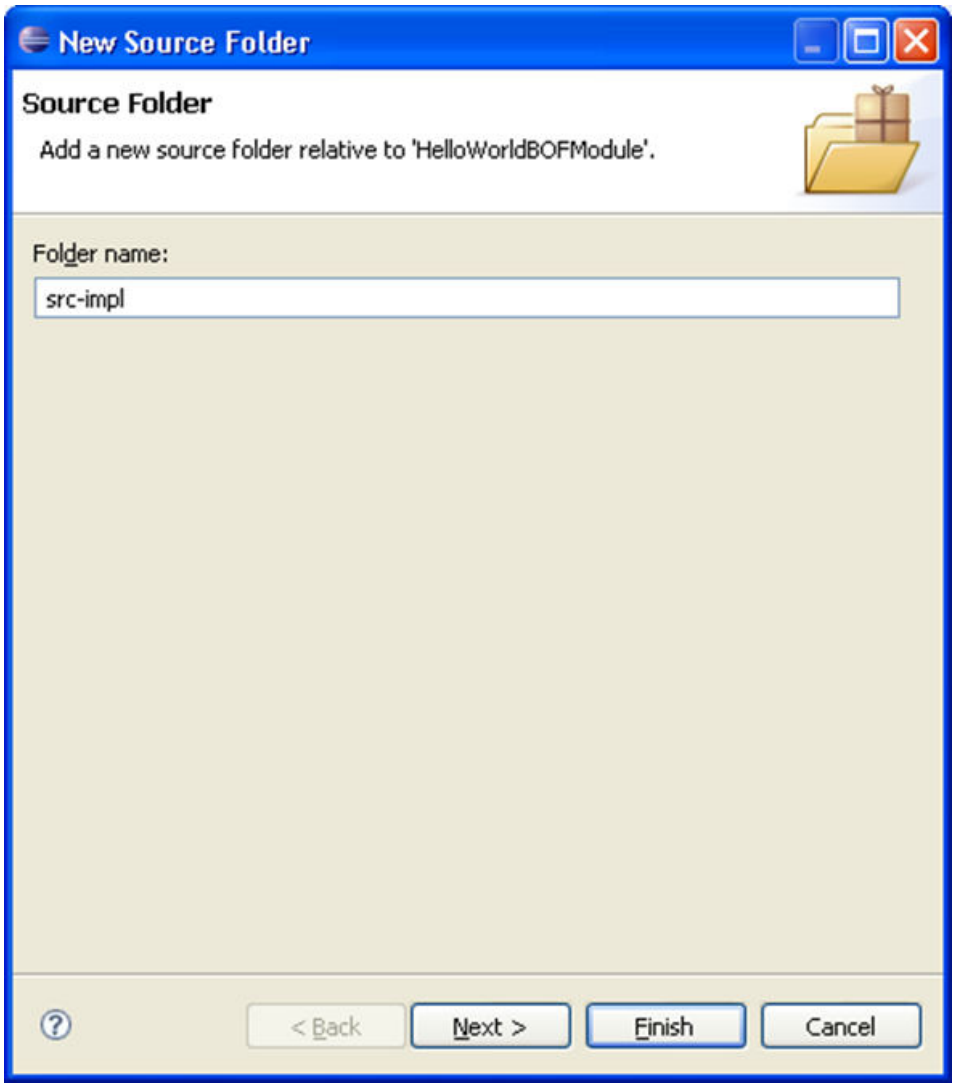
To begin, you will first create a Java project that contains the code for the Hello World BOF module. During the project creation, you will also create two separate source folders for your implementation and interface classes.

To create the Java project:

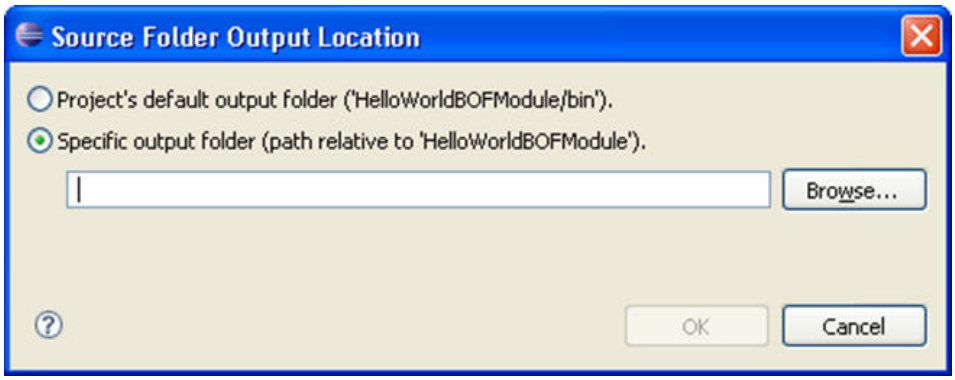
1. Start Documentum Composer and click **File > New > Java Project**. The **New Java Project** wizard appears.
2. Type `HelloWorldBOFModule` in the **Project Name** field and click **Next**. The **Java Settings** screen appears.
3. Click on the **Allow output folders for source folders** check box.



4. Click **Create new source folder**. The **Source Folder** window appears.
5. Type src-impl in the **Folder name** field and click **Finish**.



6. Click **Configure Output Folder Properties**. The **Source Folder Output Location** window appears.
7. Click the **Specific output folder** radio button and click **Browse**.



