Getting Concurrent with Java

Janette Lockhart
Impac

Introduction

Java Concurrent Programs are a great way to begin including Java programming in your development skill set. There are several reasons that your organization may want to begin utilizing Java. One reason is the Java language itself. Java is a powerful object-oriented programming language which provides classes that make many routine programming tasks very simple – such as file manipulation, xml parsing and more. Another reason is that the E-Business Suite continues to increasingly embrace Java. There are a growing number of public Java APIs in the E-Business Suite – such as APIs for integrating data. Another example is the Java APIs that are provided with XML Publisher. Also, most of the E-Business Suite web pages are built using the OA Framework, which is a Java and XML based technology stack.

This paper will demonstrate how to build and test a Java Concurrent Program using JDeveloper, as well as how to register, and run the program in the E-Business Suite. The examples will show how to call XML Publisher APIs from a Java concurrent program. Additionally, we will leverage OA Framework BC4J objects for use as the XML Publisher xml data source. The examples in the paper were built using Release 12 and JDeveloper 10g with the OA Extension. However, except for some behavioral differences between the JDeveloper versions, these examples will work in Release 11.5.10 as well.

Getting started with JDeveloper and the OA Extension

Oracle has created a special build of JDeveloper with the OA Extension which can be downloaded as a patch from Metalink. The OA Extension "customizes" the JDeveloper environment by adding and modifying files and changing some default settings. Through 11.5.10 of the E-Business Suite, the JDeveloper version is 9.0.3. The OA Extension is not available with JDeveloper 10g until Release 12 of the E-Business Suite. Furthermore, there is a different version of the JDeveloper OA Extension for each ATG patch level within 11i and R12. See Note 416708.1 on Metalink to determine the correct patch number which delivers the JDeveloper version appropriate for your ATG patch level. View the README that accompanies the patch for installation instructions. The setup is very simple. Primarily, you are just unzipping files into a directory and setting a user environment variable on your personal computer.

Preparing Your JDeveloper Environment

We need to download some of Oracle's class files that are specific to concurrent programs. Login to your application server and set your environment. Change directories to $JAVA_TOP/oracle/apps/fnd. Now zip the sub-directory named "cp" by using the following command: zip –r cp cp

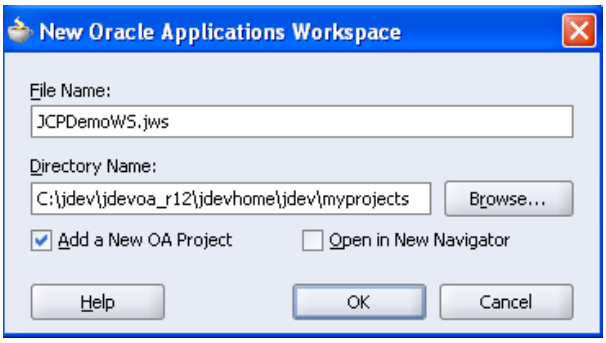
The "-r" option will recursively zip the contents of all sub-directories within the cp directory. The resulting zip file will be named cp.zip. Download the cp.zip file to your computer and unzip it into your <jdev_install>\jdevhome\jdev\myclasses\oracle\apps\fnd directory.

If you are trying this example in 11.5.10, then also repeat this step for the $JAVA_TOP/oracle/apps/jtf directory.

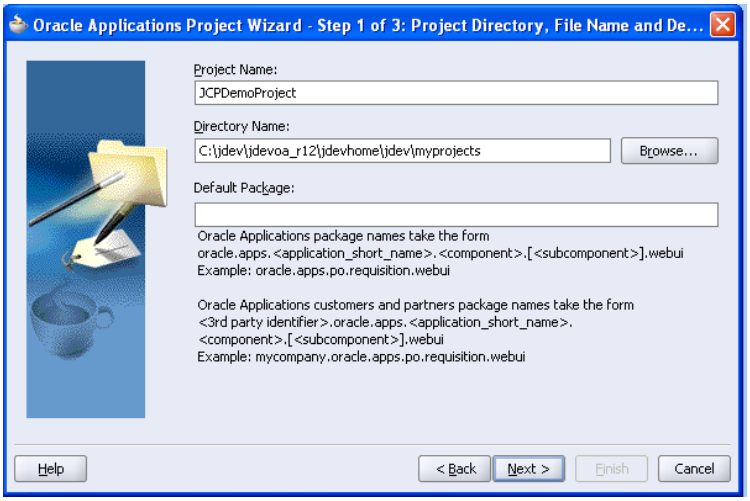
We also need to download the DBC file for the environment. Change directories to $FND_SECURE and download the .dbc file that is in that directory. Put the file into your <jdev_install>\jdevbin\oaext\dbc_files\secure directory. In 11.5.10, this would be your <jdev_install>\jdev\dbc_files\secure directory.

Setting Up an OA Workspace and OA Project

In JDeveloper's Applications Navigator (System Navigator in 11.5.10), right-click the Applications node and select "New OA Workspace". In the New OA Workspace window, enter a meaningful file name for your workspace. The filename should end with ".jws". Be sure that "Add a New OA Project" is checked. Click the OK button.

