Generating a J2C bean using the J2C Tools in Rational Application Developer V7.0

Leho Nigul

December 12, 2006

This article demonstrates how the J2C tools available in IBM® Rational® Application Developer 7.0 improve the user experience when generating a J2C bean, and when exposing interaction and connection specification properties associated with the execution of the EIS function as input arguments of the business methods in the J2C bean. This step-by-step guide will demonstrate this capability using a simple IMS example. It will also demonstrate a quick way to create a Web service that uses the functionality provided by the J2C bean to access the IMS.

Introduction

IBM Rational Software Delivery Platform desktop products V7

In support of the new V7 products, please consult the following resources:

- Improve the usability of Web applications with type-ahead input fields using JSF, AJAX, and Web services in Rational Application Developer V7
- JSF and Ajax: Web 2.0 application made easy with Rational Application Developer V7
- What's new in IBM Rational Software Architect V7.0
- What's new in IBM Rational Application Developer V7.0
- What's new in IBM Rational Software Modeler V7.0
- Introduction to IBM Rational Functional Tester 7.0

IBM® Rational® Application Developer V7.0 has offered a substantial improvement in user experience for many application development tools, including the Java™ 2 Platform, Enterprise Edition (J2EE™) Connector (J2C) suite of tools. This article will highlight some of these improvements, while demonstrating how, as a J2EE developer, you can utilize Rational Application Developer V7.0 to generate a J2C bean that allows access to functions provided by backend IBM® IMS™ (Information Management System).

In the current world of service-driven business applications, it is also very important to make sure that Enterprise Information System (EIS) functionality (surfaced through a J2C bean) is easily accessible by various components of the complicated business ecosystem. To demonstrate how such service-driven access to the EIS functionality can be implemented, this article will also show you how to create a Web service that accesses the IMS functionality through a J2C bean.
Required setup

To follow this article in a "hands-on" fashion, you should have the following set up:

- Rational Application Developer V7.0 with J2C tools and IBM® WebSphere® Application Server V6.1 installed on the developer's machine
- The PhoneBook IMS transaction (IVTNO), which is one of the Installation Verification programs that comes with your IMS.
- The sample COBOL program saved on your file system--see the Download section.

Scenario description

This article assumes the following scenario: you have a Phonebook program that runs on IMS. You want to expose the functionality of this program through the Web service. However, you have a requirement that input parameters for your application should be not only the input data required by the back-end system operation, but also IMS interaction property executionTimeout. The article will show how this scenario can be implemented by following these steps:

1. Generate Data Binding classes corresponding to COBOL input and output data structures.
2. Generate a skeleton J2C bean capable of accessing IMS.
3. Create a J2C bean method responsible for invoking PhoneBook operation on the backend IMS.
4. Create a Web service to expose functionality provided by the generated J2C bean.
5. Test the created Web service using the Web Service explorer.

Task 1: Generate Data Binding classes corresponding to COBOL input and output data structures

Data binding classes are one of the key artifacts required to access the functionality of an EIS system. In this case, you have IMS on the backend, where the PhoneBook application is running. This application expects certain input data, and produces certain output data. The tricky issue, though, is the fact that the PhoneBook application expects the data structures to be defined in COBOL language.

A J2C bean, however, operates in the Java universe and expects all the data to be specified as Java classes. Therefore, you need to generate a Java representation of COBOL data structures, which could be used for accessing the functionality of an IMS application. For the purposes of this article, you will generate Data Binding classes required to run the PhoneBook application on IMS.

To do so, you have to perform the following steps.

1. Click New > Other > J2C > CICS/IMS Java Data Binding wizard, as shown in Figure 1.
On the first page of the Data Binding wizard, you have to select the backend system language for the data structure that you want to map to Java.