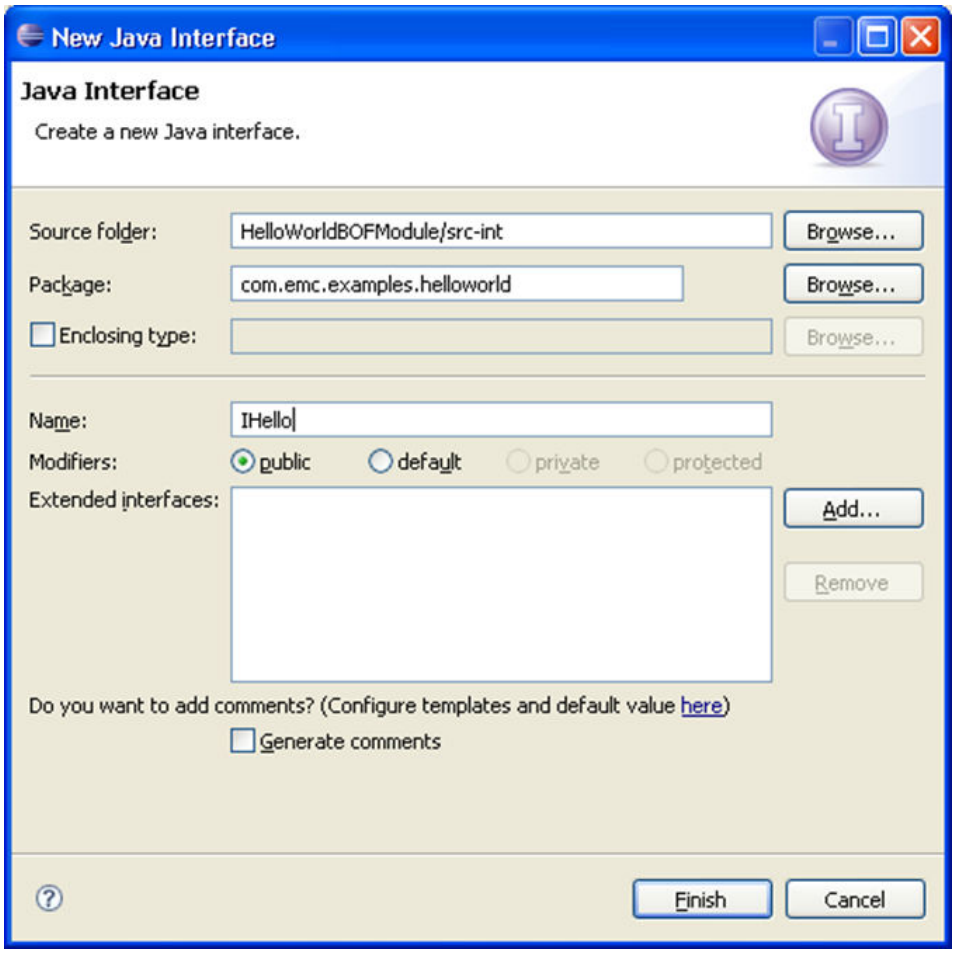
8. Click the **HelloWorldBOFModule** node and click **Create New Folder**. The **New Folder** window appears.
9. In the **Folder name** field, type `bin-impl` and click **OK** three times to return to the **Java Settings** window.
10. Repeat steps 4-9, but create a source folder named `src-int` with an output folder of `bin-int`.
11. Click on the **Libraries** tab and click **Add External JARs**.
12. Browse for `<Composer_root>\plugins\com.emc.ide.external.dfc_1.0.0\lib\dfc.jar` and click **Open**.
13. Click **Finish** to create the project. If you are prompted to switch to the Java Perspective, do so.

2.1.1 Creating the IHello interface

The IHello interface defines one method: sayHello. This interface will be packaged in a JAR file that is designated as an Interface JAR when defining the JAR Definition artifact.

To create the IHello interface:

1. In the **Package Explorer** view, right-click the `src-int` folder and select **New > Interface**. The **New Java Interface** window appears.
2. Specify the following values for the following fields and click **Finish**:
   - Package – com.emc.examples.helloworld
   - Name – IHello



3. Replace the code in the `IHello.java` file with the following code and save the file:

```
package com.emc.examples.helloworld;
    public interface IHello {
        public void sayHello();
}
```

2.1.2 Creating the Hello World implementation class

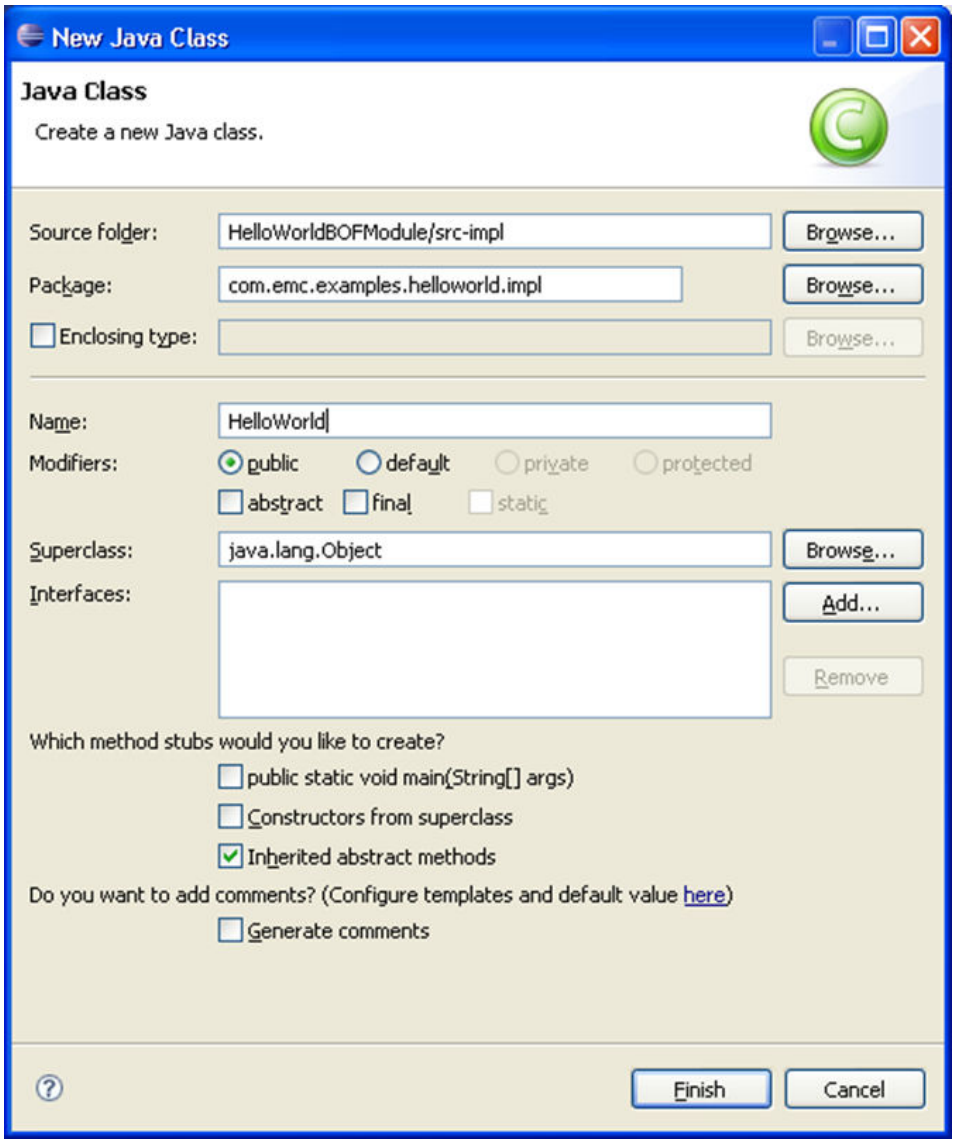
The HelloWorld class implements the IHello interface and prints out a “Hello, World” string when its sayHello method is called. The HelloWorld class will be packaged in a JAR file that will be designated as an Implementation JAR when defining the JAR Definition artifact.