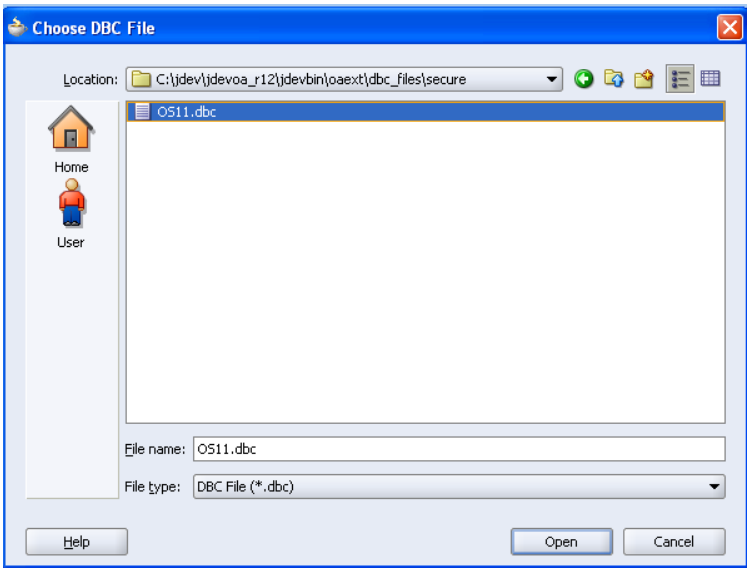


In the Project Wizard Step 1, enter a meaningful Project Name and click Next.

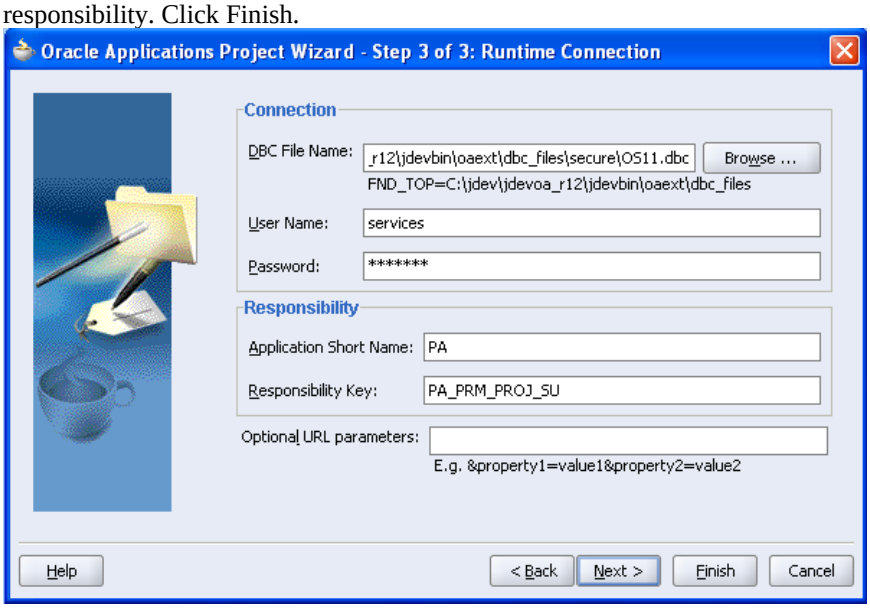


In Step 2 of the wizard, click Next without making any changes.

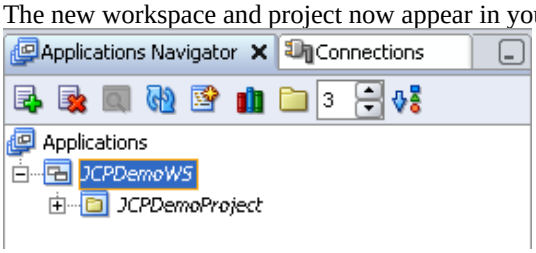
In Step 3 of the wizard, browse the find the dbc file that you copied from your middle tier in the previous section and click Open.



Still on Step 3 of the Project wizard, enter your E-Business Suite user name and password. Also enter the application short name and responsibility key for a responsibility that is assigned to the user. Since our example will later include some BC4J objects from Oracle Projects, the screenshot below is referencing the Projects Super User responsibility. Click Finish.

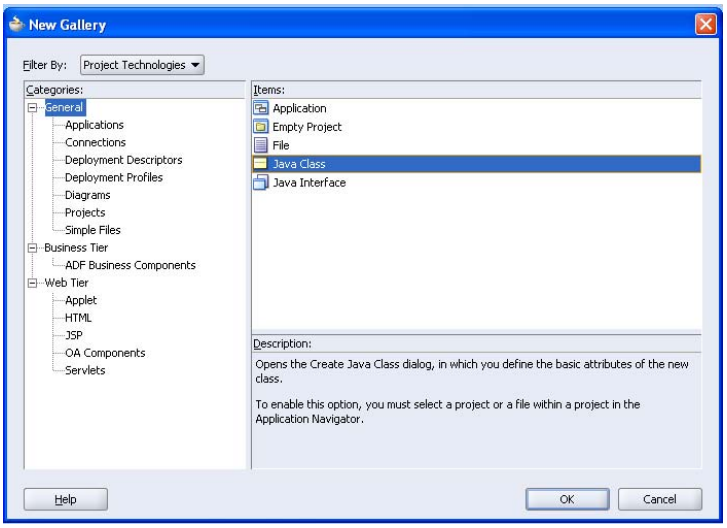


The new workspace and project now appear in your Applications Navigator.