Currently available mappings are:

- **COBOL to Java**, which allows you to create a Java representation of the data structure defined in COBOL
- **COBOL MPO to Java**, which is similar to COBOL to Java, but allows usage of the COBOL data structure with multiple possible outputs
- **PLI to Java**, which allows you to create a Java representation of the data structure defined in PLI
- **PLI MPO to Java**, which is similar to PLI to Java, but allows usage of the PLI data structure with multiple possible outputs
- **C to Java**, which allows you to create a Java representation of the data structure defined in C

2. For the purposes of your scenario, you select the COBOL to Java mapping, as shown in Figure 2.
   
   Also select the COBOL file on the file system provided with this article and click **Next**.
3. On the next page of the Data Binding wizard, you need to specify the parameters specific to the properties of the back end system (in your case, IMS) on which the COBOL program is running. Since your IMS machine is running IBM® z/OS®, you select it as the target platform, as shown in Figure 3.
4. Once the backend system-specific parameters are specified, you should select the data structure that will be mapped into the Java representation. To do that, click **Query**, select **INPUT-MSG** data structure, and click **Next**, as shown in Figure 4.
5. On the last page of the Data Binding wizard, you have to specify the project and package where you want to generate INPUTMSG java class, which represents the INPUT_MSG COBOL data structure in the Java universe. In this example, generate that class in IMSJavaProject project, as shown in Figure 5.
At this point, you have a Java class that represents the input data structure for the IMS program. However, you also have to generate the class that represents output data structure of the IMS program. To do this, repeat the previous 5 steps with the exception that you specify OUTPUT-MSG as the desired COBOL data structure and generate OUTPUTMSG class into the same project and package into which you generated INPUTMSG class.

After you follow these steps you will have the following two files, shown in Figure 6, in IMSJavaProject.

**Figure 6. IMS Java project structure**

At this point you come to the end of Task 1. You now have Java classes in your workspace that provide Java representation of the data structures required to run the PhoneBook operation on the backend IMS.

Now you need to produce some code that knows how to connect to the IMS. Task 2 shows you how.
Task 2: Generate a skeleton J2C bean capable of accessing IMS

One of the most important tasks that the J2C bean should be able to do is to know how to connect to the specific backend system.

1. Bring up the J2C Java Bean wizard shown in Figure 7 and see what information you should provide to generate a skeleton J2C bean.

Figure 7. New J2C Wizard

On the first page of the wizard, you see a list of adapters that can be used to generate a J2C bean. Rational Application Developer V7.0 ships with 2 IMS and 2 CICS (for 1.0 and 1.5 versions of J2C Architecture specifications) adapters. For the purposes of this article, you will use the IMS adapter (for 1.5 version of the J2C specification), which is shipped with Rational Application Developer V7.0.

2. When you select the IMS adapter, as shown in Figure 8, you see a graphical representation of the scenario for which the code will be generated. The J2C wizard generates a J2C bean, which will have enough information to access IMS through the specific IMS adapter. Also, you can later use the generated J2C bean to generate such J2EE artifacts as Session Enterprise JavaBeans™ (EJB™), JavaServer Pages™ (JSP™), and Web services. Later in this article, you will create a Web service that uses the generated J2C bean.
Figure 8. Select IMS adapter

3. If the selected resource adapter is not in the workspace already, it will be imported into the current workspace after you click the Next button. During the import you might see the progress indicator shown in Figure 9.
The next page of the wizard gives you the ability to input the information that will be used to connect to the underlying EIS system using the chosen J2C resource adapter. This information is backend-system specific. There are three ways that this connection information can be specified:

- Providing managed connection information
- Providing non-managed connection information
- Providing a combination of both

What are the differences between these three?