To implement the IHello interface:

1. In the **Package Explorer** view, right-click the `src-impl` folder and select **New > Class**. The **New Java Class** window appears.

2. Specify the following values for the following fields and click **Finish**:

   - Package – com.emc.examples.helloworld.impl
   - Name – HelloWorld



3. Replace the code in the `HelloWorld.java` file with the following code and save the class:

```
package com.emc.examples.helloworld.impl;

import com.documentum.fc.client.IDfModule;
import com.emc.examples.helloworld.IHello;

public class HelloWorld implements IHello, IDfModule{
    public void sayHello() {
```

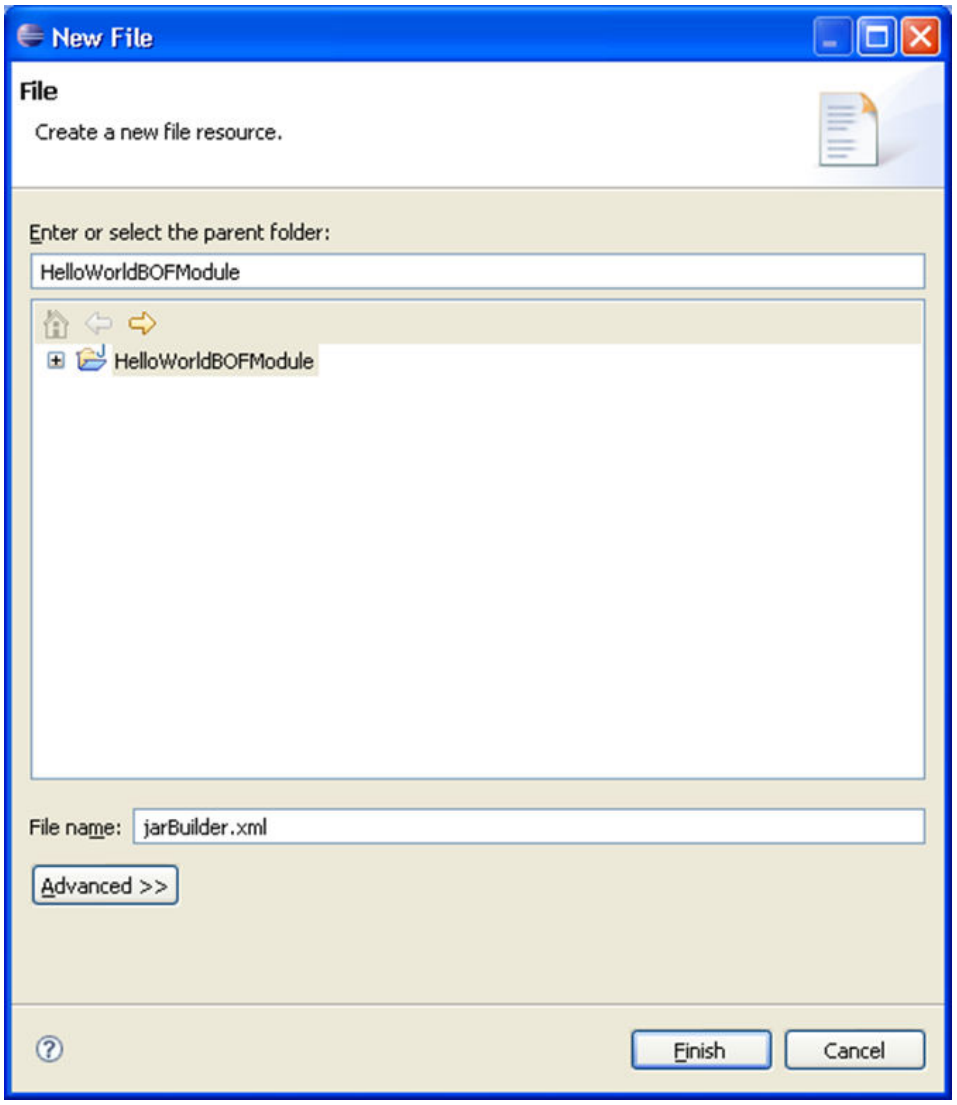
```
        System.out.println("Hello World!");
    }
}
```

2.1.3 Creating the JAR Ant Builder

You will now create an Ant Builder to automatically build the interface and implementation classes into two separate JAR files: `hello-api.jar` and `hello.jar`. When you make changes to any of your code, the Ant Builder automatically rebuilds the JAR files.

To create the Ant Builder:

1. Right click the **HelloWorldBOFModule** node in the **Package Explorer** view and select **New > File**. The **New File** window appears.
2. In the **File name** field, type `jarBuilder.xml` and click **Finish**.

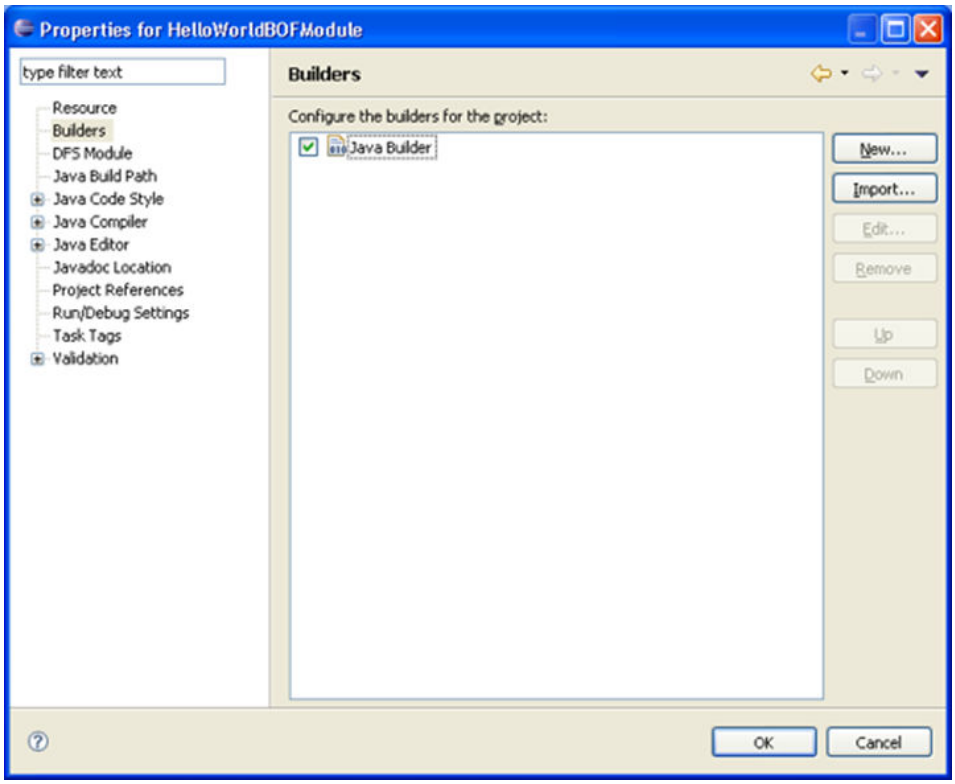


The `jarBuilder.xml` file appears in the **Package Explorer** view and is opened in an editor.