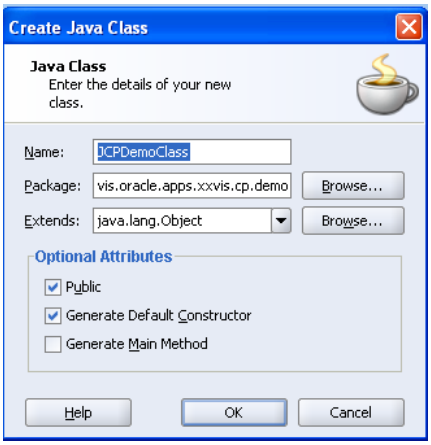


Developing Your First Java Concurrent Program

Right-click on JCPDemoProject and select "New…" from the context menu. In the New Gallery, select the "General" node and then "Java Class" from the Items list. Click OK.



In the Create Java Class window, enter JCPDemoClass for the class name and enter vis.oracle.appps.xxvis.cp.demo for the package. Instead of using “vis” and “xxvis”, use your company identifier instead of “vis” and your custom application short name instead of “xxvis”. Make the same substitutions wherever you see “vis” and “xxvis” in the remainder of the examples. Leave the rest of the items at their default values and click OK.



A Code Editor window will open with generated code for your new class. We will make three changes to the generated code.

First, add the following import statements below the package declaration:

```
import oracle.apps.fnd.cp.request.ReqCompletion;
import oracle.apps.fnd.cp.request.JavaConcurrentProgram;
import oracle.apps.fnd.cp.request.CpContext;
```

Then, add “implements JavaConcurrentProgram” following the “public class JCPDemoClass” declaration:

```
public class JCPDemoClass implements JavaConcurrentProgram
```

Finally, add the following method immediately after the class declaration and opening bracket:

```
    public void runProgram(CpContext ctx)
    {
     ctx.getLogFile().writeln("Starting to run Java concurrent program", 0);
     ctx.getReqCompletion().setCompletion(ReqCompletion.NORMAL, "");
    }
```

The final code will look like this.

```
package vis.oracle.apps.xxvis.cp.demo;

import oracle.apps.fnd.cp.request.ReqCompletion;
import oracle.apps.fnd.cp.request.JavaConcurrentProgram;
import oracle.apps.fnd.cp.request.CpContext;

public class JCPDemoClass implements JavaConcurrentProgram
{
    public void runProgram(CpContext ctx)
```



```
    {
     ctx.getLogFile().writeln("Starting to run Java concurrent program", 0);
     ctx.getReqCompletion().setCompletion(ReqCompletion.NORMAL, "");
    }

    public JCPDemoClass() {
    }
}
```

Right-click anywhere inside the Code Editor window and choose "Make" to compile the class. Fix any compilation errors and re-compile until there are no more errors.

Code Explanation

We have just created a class named JCPDemoClass. Our class resides in a directory structure named vis/oracle/apps/xxvis/cp/demo. This is the package name that is referenced on the first line of the class. After compiling our class, we will find that this directory structure has been created for us in our myprojects and myclasses folders. Source code (.java files) are located in the myprojects directory and compiled code (.class files) are located in the myclasses directory. We are following Oracle OA Framework development standards in our package naming. Since the examples were setup in a Vision environment, we are pretending that our company identifier is "vis" and that our custom application short name is "xxvis".

The class has one method named runProgram and a constructor named JCPDemoClass. Constructors always have the same name as the class. If you are new to Java, think of a class as a pl/sql package and methods as the functions and procedures within the package. The import statements refer to various Oracle classes that we will be using in our own class. These classes were contained in the cp.zip file that we zipped , downloaded, and unzipped earlier.

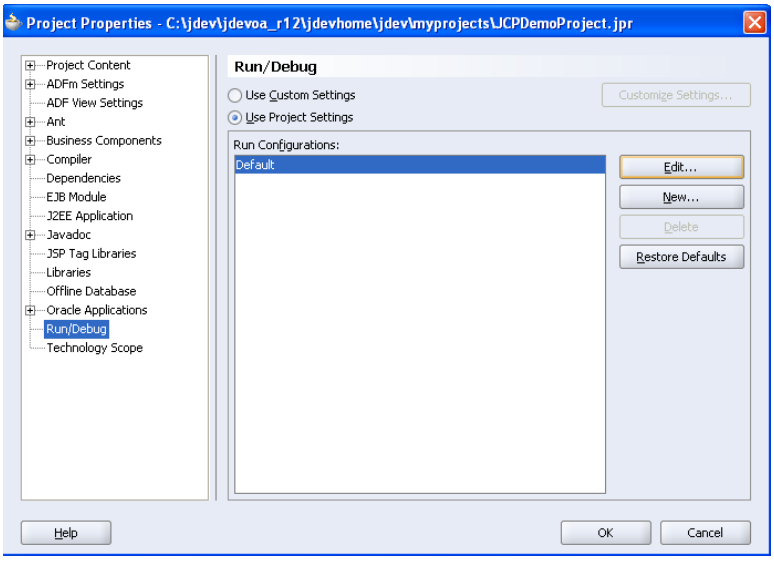
We have said that our class "implements JavaConcurrentProgram". JavaConcurrentProgram is an interface. The concept of an interface in Java is entirely different from the way a traditional Oracle E-Business Suite developer thinks of an interface. In Java, an interface provides a template for any classes that implement it. The classes that implement the interface promise to provide the actual code for the methods in the template. In this case, our class must promise to provide the code for the runProgram method.