- **Managed connection** means that the application server manages the lifecycle of the connection (3-tier model). Managed connection is specified by the Java™ Naming and Directory Interface (JNDI) name of the managed connection factory defined on the application server. The managed connection factory is EIS adapter-specific, and contains all of the required connection information. Managed connection implies that the resource adapter is installed directly on the server and, therefore, the JNDI name of the managed connection factory is visible to all EARs (enterprise application archive files) installed on the application server.
• **Non-managed connection** implies a two-tier model in which the application communicates directly to the EIS and manages all the connections itself. Therefore, all the connection information in this setup is encapsulated within the application. In the case of a J2C bean, this means that all of the connection information is specified with doclet tags in the generated J2C bean.

• A **Combination** of these approaches means that the client will first try to fetch a managed connection factory from the server using the provided JNDI name and, if this is not successful, it will use information defined for the non-managed case.

Best practices for J2C development include using a managed connection, because it allows you to leverage capabilities of the application server and to develop complex multi-tier systems. It also provides better resource re-use because connection factories defined on the server can also be used by other applications.

J2C tools allow for a convenient way to define a new (or select an existing) managed connection factory on the server.

4. To do that, you specify the JNDI resource name for the managed connection factory that you want to create and click **New**, as shown in Figures 10 and 11.

**Figure 10. Create a new managed connection factory**
As mentioned before, a managed connection implies that the corresponding resource adapter is installed on the server in stand-alone mode (in other words, it is visible to all EARs installed on this server). If the resource adapter is not installed on the server instance that you specified, the J2C tools will automatically install a resource adapter on the server in the process of creating the managed connection factory.

**Figure 11. Select server instance**

![JNDI Lookup Wizard](image)

Once the server instance is specified, you have to specify the values for IMS specific properties that are required for successful connection to your IMS machine.

5. After you click **Finish** button J2C Tools will try to create specified managed connection factory on the server instance, as shown in Figure 12.

If the selected WebSphere v6.1 instance is already started, the managed connection factory will be created right away. If the server is not started, the creation of the managed connection factory will be postponed until the next time that the server is started and its configuration information is republished. This is a significant usability improvement in comparison to prior versions of the J2C Tools in Rational Application Developer, when the startup of the server was mandated.
6. Now you are ready to proceed to the next page of the J2C wizard, on which you need to specify the project and package where the code for the J2C bean will be generated, as well as the actual names for the J2C bean implementation and interface files. For the purposes of this article, you will generate the J2C bean in the same project where you generated the Data Binding classes. It will be best if you use the names shown in Figure 13 to be consistent with the naming conventions used later in the article.
7. After you click **Finish**, the J2C wizard generates code that is able to connect to the IMS. Let's look at how this information is presented in the generated J2C bean. If you open the generated J2C bean, you will see that the JNDI name of the managed connection factory is specified in the doclet tag within the code.

**Figure 14. JNDI name in the doclet tag**
Generated code modifications

It is useful to know that even after J2C bean code has been generated, you can still do some modifications to the generated code by modifying the values of the doclet tags. For example, if you want to change the JNDI name, just change the value of the jndi-name attribute of the @j2c.connectionFactory doclet tag. The body of the J2C bean regenerates for you automatically when you save your change.

8. Scroll down and you will also see that the doclet annotation builder used the information specified in the doclet tag to create the snippet of code that will be responsible for connection to the IMS, as shown in Figure 15.

Figure 15. JNDI name in the generated code

```java
/**
 * @generated
 */
protected void initializeBinding() throws ResourceException {
    String jndiName = "MyDefaultJNDIName";
    JNDIConnection connection = null;
    try {
        connection = Naming.lookup("java:comp/env/" + jndiName);
        if (connection == null)
            throw new javax.naming.NamingException();
    } catch (javax.naming.NamingException e) {
        try {
            if (connection == null)
                throw new javax.naming.NamingException();
        } catch (javax.naming.NamingException e) {
            throw new ResourceException("error message");
        }
    }
    setConnectionFactory(connection);
    ConnectionFactory cf = new com.ibm.connector2.jms.jms.IBMConnectionFactory();
    setConnectionSpec(cf);
}
```

9. Now, start the instance of the WebSphere v6.1 server which you specified while creating the managed connection factory, and open the Administration Console for this server, you will see the proper managed connection factory defined there.
At this point, you have the J2C bean that knows how to connect to the IMS, and you have the Data Binding classes that represent the data required to run the PhoneBook IMS application. In the next Task, you will see how you can define the Java method in the J2C bean, that will be responsible for invoking the PhoneBook operation on IMS.

