

Innovate 2010 Technical Workshop

IBM Rational Team Concert Extensibility

TW-2197A

Workshop Exercises



© Copyright International Business Machines Corporation, 2010. All rights reserved.

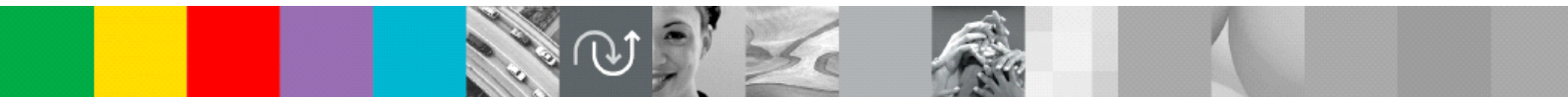
US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

LAB 1	SETTING UP THE IBM® RATIONAL® TEAM CONCERT (RTC) SDK	4
1.1	DOWNLOAD AND UNZIP THE REQUIRED FILES FROM JAZZ.NET	5
1.2	SETUP FOR DEVELOPMENT	7
1.3	SETUP FOR TESTING WITH TOMCAT	11
1.4	TEST CONNECTING THE ECLIPSE DEBUGGER TO TOMCAT	13
1.5	SETUP FOR JETTY BASED SERVER LAUNCH.....	21
1.6	TEST THE JETTY JAZZ SERVER LAUNCH	39
1.7	SETUP JETTY LAUNCH USING THE TOMCAT REPOSITORY	43
1.8	TEST THE TOMCAT JAZZ SERVER LAUNCH.....	46
1.9	SETUP TO DEBUG AN RTC ECLIPSE CLIENT	47
1.10	TEST YOUR RTC CLIENT LAUNCH	54
1.11	SETUP FOR THE REMAINING LABS.....	55
LAB 2	CREATE A SIMPLE BUILD ON STATE CHANGE OPERATION PARTICIPANT	56
2.1	CREATE A BASIC SERVER SIDE SERVICE	57
2.2	LAUNCH THE SERVER FOR DEBUG USING JETTY	66
2.3	LAUNCH AN RTC CLIENT AND CONNECT TO THE SERVER	68
2.4	EDIT THE PROCESS TO USE THE PARTICIPANT	72
2.5	TRIGGER THE PARTICIPANT.....	78
2.6	RENAME BUILD DEFINITION AND TRY AGAIN	85
LAB 3	ADD ERROR HANDLING.....	90
3.1	UNDERSTANDING ERROR HANDLING CODE.....	90
3.2	LAUNCH THE SERVER FOR DEBUG USING JETTY	94
3.3	LAUNCH AN RTC CLIENT AND CONNECT TO THE SERVER	95
3.4	TRIGGER THE PARTICIPANT.....	96
3.5	RENAME BUILD DEFINITION AND TRY AGAIN	97
LAB 4	PARAMETERIZATION	101
4.1	UNDERSTANDING PARAMETERIZATION	101
4.2	LAUNCH THE SERVER FOR DEBUG USING JETTY	112
4.3	LAUNCH AN RTC CLIENT AND CONFIGURE THE PARTICIPANT.....	113
4.4	TRIGGER THE PARTICIPANT.....	118
4.5	CHANGE THE BUILD ID IN THE CONFIGURATION AND TRY AGAIN	120
LAB 5	ADDING AN ASPECT EDITOR	124
5.1	UNDERSTANDING THE ASPECT EDITOR	124
5.2	LAUNCH THE SERVER FOR DEBUG USING JETTY	133
5.3	LAUNCH AN RTC CLIENT AND CONFIGURE THE PARTICIPANT.....	134
5.4	TRIGGER THE PARTICIPANT.....	139
5.5	ADD ANOTHER INSTANCE OF THE FOLLOW-UP ACTION AND TRY AGAIN	141
LAB 6	DEPLOYING THE SERVER SIDE	143
6.1	CREATING A SERVER SIDE FEATURE.....	143
6.2	CREATE THE SERVER UPDATE SITE	151
6.3	DEPLOY THE SERVER SIDE FEATURE	158
6.4	DEPLOY THE CLIENT PLUG-INS.....	162
6.5	TEST THE DEPLOYED PARTICIPANT.....	165
6.6	COMPLETE DEVELOPMENT	171
APPENDIX A.	NOTICES.....	173
APPENDIX B.	TRADEMARKS AND COPYRIGHTS	175