Consequently, our runProgram method must have the exact signature "public void runProgram(CpContext ctx)", since this is how the interface defines it. We are then free to add whatever code necessary in the runProgram method to accomplish the requirements for our concurrent program. The CpContext that is passed into the runProgram method provides us with the "context" for our concurrent program and gives us access to objects such as the concurrent request log file and output files, profile values, a JDBC connection, and many other objects. In the code that we added to runProgram, we've used the context to get a handle on the log file then write some text to the log. We also set the completion status of our request to Normal.

Running the Concurrent Program from JDeveloper

Since we are in the early stages of developing our program, it would be convenient to be able to test the program without having to deploy it to the application server and register it in AOL. We can easily do that from JDeveloper after first completing a few setup steps. First we need to set the Java Options so that the JVM will know where to find the dbc file, concurrent request log file, and concurrent request output file on the desktop computer. In the Applications Navigator, right-click on your JCPDemoProject and choose "Project Properties" from the context menu. (In 11.5.10, this would be Project Settings.)

Choose Run/Debug from Project Properties and then click the "Edit" button for the Default run configuration.

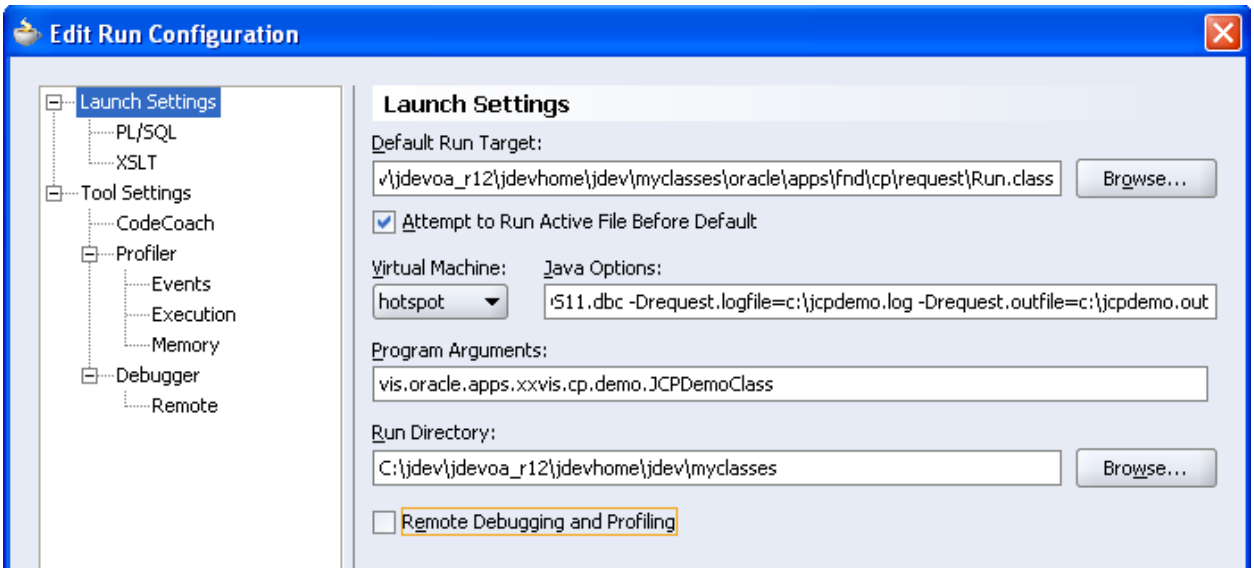


In the Launch Settings, for "Default Run Target", browse to find <jdev_install>\myclasses\oracle\apps\fnd\cp \request\Run.class. This was one of the classes downloaded in the cp.zip file from the middle tier.

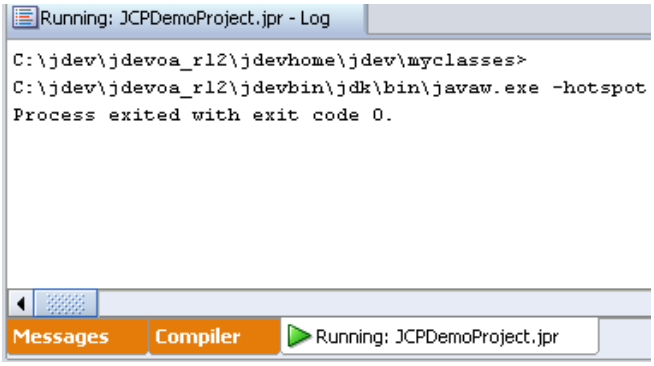
In the Java Options item, go to the end of the existing entries and add the following:

-Ddbcfile=<jdev_install>\jdevhome\jdev\dbc_files\secure\OS11.dbc -Drequest.logfile=c:\jcpdemo.log -Drequest.outfile=c:\jcpdemo.out

Change the dbcfile argument to reference the DBC file that you downloaded in a previous step. Change the directory location and name of the log and out files to be whatever you wish. In Program arguments, enter the name of your class, vis.oracle.apps.xxvis.cp.demo.JCPDemoClass. Click OK.