**Task 3: Create the J2C bean method responsible for invoking the PhoneBook operation on the backend system**

1. To add a new method to the J2C bean, open the J2C bean implementation Java class, right click anywhere in the source of the Java class, and select Source > Add Method to J2C bean, as shown in Figure 17.

The other way to add J2C method is using the corresponding J2C snippet from the Snippets view.
2. Once the New Java Method dialog comes up, click **Add** to add a new method, as shown in Figure 18.
3. There are number of things you need to specify to define a Java method that will correspond to the backend operation of the EIS system. First, you have to specify the name of the method, as shown in Figure 19.
4. Second, you need to specify the input and output types for the method. These types are the Data Binding types that you created in the Task 1. Therefore, you just need to use the **Browse** button to select these classes, as shown in Figure 20.
5. Once you specify the input and output types, click the **Next** button.
6. On this page of the wizard, you can specify the interaction and connection specification properties, associated with the execution of the EIS function, that you want to expose as the input arguments of the business methods in the J2C bean. For the purpose of this article, select to expose the `executionTimeout` interaction specification property, as shown in Figure 21.
Figure 21. New method

When you select the corresponding property, you can see the change in the method signature at the top of the page. The ability to specify this information through the UI is one of the many improvements in the J2C tooling of Rational Application Developer V7.0 in comparison to previous versions. Note that the UI field corresponding to the `executionTimeout` property in the New Java Method dialog is shaded. This is because the hardcoded value for this interaction property cannot be provided now, since the value is provided through the argument of the Java method.

7. For the purposes of this article, define only one business method for J2C bean. Therefore, you can click **Finish** in the New Java Method dialog.
8. If you now open the J2C bean implementation file, you will see the new method shown in Figure 23 there. Note that the message signature is the same as it was outlined in the New Method Dialog.

**Figure 22. New method**

![New Java Method dialog box](image)

**Figure 23. New method**

```java
/**
 * @InteractionSpec class="com.ibm.connector2.ims.ims.IMSInteractionSpec"
 * @InteractionSpec property name="executionTimeout" argumentName="myExecutionTimeout"
 * @generated
 */

public static OUTPUTMSG sottPhoneBook(String arg1, int myExecutionTimeout) throws WorkflowException {
    String argv1 = WorkflowUtil.argv1(arg1);
    try {
        IMSTransactionSpec ts = new IMSTransactionSpec();
        int i = Integer.parseInt(argv1); 
        ts.setExecutionTimeout(myExecutionTimeout);
        ..
        return output;
    } catch (ParseException e) {
        ..
    }
}
```

At this point you have a functional J2C bean that is able to access the IMS and retrieve the PhoneBook entries.

The next Task shows how you can expose this functionality through the Web service.
Task 4: Create a Web service to expose functionality provided by the generated J2C bean

Before going through the steps specified in this task, start the WebSphere v6.1 server if it is not already started.

J2C Tools provide an easy way to generate such deployable artifacts as WebServices, JSPs and Session beans, all of which can be used to test the functionality of the J2C bean. For the purposes of this article, you will generate a Web service that will be able to access the PhoneBook application through the use of a J2C bean.

1. From the Select a wizard screen, select the **Web Page, Web Service, or EJB from J2C bean** wizard under the J2C category, as shown in figure 24.