THIS PAGE INTENTIONALLY LEFT BLANK

IBM Software Group



Lab 1 Setting up the IBM® Rational® Team Concert (RTC) SDK



Lab Scenario

You have a new assignment on a team creating RTC extensions. The first thing you need to do is to set up your development environment.

Once you have completed this module, you will be ready to start developing RTC extensions.



In order to complete and get the most out of this workshop, it is recommended that you are already familiar with RTC as a user. Of particular help would be familiarity with work items, build definitions and basic process customization. In addition, you should be familiar with Java programming and debugging using Eclipse. Some familiarity with Eclipse plug-in programming would also be helpful but is not strictly required. There are a number of Eclipse plug-in development tutorials available on the web (for example, <http://www.ibm.com/developerworks/library/os-eclipse-plugindev1/>).



Note that these instructions are written specifically for RTC 2.0.0.2 on Windows®. Please adjust accordingly for different operating systems (primarily the RTC Eclipse client download and the file paths) and RTC versions (downloads).

On jazz.net you will also find various options for testing (server Jetty launch verses Tomcat attached debug) and different types of API usage (for example, plain Java client verses plug-in extensions). Although you will not make use of all these options in this workshop, this setup will enable all these scenarios.



Along with this lab document(s), you should have received or downloaded a file with a name of the form ExtensionsAndIntegrationsLabCodeRepository-yyyymmdd.tar. It is an exported RTC repository. You will import it at the end of this lab.



The repository will contain materials for other related workshops. Those other workshops also have initial labs like this one to set up the environment. You can run through more than one setup lab on the same RTC installation. Just note that some steps may be duplicated, such as the initial download steps or the final step to import the repository.

1.1 Download and Unzip the Required Files from jazz.net

- ___1. Download the Express-C edition zip files.
 - ___a. Go to the RTC 2.0.0.2 all downloads page at <https://jazz.net/downloads/rational-team-concert/releases/2.0.0.2?p=allDownloads>. There may be iFix releases available beyond 2.0.0.2. You can download one of those instead. The file sizes will vary from what is shown below.
 - ___b. Scroll down to the **Express-C** section and download the highlighted files. The first contains the client and server (including the build engine). The second contains the libraries for standalone Java applications. The third contains all the RTC source code.

Express-C

Description	Platform
ZIP	
Client for Eclipse IDE	Windows x86 (475.34 MB)
Client for Eclipse 3.5.x (p2 Install)	All (210.38 MB)
Client for Eclipse 3.5.x (Extension Install)	All (211.01 MB)
Client for Eclipse IDE and Server	Windows x86 (744.48 MB)
Server	Windows x86 (231.74 MB)
	Linux x86 (220.88 MB)
Build System Toolkit	Windows x86 (38.01 MB)
Plain Java Client Libraries	All (20.44 MB)

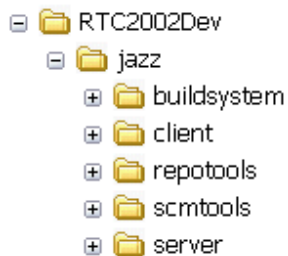
- ___c. Scroll down to the **Source Code** section and download the highlighted file.