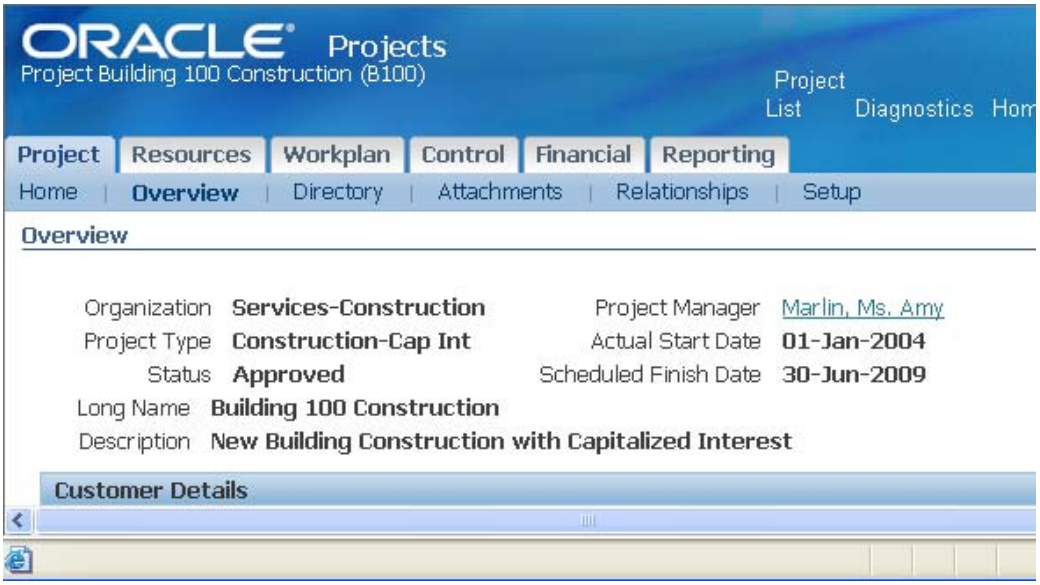
Now, in the Applications Navigator, you can right-click the JCPDemoProject and select “Run” from the context menu. Shortly, you should see a successful completion message, “Process exited with exit code 0”, in the output window:



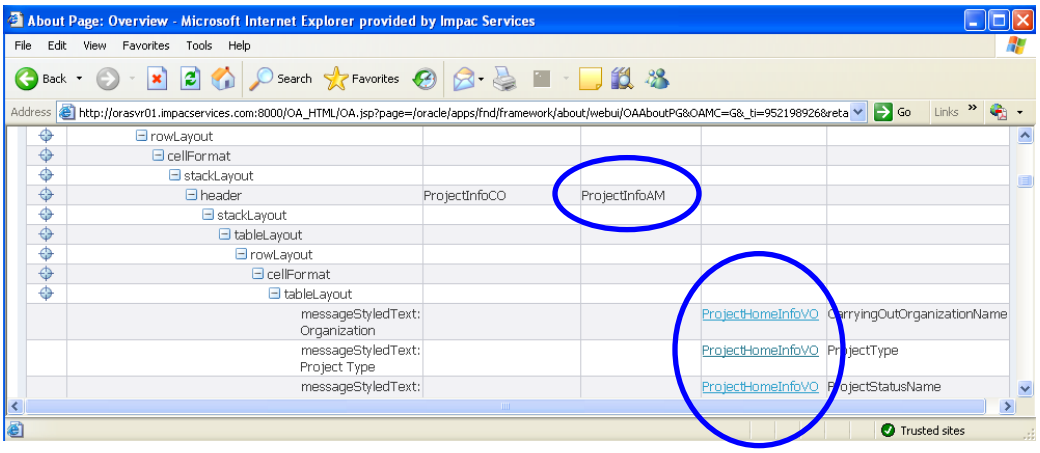
If you go look for the log file that you specified in the run parameters, you should find that it has been created and contains the text “Starting to run Java concurrent program”.

Leveraging OA Framework BC4J Objects in a Java Concurrent Program

Our requirement for the demo Java Concurrent Program is to ultimately produce an XML Publisher report based on the data that is displayed in the Oracle Projects Project Overview page, as seen in the screenshot below.



If we use the “About this Page” link we will learn that the project level data is coming from the ProjectHomeInfoVO BC4J View Object, which is contained in the ProjectInfoAM BC4J Application Module. If you’re not familiar with BC4J, then a quick explanation is in order. An Application Module can be thought of as a container that gives access to other BC4J objects that have been assigned to the container (application module). A View Object defines how the database is queried for a specific set of data.



Let's instantiate the Application Module and the ProjectHomeInfoVO view object in our concurrent program. Then, we can call the easy to use writeXml method on the View Object to generate our XML data. To do so, we'll create a new method in our class, named buildXml. We'll pass the CpContext as an argument to our new method.

The first thing that our new method will do is create an instance of an Application Module Factory. We will then use the AM Factory to create an instance of the ProjectInfoHomeAM application module. Next, we'll use the application module's findViewObject method to get an instance of the ProjectHomeInfoVO view object. Then, we'll add a where clause to the view object's query to specify the project number we are interested in. At this point in the example, we're going to simply hard-code a project number. The complete code, found in the Appendix at the end of this paper, shows how to handle concurrent request parameters. To accomplish the steps I just described, our new buildXml method will have the following lines of code.

```
    private void buildXml(CpContext ctx)
    {
      String amName = "oracle.apps.pa.project.server.ProjectInfoAM";
      String voName = "ProjectHomeInfoVO";
      OAApplicationModuleFactory amFactory =
              new OAApplicationModuleFactory();
      OAApplicationModule am =
              amFactory.createRootOAApplicationModule(ctx, amName);
      OAViewObjectImpl vo =
              (OAViewObjectImpl)am.findViewObject(voName);
      vo.addWhereClause("project_number = :1");
      vo.setWhereClauseParam(0, "B100"); //temporarily hard-coded
      vo.executeQuery();
    }
```

Now that we've queried the View Object for the project that we're interested in, we're ready to generate the XML. There is a method available for all View Objects named writeXML. This method will generate an XML structure for the view object and, if any view links are defined on the View Object, is also capable of querying the related View Objects as well. For example, if the ProjectHomeInfoVO were declaratively linked to another VO for the project's tasks, then we could produce an XML document that also includes the data from the task VO. Obviously this is a very powerful and convenient way to generate an XML document. Our goal is to generate the XML and write it to our concurrent request output file. To accomplish this, we need to add the following lines of code to the buildXml method.