   **Figure 24. Open J2C deployment wizard**

2. On the first page of this wizard, you need to specify for which J2C bean you want to generate the deployable artifacts, as shown in Figure 25.
Figure 25. Select J2C bean to generate the deployment artifacts

3. On the next page, select to create a Web service, as shown in Figure 26.
On the next page you will need to specify (or create a new) Dynamic Web project where the Web service will be generated.

It is also advisable to use Resource References to access the resources defined on the server. By using the resource reference you will make sure that real JNDI name of your resource is not hard coded anywhere in your code. The resource reference is defined at the deployment descriptor of your application, and can be changed to point to different "real" resources, depending on to which server you are planning to deploy your application. If you specify the name of the resource reference in this page of the wizard, the J2C tools will create the resource reference and update the application descriptor for us automatically.

4. By clicking **Finish**, you generate a Web service with default settings, as shown in Figure 27. **Note:** If you want to create a Web Service with some non-default settings, use the Web Services wizard from Web Service Tools to generate a bottom-up Web service (Web Service from Java Bean) using that set of tools.
5. Look at the content of the Web project and you will see the Web service artifacts generated there, as shown in Figure 28.

Figure 28. Web Service artifacts in the Web project
Task 5 shows you how to test the Web service you created.

**Task 5: Test the created Web service using the Web Services explorer**

Now you can test the generated Web service on the WebSphere 6.1 server using the Web Services explorer.

1. First, you have to deploy an EAR that contains your Web project to the WebSphere 6.1 Server, as shown in Figure 29.

   **Figure 29. Add EAR to the server**

   ![Add EAR to the server](image)

2. Now, right-click the wsdl file corresponding to the created Web service, then select **Web Services > Test with Web Services Explorer**, as shown in Figure 30.
3. In the Web Service explorer, specify to run the `runPhoneBook` method, as shown in Figure 31.
4. Next, provide the inputs shown in Figure 32 to this method. **Note:** specify 1000 milliseconds as the value for the `executionTimeout` property. This should give your system more than enough time to execute the PhoneBook operation.
5. You should get a response that looks similar to that shown in Figure 33.

**Figure 33. New method**
This proves that your program is working correctly. You obtained the proper data from the PhoneBook application on the IMS.

You also can make sure that the value of `myExecutionTimeout` that you specify on the input is actually used by backend system.

6. To do this, set the value of `myExecutionTimeout` to `1`. It is very likely that your transaction takes more time than `1` millisecond to run, so you should get an exception. Now run the Web Service explorer again and you should see something similar to Figure 34 in the result frame. This means that some exception has occurred.

   **Figure 34. New method**

   ![Image of status message]
   There is nothing to be displayed in the form view. Please switch to the source view for the SOAP request and response.

7. If you choose to see the source, you will see an exception very similar to that in Listing 1 in the response section. This exception indicates that the system has not been able to finish the execution within `1` millisecond.

   **Listing 1. Exception**

   ```java
   javax.resource.spi.EISSystemException: ICO0081E: 
   com.ibm.connector2.ims.ico.IMSTCPIMManagedConnection@1b521b52.
   processOutputOTMAMsg(byte[], IMSInteractionSpec, int) error. Execution timeout has occurred for this interaction.
   The executionTimeout value specified was [1] milliseconds.
   The value used by IMS Connect was [10] milliseconds.
   ```

This means that you have a fully functional Web service that can not only invoke functions on the backend IMS, but also pass values corresponding to different IMS interaction properties.

**Conclusion**

This article showed you how the J2C tools provided in Rational Application Developer V7.0 can be successfully used to rapidly create a J2C bean that is able to access the functionality of the backend IMS. It also demonstrated how this functionality can be quickly exposed to other application and business processes through the Web service mechanism.
### Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cobol sample</td>
<td>Ex01.zip</td>
<td>1KB</td>
</tr>
</tbody>
</table>
Related topics

- Build your next development project with IBM trial software, available for download directly from developerWorks.
- Download a free trial version of Rational Application Developer

© Copyright IBM Corporation 2006
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)