Source Code

Description	Platform
ZIP	
Rational Team Concert SDK	All (454.79 MB)

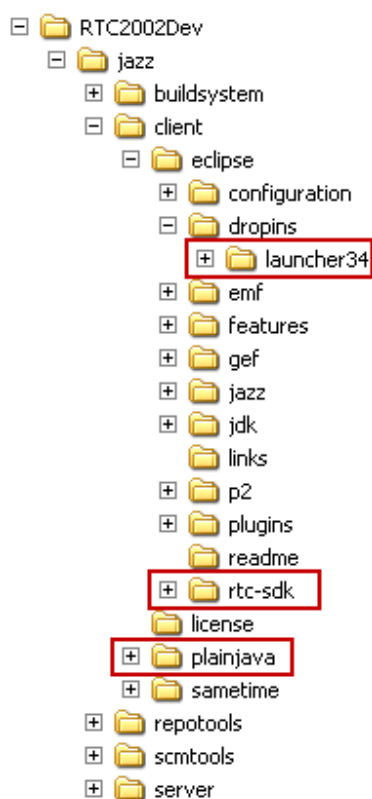
- ___d. Go to the feature based launches wiki page at <https://jazz.net/wiki/bin/view/Main/FeatureBasedLaunches>.
 - ___e. Scroll to the bottom of the page and download the attached launcher34.zip file.
- ___2. Unzip the product files. For this lab, your installation directory will be assumed to be C:\RTC2002Dev.
 - ___a. Unzip the Eclipse IDE and server download into C:\RTC2002Dev.

- ___b. Your C:\RTC2002Dev folder will look pretty standard at this point. Much like setting up a sandbox or demo environment.



- ___3. Add the feature based launches capability to the RTC Eclipse client.
 - ___a. Unzip the feature based launches download into C:\RTC2002Dev\jazz\client\eclipse\dropins. Do not include the MACOSX specific files. Select just the launcher34 folder and extract the folder and its contents into the dropins folder.
- ___4. Unzip the development time files into the RTC Eclipse client folders.
 - ___a. Unzip the RTC SDK zip file into C:\RTC2002Dev\jazz\client\eclipse. This zip file has path lengths longer than 250 characters and may cause trouble for some extractor tools on Windows. One zip extractor tool that works is [7Zip](#).
 - ___b. Unzip the plain java client libraries zip file into C:\RTC2002Dev\jazz\client\plainjava. You or your extraction tool will have to create the plainjava folder. It is not contained in the zip file.