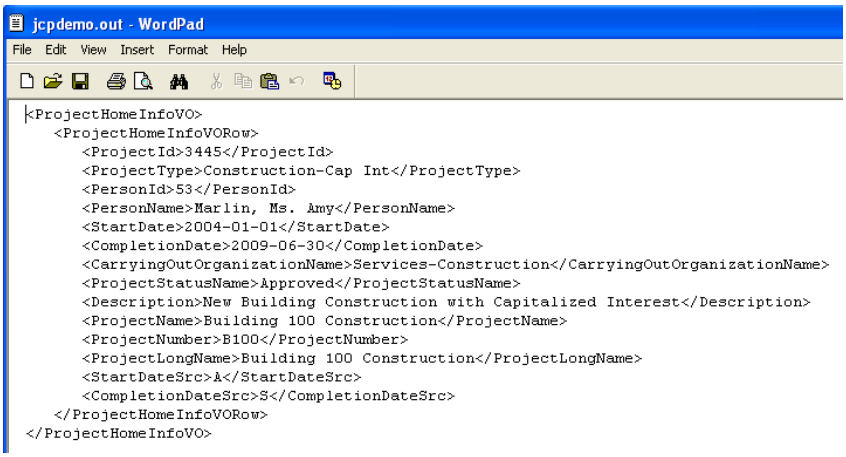
```
XMLNode root = (XMLNode)vo.writeXML(0, XMLInterface.XML_OPT_ALL_ROWS);
BlobDomain result = new BlobDomain();
try
{
```

```
  root.print((OutputStream)result.getBinaryOutputStream());
  ctx.getOutFile().writeln(result.toString());
}
```

The writeXml method returns the generated XML into an XMLNode object. We are then writing the XMLNode into the output stream of a BlobDomain. Finally, we are converting the BlobDomain into a string and writing that to the concurrent request output file.

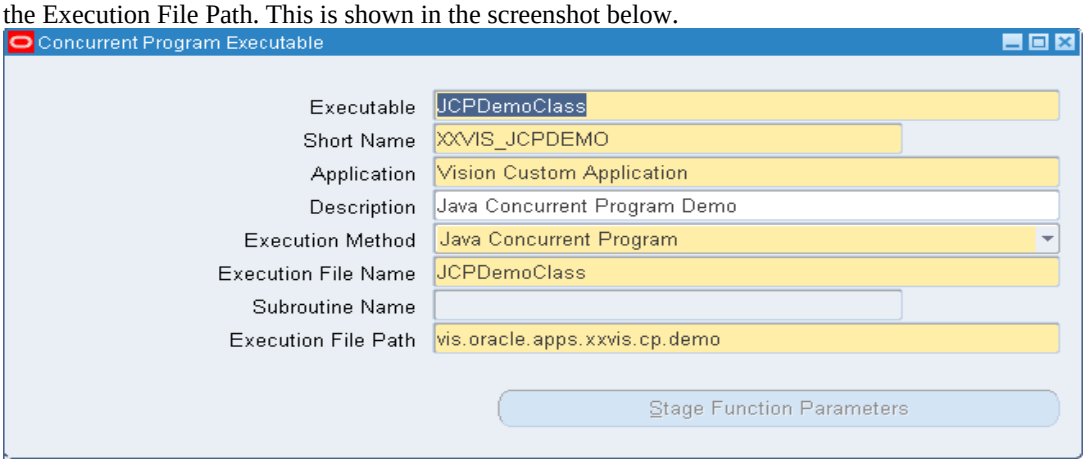
We also require some more code in addition to that shown above in order to "catch" the possible exceptions that these operations might raise or "throw" as we say in Java. This is included in the complete program code found in the Appendix.

If we run our program at this point, we could then open our concurrent request output file and find the generated XML. Notice that the root node has the same name as the View Object – "ProjectHomeInfoVO". The XML elements have the same name as the attributes in the View Object.