3. Click on the **Source** tab in the XML file editor, copy and paste the following code into the editor, and save the file:

```
<project name="JARBuilder" default="main">
    <target name="main">
        <delete file="bin-impl/hello.jar" />
        <delete file="bin-int/hello-api.jar" />
        <jar destfile="bin-impl/hello.jar" basedir="bin-impl"/>
        <jar destfile="bin-int/hello-api.jar" basedir="bin-int"/>
        <eclipse.refreshLocal resource="HelloWorldBOFModule"/>
    </target>
</project>
```

4. Right click the **HelloWorldBOFModule** node in the **Package Explorer** view and select **Properties**. The **Properties for HelloWorldBOFModule** window appears.



5. Select **Builders** on the left and click **New**. The **Choose configuration type** window appears.

6. Select **Ant Builder** from the list and click **OK**. The **Edit Configuration** window appears.

7. Specify the following values for the fields listed and click **OK**:

   - Main tab > Name – `JAR_Builder`
   - Main tab > Buildfile – Click **Browse workspace** and select **HelloWorldBOFModule > jarBuilder.xml**.
   - Targets tab > Auto Build – Click **Set Targets**, select **main**, and click **OK**.

8. Select **JAR_Builder** and click **Up** to move JAR_Builder above Java Builder and click **OK**. The Ant Builder builds the JAR files and outputs them to the `bin-impl` and `bin-int` folders.

2.2 Creating the Documentum Composer project

The Documentum Composer project contains the Documentum artifacts that are needed for your BOF module. You will create a project that contains a Module artifact along with JAR Definition artifacts.

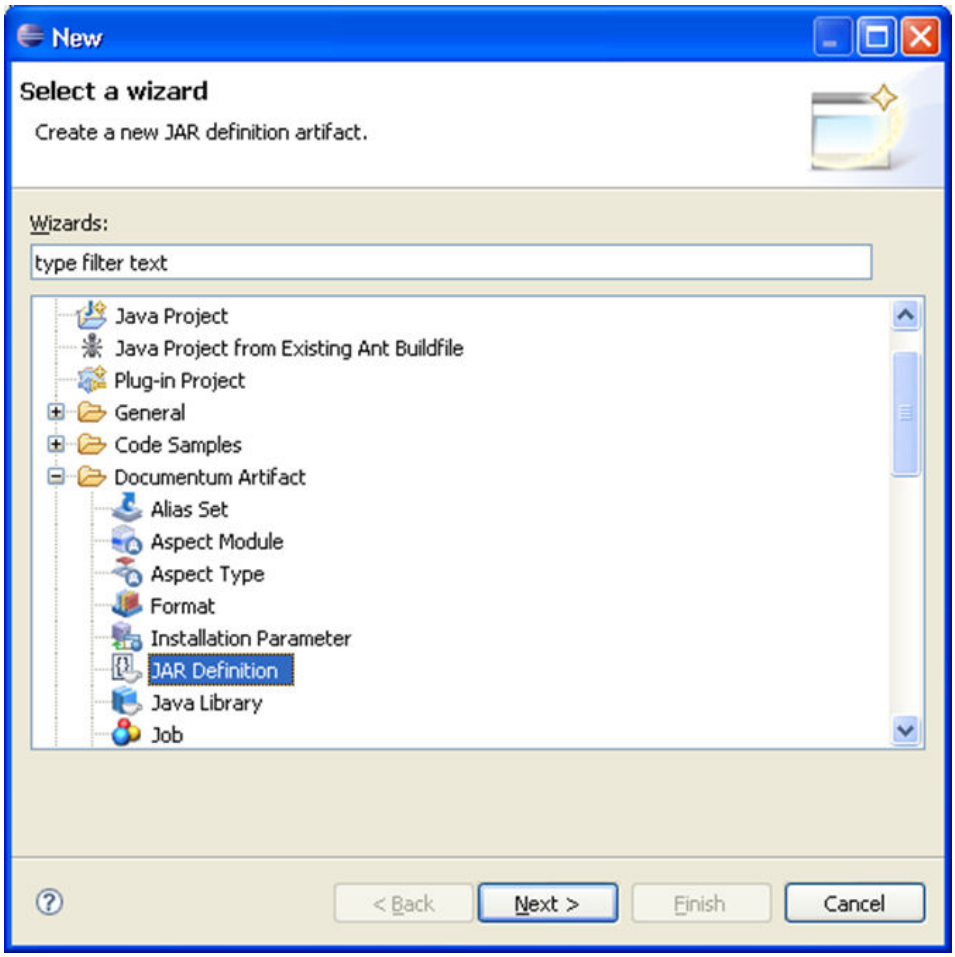
1. Click **File > New Project**, select **Documentum Project > Documentum Project** from the **New Project wizard**, and click **Next**. The **New Documentum Project** window appears.
2. In the **Project Name** field, type `HelloWorldArtifacts` and click **Finish**. Documentum Composer takes a few minutes to create the project. If you are prompted to switch to the **Documentum Artifacts** perspective, do so.

2.2.1 Creating the JAR Definition artifacts

Before you create the BOF module artifact, you must create JAR Definition artifacts that reference your implementation and interface JAR files. The BOF module cannot reference JAR files directly.

To create the JAR Definition artifacts:

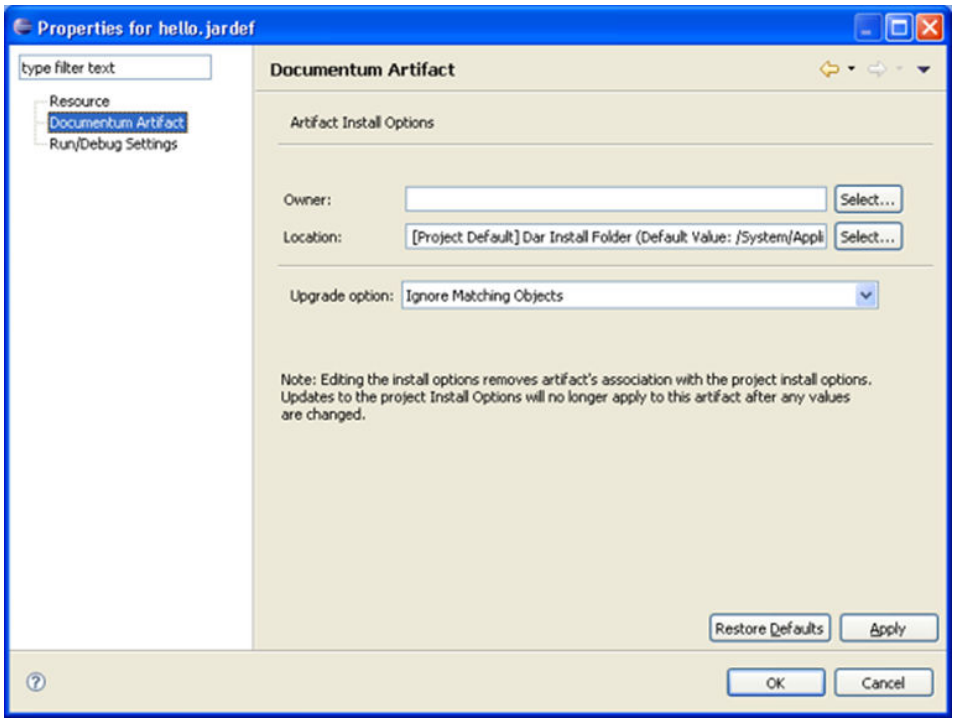
1. In the **Documentum Navigator** view, right click the **HelloWorldArtifacts > Artifacts > JAR Definitions** folder and select **New > Other**. The **New Wizard** appears.
2. Select **Documentum Artifact > JAR Definition** and click **Next**.



3. In the **Artifact name** field, type `hello` and click **Finish**. The hello editor opens.
4. In the **JAR Content** section, click **Browse** and select the `hello.jar` file that is located in the `<workspace>\HelloWorldBOFModule\bin_impl` directory.
5. In the **Type** list, select **Implementation** and save the JAR definition.
6. In the **Documentum Navigator** view, right click the **HelloWorldArtifacts > Artifacts > JAR Definitions** folder and select **New > Other**. The **New Wizard** appears.
7. Select **Documentum Artifact > JAR Definition** and click **Next**.
8. In the **Artifact name** field, type `hello-api` and click **Finish**. The hello-api editor opens.
9. In the **JAR Content** section, click **Browse** and select the `hello-api.jar` file that is located in the `<workspace>\HelloWorldBOFModule\bin_int` folder.
10. In the **Type** list, select **Interface** and save the JAR definition.

11. In the **Documentum Navigator** view, right-click **HelloWorldArtifacts > Artifacts > JAR Definitions > hello.jardef** and select **Properties**.

12. Select **Documentum Arfitact** on the left, select **Ignore Matching Objects** for the **Upgrade option** field and click **Apply**.



13. For the **Upgrade option** field, re-select the **Create New Version of Matching Objects** option and click **OK**. Documentum Composer does not set the **Create New Version of Matching Objects** option unless you set it to something else first. This bug will be addressed in future releases. This option allows the client to detect new changes in JAR files in the repository. If you do not set the JAR Definition to this property, updated JAR files will not get downloaded to the client unless the BOF cache is cleared.

The `hello.jar` and `hello-api.jar` files are now associated with a JAR definition and can be used within a module. If you decide to modify any code within these JAR files, you must remove the JAR file from the JAR definition and re-add it. You must do this, because Documentum Composer does not use the JAR file in the location that the Ant builder outputs it to. Documentum Composer actually copies that JAR file to another location and uses that copy. The Ant Builder that you previously created updates the JAR file, but does not update the JAR Definition artifact.

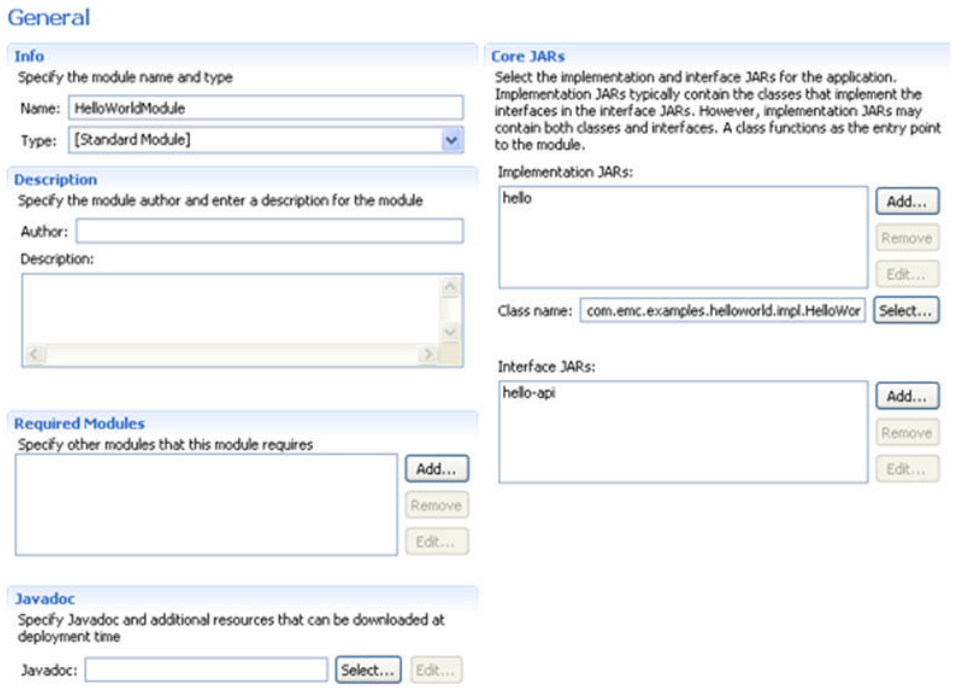
You can update the artifact manually by clicking the **Remove** button, clicking the **Browse** button, and reselecting the appropriate JAR file whenever the JAR file is updated. Later on in this tutorial, you will learn how to automate this requirement with another Ant script and Headless Composer.

2.2.2 Creating the BOF module artifact

Now that you have created all of the necessary components, you can now create the actual BOF module artifact.

To create the BOF module artifact:

1. In the **Documentum Navigator** view, right click the **HelloWorldArtifacts > Artifacts > Modules** folder and select **New > Other**. The **New Wizard** appears.
2. Select **Documentum Artifacts > Module** and click **Next**.
3. In the **Artifact name** field, type **HelloWorldModule** and click **Finish**. The HelloWorldModule editor opens.



4. In the **Implementation JARs** section, click the **Add** button, select **hello** from the list that appears, and click **OK**.
5. For the **Class name** field, click **Select**, select **com.emc.examples.helloworld.impl.HelloWorld** from the list, and click **OK**. This sets the HelloWorld class as the entry point for the module.
6. In the **Interface JARs** section, click the **Add** button, select **hello-api** from the list that appears, and click **OK**.
7. Save the module. The `hello.jar` file and the `hello-api.jar` file are now associated with the module. You can now install the module to a repository.