__c. Your C:\RTC2002Dev folder will not look a bit different.




1.2 Setup for Development



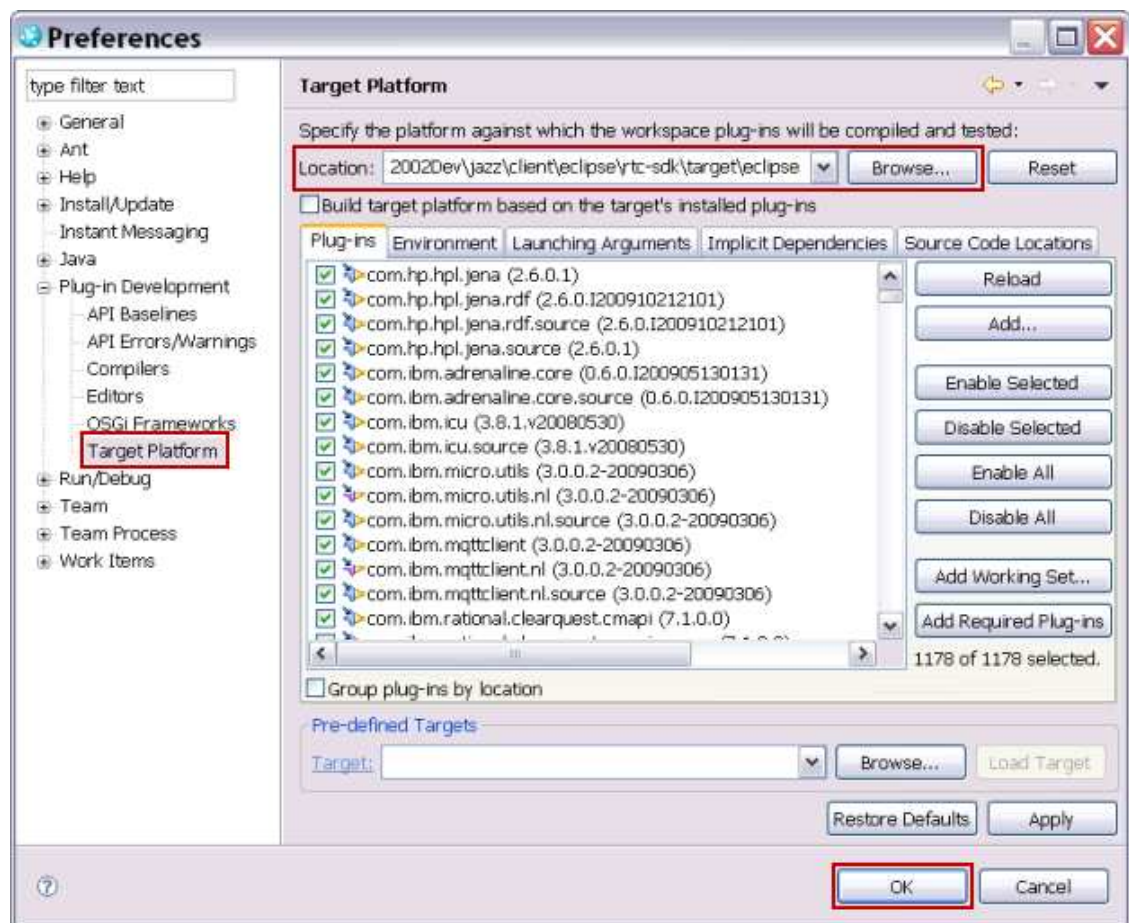
In this section you will setup your RTC Eclipse client for developing RTC plug-ins. This consists of letting Eclipse know what platform (set of Eclipse features and plug-ins) you are developing for, opening the Eclipse perspective designed for plug-in development and letting the Eclipse Java Development Tooling (JDT) know about all the RTC platform source code.



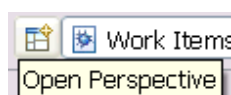
In the next eight sections you will alternate between setting up and testing four particularly useful launch and debug scenarios. Some of these will be used during this workshop.

- __1. Set the target platform.
 - __a. Start the RTC Eclipse client (C:\RTC2002Dev\jazz\client\eclipse\eclipse.exe).
 - __b. When prompted, select an Eclipse workspace. These instructions will use C:\RTC2002Dev\DevWS.
 - __c. Minimize the **Welcome** via this () button near the top of the window.

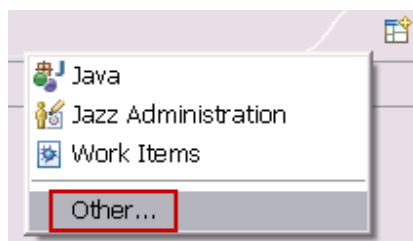
- ___d. From the menu bar, select **Window > Preferences**. Refer to the following screen shot of the **Preferences** dialog for the following steps.



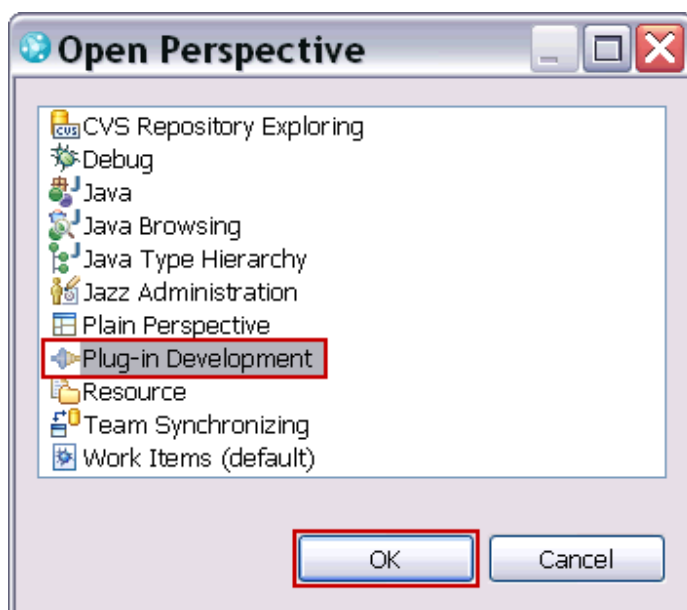
- ___e. In the **Preferences** dialog, select **Plug-in Development > Target Platform**.
- ___f. Next to the **Location** field, click the **Browse...** button.
- ___g. In the **Browse For Folder** dialog, select C:\RTC2002Dev\jazz\client\eclipse\rtc-sdk\target\eclipse and then click **OK**.
- ___h. After the operation completes, click **OK** in the **Preferences** dialog.
- ___2. Open the Plug-in Development perspective.
- ___a. In the toolbar toward the right, click the **Open Perspective** button.