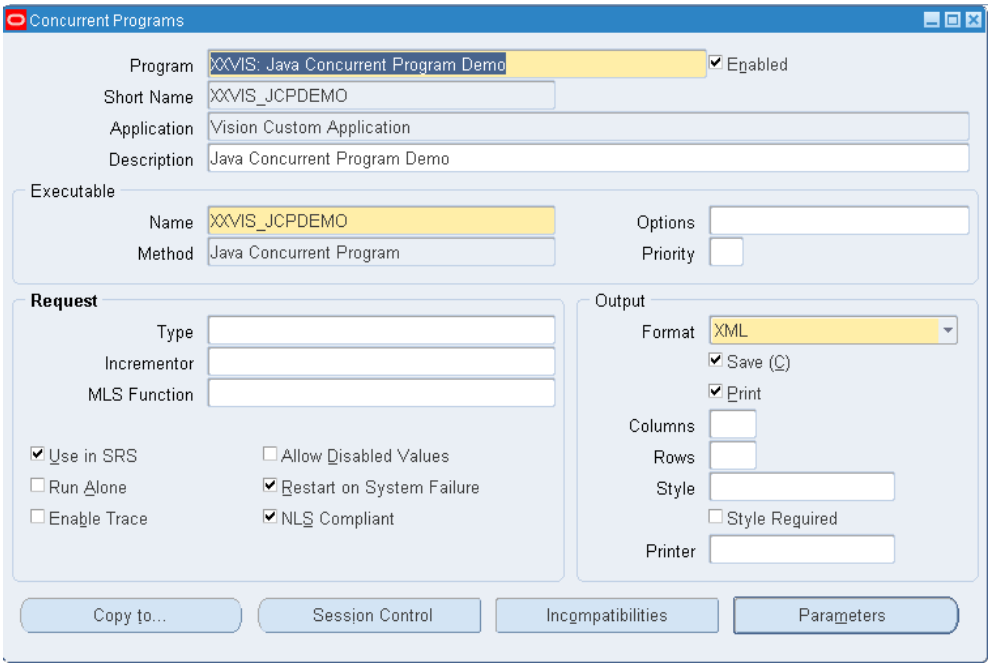


Now, we're ready to deploy our work to the application server. Up to this point, we've only been running the concurrent program from JDeveloper. The first step is to copy your java file (JCPDemoClass.java) to application server's $JAVA_TOP directory. Create a directory structure that is the same as your package structure. In our example, that would be $JAVA_TOP/vis/oracle/apps/xxvis/cp/demo. After copying the .java file, compile it by using the javac command: javac JCPDemoClass.java. That will create your JCPDemoClass.class executable file.

The next step is to register the concurrent executable in AOL. That is simply a matter of using the "Java Concurrent Program" execution method and pointing to the class name as the Execution File Name and to the package name as the Execution File Path. This is shown in the screenshot below.



Next, we would register our concurrent program, setting the output format to XML. Assign the concurrent program to a request group. The last step would be to register the concurrent program in the XML Publisher responsibility as a data definition and to register an XML Publisher template. After running our concurrent request from the concurrent manager, we'll find that the XML is generated and that the XML Publisher template has been applied!



Calling XML Publisher APIs from a Java Concurrent Program

As a final step in the demo program, we are going to add the requirement that we want to (1) programmatically apply the XML Publisher template and (2) email the resulting PDF output. To do this, we'll use the XML Publisher Bursting API. In fact, in Release 12, bursting comes "out of the box" and we don't necessarily have to write a concurrent program. However, this example is just to illustrate the concepts of calling XML Publisher APIs from a concurrent program. Also, there are many people who are not yet on Release 12 that might find this approach useful.

In order to work with XML Publisher APIs, we need to acquire some additional class files from the middle tier. Zip and download the $JAVA_TOP/oracle/apps/xdo directory and unzip it into your myclasses/oracle/apps directory.

The bursting API uses an XML control file to manage the bursting options. The bursting engine can be used to split the output file based on an XML element. In our example, we'll split the file out by project. The bursting engine can then deliver the documents via the file system, email, as well as other channels. Our control file will send each pdf file as an attachment via email. Here's the control file we'll use:

```
<?xml version="1.0" encoding="UTF-8"?>
<xapi:requestset xmlns:xapi="http://xmlns.oracle.com/oxp/xapi">
<xapi:request select="/ProjectHomeInfoVO/ProjectHomeInfoVORow">
<xapi:delivery>
<xapi:email server="vansmtp.impacservices.com" port="25"
from="jlockhart@impacservices.com" reply-to ="jlockhart@impacservices.com">
<xapi:message id="123" to="jlockhart@impacservices.com" attachment="true"
subject="JCP Demo Test">Please review the attached document.</xapi:message>
</xapi:email>
</xapi:delivery>
<xapi:document output-type="pdf" delivery="123">
```



```
<xapi:template type="rtf" location=
"xdo://XXVIS.XXVIS_JCPDEMO.en.US/?getSource=true" >
</xapi:template>
</xapi:document>
</xapi:request>
</xapi:requestset>
```

The control file indicates that the xml output should be burst into a separate email for each occurrence of ProjectHomeInfoVORow, for each project in this case. The pdf files are being emailed to the same person in this example, however you could easily set dynamic email addresses from the XML output. The location attribute reference to "xdo://XXVIS.XXVIS_JCPDEMO.en.US/?getSource=true" references the template application and template code as defined in the XML Publisher repository. This is a nice way to avoid having to hard-code a path to the template file. We can place the control file into our $CUSTOM_TOP directory and it will be available to reference by our concurrent program.

We're now ready to add a new method to our java concurrent program to call the XML Publisher bursting API.

```
    private void emailOutput(CpContext ctx){
      //get the path to our custom top directory
      String sCustomTop = ctx.getAppsEnvironmentStore().getEnv("XXVIS_TOP");
     //concat the directory path and file name for the bursting control file
      String sCtlFile = sCustomTop + "/jcpdemo_ctl.xml";
      try {
        //get the file name for the conc request output file
        String sXmlFile = ctx.getOutFile().getFileName();

        //call the document processor, passing control file and xml file
        DocumentProcessor dp = new DocumentProcessor(sCtlFile, sXmlFile);
        dp.process();
      }
      catch (Exception e)
      {
        ctx.getLogFile().writeln(e.getMessage(), 0);
        mReqStatus = ReqCompletion.ERROR;
      }
    }
```

We're calling the getAppsEnvironmentStore method on the Context to get the path for $XXVIS_TOP (the custom top directory). We then concatenate that to the file name of the bursting control file. We call get the getOutFile().getFileName() method on the Context to get the request output file name. Those are passed into the Document Processor which uses the control file to burst the XML output and email the resulting PDF files as per the control file instructions.