2.3 Installing the BOF module

Now that you have created all of the needed artifacts, you can install the BOF module to the repository. Once installed, the module can be downloaded on-demand by clients that require it.

To install the BOF module:

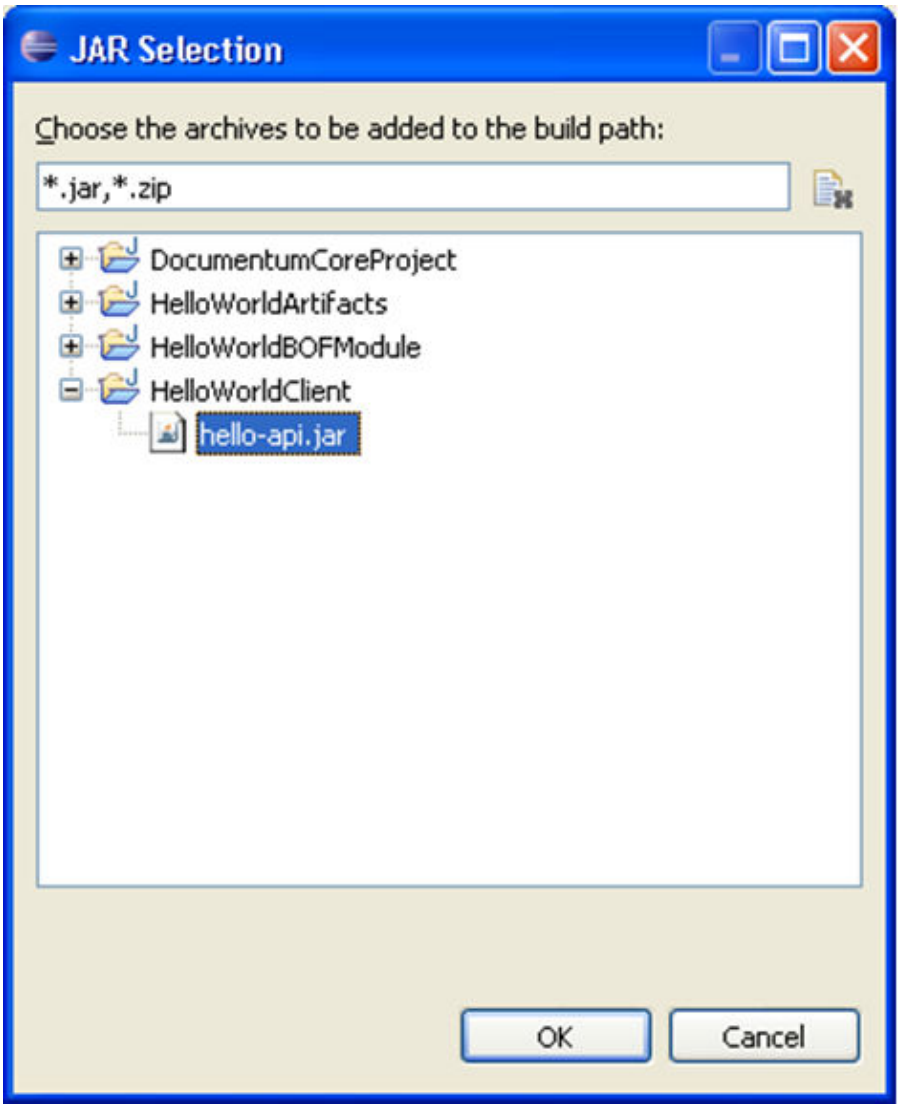
1. Ensure that your `<Composer_root>\plugins\com.emc.ide.external.dfc_1.0.0\documentum.config` is properly configured with the correct connection broker information.
2. In the **Documentum Navigator** view, right-click the **HelloWorldArtifacts** node and select **Install Documentum Project**. The **Install Wizard** appears.
3. Select the repository that you want to install the BOF module, enter the credentials for that repository, and click **Login**. If the Login is successful, the **Finish** button is enabled.
4. Click **Finish** to install the project.

2.4 Creating the HelloWorld BOF module client

Once the BOF module is installed, you can write a client to test its functionality. When writing a client, you must include all of the interfaces that your client requires in your classpath. In this case, the client requires the `hello-api.jar` interface JAR file. If you do not package the interface, the client is unaware of the API for the BOF module.

To create the client:

1. Create the project:
   a. In Documentum Composer, click **File > New Project**.
   b. Select **Java Project** and click **Next**.
   c. In the **Project Name** field, type `HelloWorldClient` and click **Finish**. If prompted to switch to the Java perspective, do so.
2. Add `hello-api.jar` and `dfc.jar` to the build path:
   a. Copy the `hello-api.jar` file from the `<workspace>\HelloWorldBofModule\bin-int` directory to the `<workspace>\HelloWorldClient` directory.
   b. Right-click the **HelloWorldClient** node in the **Documentum Navigator** view and select **Properties**.
   c. Select **Java Build Path** from the left and click on the **Libraries** tab.
   d. Click **Add JARs**, select **HelloWorldClient > hello-api.jar** to add the JAR file, and click **OK**.



e. Click **Add External JARs**, select `<Composer>\plugins\com.emc.ide.external.dfc_1.0.0\lib\dfc.jar`, and click **OK**.

f. Click **OK** again to close the **Properties for HelloWorldClient** window.

3. Create the `dfc.properties` file for the client:

a. In the **Documentum Navigator** view, right click the **HelloWorldClient > src** folder and select **New > File**. The **New File** window appears.

b. In the **File name** field, type `dfc.properties` and click **Finish**.

c. Specify values for the `dfc.properties` file as follows:

```
dfc.docbroker.host[0]=<Docbroker host>
dfc.docbroker.port[0]=<Docbroker port>

# Global registry settings are optional,
# but the client will throw an exception if not specified
```

```
dfc.globalregistry.repository=<global registry repository name>
dfc.globalregistry.username=<global registry repository user>
dfc.globalregistry.password=<global registry repository password>
```

4. Optionally, create the `log4j2.properties` file for the client. If you do not have this file, the log4j logger will use a default configuration, but will post warnings to the console.

   a. In the **Documentum Navigator** view, right-click the **HelloWorldClient > src** folder and select **New > File**. The **New File** window appears.

   b. In the **File name** field, type `log4j2.properties` and click **Finish**.

   c. Specify values for the `log4j2.properties` file as follows:

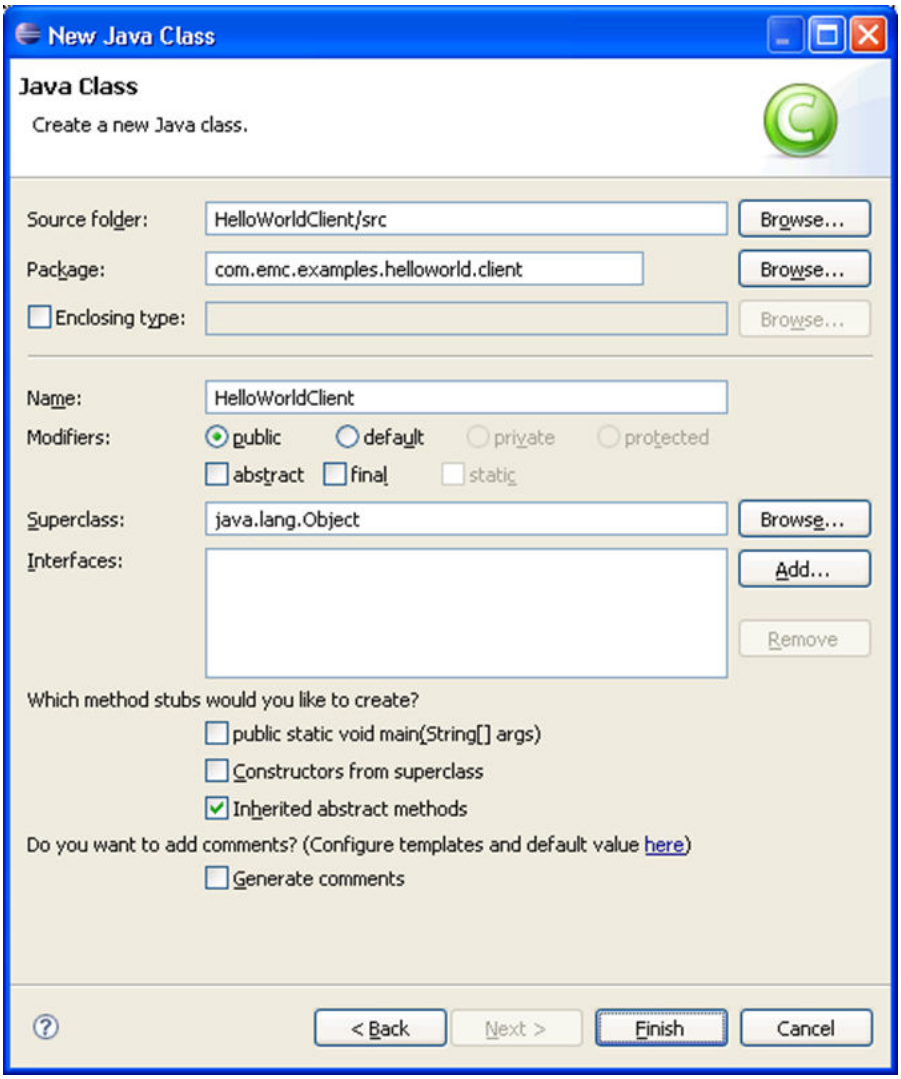
```
# ***** Set root logger level to WARN and its two appenders to stdout and R.
log4j.rootLogger=warn, stdout, R

# ***** stdout is set to be a ConsoleAppender.
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
# ***** stdout uses PatternLayout.
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
# ***** Pattern to output the caller's file name and line number.
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n

# ***** R is set to be a RollingFileAppender.
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=example.log
# ***** Max file size is set to 100KB
log4j.appender.R.MaxFileSize=100KB
# ***** Keep one backup file
log4j.appender.R.MaxBackupIndex=1
# ***** R uses PatternLayout.
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n
```

5. Create the client class file:

   a. Right-click the **src** folder and select **New > Class**. The **New Java Class** window appears.

   b. Specify the following values for the fields and click **Finish**:

      - Package – com.emc.examples.helloworld.client
      - Name – HelloWorldClient



c. Copy and paste the following code for the `HelloWorldClient.java` and save the file:

```
package com.emc.examples.helloworld.client;

import com.documentum.fc.client.IDfSession;
import com.documentum.com.DfClientX;
import com.documentum.fc.client.DfClient;
import com.documentum.fc.client.DfServiceException;
import com.documentum.fc.client.IDfModule;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.DfLoginInfo;
import com.documentum.fc.common.IDfLoginInfo;
import com.emc.examples.helloworld.IHello;

public class HelloWorldClient {

    public void run(String repository, String user,
        String password, String module){
       System.err.println("Connecting to repository [" + repository +
```

```
               "]    as user [" + user + "]");
      IDfSession session = null;
      IDfSessionManager sm = null;
     try {
         sm = newSessionManager(repository, user, password);
       session = sm.getSession(repository);
       run(sm, session, module, repository);
       }
    catch (Exception e) {
       e.printStackTrace();
    }
    finally {
       if (sm != null) {
          if (session != null)
             sm.release(session);
          sm.clearIdentities();
       }
    }
   }

   private IDfSessionManager newSessionManager(String docbase, String user,
        String password) throws DfException
   {
      IDfSessionManager sm = DfClient.getLocalClient().newSessionManager();
      IDfLoginInfo info = new DfLoginInfo();
      info.setUser(user);
      info.setPassword(password);
      info.setSecurityMode(IDfLoginInfo.SECURITY_MODE_TRY_NATIVE_FIRST);
      sm.setIdentity(docbase, info);
      return sm;
   }

   public void run(IDfSessionManager manager, IDfSession session,
        String module, String repository)
        throws Exception
   {
     IDfModule idfModule = null;
        try
     {
           idfModule = new DfClientX().getLocalClient().newModule(repository,
                module, manager);

        }
     catch (DfServiceException e)
     {
            e.printStackTrace();
        }
     catch (DfException e)
     {
            e.printStackTrace();
        }

        IHello hello = (IHello)idfModule;
        hello.sayHello();
     }
     public static void main(String args[]){
        HelloWorldClient client = new HelloWorldClient();
        client.run("GlobalRegistry", "Administrator", "emc",
"HelloWorldModule");
     }
}
```

6. In the **Documentum Navigator** view, right-click the `HelloWorldClient.java` file and select **Run As > Java application**. The Documentum Composer console should output the message “Hello, World!” if everything runs correctly.

Chapter 3

Automating the updating, building, and installation process with Headless Composer and Ant

Now that you have a working BOF module and client, it is useful to have a process in place to update the BOF module in the repository automatically. Previously, you learned that when updating code in JAR files, you had to also remove and re-add the JAR file to the appropriate JAR definition if you wanted the JAR definition to pick up the new changes. You can automate this step with Headless Composer, a command line driven version of Documentum Composer that is used for build and deployment. The Ant scripts that you will create automatically update the hello JAR Definition with the most recent `hello.jar` implementation JAR file, build the project, and install it to a repository.