Conclusion

After walking through this Java Concurrent Program example, I'm sure you'll agree that we were able to accomplish quite a bit of functionality with very little code written. It was simple to create a basic Java Concurrent Program and then test it by running from JDeveloper. Then, we were quickly able to generate an XML data source by instantiating some OA Framework BC4J objects. Next, we registered our concurrent program in AOL, being sure to use an execution method of Java Concurrent Program. Finally, we enhanced our program's functionality by making use of the XML Publisher document processor to "burst" our XML output by project number, apply an XML Publisher template to the burst XML output, and email the resulting PDF files.

Hopefully, you'll be able to apply these concepts and begin experimenting with the power and flexibility of Java Concurrent Programs in the E-Business Suite.

Appendix – Complete Program Code

```
package vis.oracle.apps.xxvis.cp.demo;

import java.io.IOException;
import java.io.OutputStream;

import java.sql.SQLException;

import oracle.apps.fnd.cp.request.CpContext;
import oracle.apps.fnd.cp.request.JavaConcurrentProgram;
import oracle.apps.fnd.cp.request.ReqCompletion;
import oracle.apps.fnd.framework.OAApplicationModule;
import oracle.apps.fnd.framework.OAApplicationModuleFactory;
import oracle.apps.fnd.framework.server.OAViewObjectImpl;

import oracle.apps.fnd.util.NameValueType;
import oracle.apps.fnd.util.ParameterList;

import oracle.jbo.XMLInterface;
import oracle.jbo.domain.BlobDomain;

import oracle.xml.parser.v2.XMLNode;
import oracle.apps.xdo.batch.DocumentProcessor;

public class JCPDemoClass implements JavaConcurrentProgram
{

    //declare an integer to hold the req status
    int mReqStatus;

    public void runProgram(CpContext ctx)
    {
     ctx.getLogFile().writeln("Starting to run Java concurrent program", 0);

     //preset status to Normal completion
     mReqStatus = ReqCompletion.NORMAL;

     //build XML data source
     buildXml(ctx);

     //only email if status is still NORMAL
     if (mReqStatus == ReqCompletion.NORMAL)
     {
         emailOutput(ctx);
     }
     //set request completion
     ctx.getReqCompletion().setCompletion(mReqStatus, "");
    }
    private void buildXml(CpContext ctx)
    {
      //string variables to hold AM Name and VO Name
      String amName = "oracle.apps.pa.project.server.ProjectInfoAM";
      String voName = "ProjectHomeInfoVO";
      //get AM Factory and create root AM
```



```
      OAApplicationModuleFactory amFactory = new
OAApplicationModuleFactory();
      OAApplicationModule am = amFactory.createRootOAApplicationModule(ctx,
amName);

      //find the View Object
      OAViewObjectImpl vo = (OAViewObjectImpl)am.findViewObject(voName);

      //get the Project Number parameter
      String sProjNum = getProjNumParam(ctx);

      //add where clause to VO query and set project number bind variable
      vo.addWhereClause("project_number = :1");
      vo.setWhereClauseParam(0, sProjNum);

      //execute VO query
      vo.executeQuery();

      //get the XML
      XMLNode root = (XMLNode) vo.writeXML(2,XMLInterface.XML_OPT_ALL_ROWS);
      //declare Blob domain
      BlobDomain result = new BlobDomain();
      try
      {
        //write XML to BLOB's output stream
        root.print((OutputStream)result.getBinaryOutputStream());

        //convert Blob to string and write to conc request out file
        ctx.getOutFile().writeln(result.toString());

      }

      catch (SQLException e1)
      {
        ctx.getLogFile().writeln(e1.getMessage(), 0);
        mReqStatus = ReqCompletion.ERROR;
      }
      catch (IOException e2)
      {
        ctx.getLogFile().writeln(e2.getMessage(), 0);
        mReqStatus = ReqCompletion.ERROR;
      }
      finally
      {
        am.remove();
      }
    }

    private String getProjNumParam(CpContext ctx)
    {
      String pNum = null;
      ParameterList pList = ctx.getParameterList();
      while (pList.hasMoreElements())
      {
          NameValueType aNVT = pList.nextParameter();
          if (aNVT.getName().equals("PROJECT_NUMBER")) //token name in conc
pgm parameter
```

```
          {
            pNum =  aNVT.getValue();
            break;
          }

      }
      return pNum;
    }

    private void emailOutput(CpContext ctx)
    {

        //get the path to our custom top directory
        String sCustomTop =
ctx.getAppsEnvironmentStore().getEnv("XXVIS_TOP");
        //concat the directory path and file name for the bursting control
file
        String sCtlFile = sCustomTop + "/jcpdemo_ctl.xml";

        try
        {
          //get the file name for the conc request output file, which has the
XML
          String sXmlFile = ctx.getOutFile().getFileName();

          //call the document processor, passing control file and xml file
          DocumentProcessor dp = new DocumentProcessor(sCtlFile, sXmlFile);
          dp.process();
        }
        catch (Exception e)
        {
          ctx.getLogFile().writeln(e.getMessage(), 0);
          mReqStatus = ReqCompletion.ERROR;
        }

    }

    public JCPDemoClass() {
    }
}
```