To create the Headless Composer Ant scripts:

1. Extract the Headless Composer package to a location of your choice. The extraction process unzips the package to a ComposerHeadless directory. In our examples, it is assumed Headless Composer is unzipped to the C:\ drive.
2. Modify the `ComposerHeadless\plugins\com.emc.ide.external.dfc_1.0.0\documentum.config\dfc.properties` to specify the correct connection broker information. You can copy your existing the `dfc.properties` file from Documentum Composer UI if you want to use the same settings.

   Note: dfc.globalregistry.password and dfc.security.ssl.truststore_password can be managed by Vault. For more information about Vault, see *Documentum Server* documentation.
3. Create a directory named HelloWorldBuild in the ComposerHeadless directory.
4. Create a batch file, `ComposerHeadless\HelloWorldBuild\run.bat`, that defines the necessary environment variables and runs the Ant scripts. An example batch file is shown in the following sample. You can modify the strings in bold to meet your environment needs:

```
ECHO OFF
REM Set environment variables to only apply to this command prompt
SETLOCAL

REM Sets the root location of headless Composer
SET ECLIPSE="C:|ComposerHeadless"

REM Sets the location of your source projects. This location gets copied into
REM your build workspace directory
SET PROJECTSDIR="C:|My|Projects"

REM Sets the workspace directory where Composer builds the projects that you
REM want to install to a repository
SET BUILDWORKSPACE="C:|ComposerHeadless|HelloWorldBuild|build_workspace"

REM Sets the workspace directory where Composer extracts built DAR files before
```

```
REM installing them to a repository
SET INSTALLWORKSPACE="C:|ComposerHeadless|HelloWorldBuild|install_workspace"

REM Sets the Ant script that builds your projects
SET BUILDFILE="C:|ComposerHeadless|HelloWorldBuild|build.xml"

REM Sets the Ant script that installs your projects
set INSTALLFILE="C:|ComposerHeadless|HelloWorldBuild|install.xml"

set JDK_JAVA_OPTIONS=--add-modules java.se --add-exports java.base/
jdk.internal.ref=ALL-UNNAMED --add-opens java.base/java.lang=ALL-UNNAMED --add-
opens java.base/java.nio=ALL-UNNAMED --add-opens java.base/sun.nio.ch=ALL-UNNAMED --
add-opens java.management/sun.management=ALL-UNNAMED --add-opens jdk.management/
com.sun.management.internal=ALL-UNNAMED --add-opens=java.base/java.util=ALL-UNNAMED
--add-exports java.naming/com.sun.jndi.ldap=ALL-UNNAMED --add-opens java.base/
java.text=ALL-UNNAMED --add-opens java.desktop/java.awt.font=ALL-UNNAMED --add-
opens java.naming/com.sun.jndi.toolkit.url=ALL-UNNAMED

REM Delete old build and installation workspaces
RMDIR /S /Q %BUILDWORKSPACE%
RMDIR /S /Q %INSTALLWORKSPACE%

REM Copy source projects into build workspace
XCOPY %PROJECTSDIR% %BUILDWORKSPACE% /E /I

REM Run Ant scripts to build and install the projects
JAVA -cp %ECLIPSE%\startup.jar org.eclipse.equinox.launcher.Main -data
%BUILDWORKSPACE%
-application org.eclipse.ant.core.antRunner -buildfile
%BUILDFILE%
JAVA -cp %ECLIPSE%\startup.jar org.eclipse.equinox.launcher.Main -data
%INSTALLWORKSPACE%
-application org.eclipse.ant.core.antRunner -buildfile %INSTALLFILE%
```

Note: The JAVA commands should be on one line each in your batch file. When running the XCOPY command in the batch file, Windows can return an error displaying “Insufficient memory.” This occurs if the filepaths during the copy process are too long. If this occurs, change your build workspace to a directory with a shorter name or download the Windows 2003 Resource Kit Tool which contains ROBOCOPY, a robust version of the COPY command that can handle longer filepaths.

5. Copy and paste the following Ant script to a file named `ComposerHeadless\HelloWorldBuild\build.xml`. This Ant script updates the HelloWorld JAR definition artifact with the most recent `HelloWorld.jar` file and builds and packages the project.

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

<project name="HelloWorldBuild" default="package-project">

    <target name ="import-project" description="
    Must import a project before updating, building, or installing it">
        <emc.importProject
            dmproject="HelloWorldArtifacts"
            failonerror="true"/>
    </target>

    <target name="update-jardef" depends="import-project" description="
    Update JARDef with most current JAR file">
        <emc.importContent
            dmproject="HelloWorldArtifacts"
            artifactpath="Artifacts/JAR Definitions/hello.jardef"
            contentfile="build_workspace/HelloWorldBOFModule/bin-impl/Hello.jar" />
    </target>
```

```
    <target name="build-project" depends="update-jardef"
        description="Build the project">
        <emc.build
            dmproject="HelloWorldArtifacts"
            failonerror="true"/>
    </target>

    <target name="package-project" depends="build-project" description="
     Package the project into a DAR for installation">
        <delete file="HelloWorld.dar" />
            <emc.dar
                dmproject="HelloWorldArtifacts"
                manifest="bin/dar/default.dardef.artifact"
                dar="HelloWorld.dar" />
    </target>
</project>
```

6. Create an Ant script, `ComposerHeadless\HelloWorldBuild\install.xml`, that installs the HelloWorldArtifacts project to a repository. An example Ant script is shown in the following sample. Modify the strings in bold to meet your environment needs:

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

<project name="headless-install" default="install-project">

    <target name="install-project" description="Install the project to
        the specified repository. dfc.properties must be configured">
        <emc.install
            dar="HelloWorld.dar"
            docbase="MyRepository"
            username="user"
            password="password"
            domain="" />
    </target>

</project>
```

7. Go back to Documentum Composer UI and modify the `HelloWorld.java` file to print out “Goodbye World!”: Save the `HelloWorld.java` file.

```
package com.emc.examples.helloworld.impl;

import com.documentum.fc.client.IDfModule;
import com.emc.examples.helloworld.IHello;


public class HelloWorld implements IHello, IDfModule{
    public void sayHello() {
        System.out.println("Goodbye World!");
    }
}
```

8. Run the `run.bat` file to begin the build and installation process.

9. Run the HelloWorld client in Documentum Composer UI. The Documentum Composer console output displays “Goodbye World!” This verifies that the HelloWorldArtifacts project was updated and installed correctly to a repository.

Congratulations! You have successfully built a BOF module, built a client to access the BOF module, and automated the updating, build, and deployment of the module with Headless Composer.