- __b. Then from the menu, select **Other...**

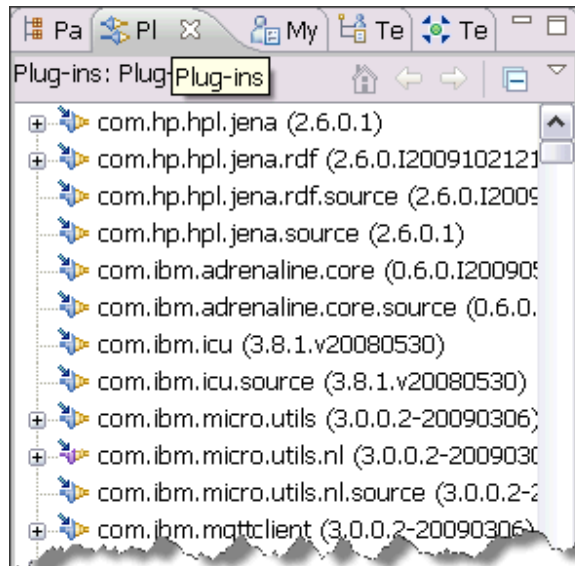


- __c. In the **Open Perspective** dialog, select **Plug-in Development** and then click **OK**.

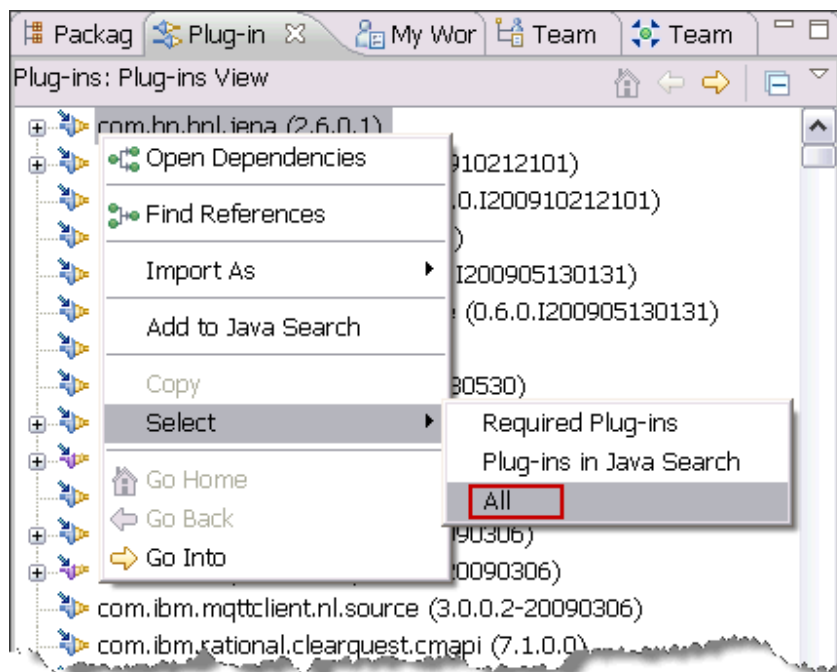


__3. Add RTC source code to Java search.

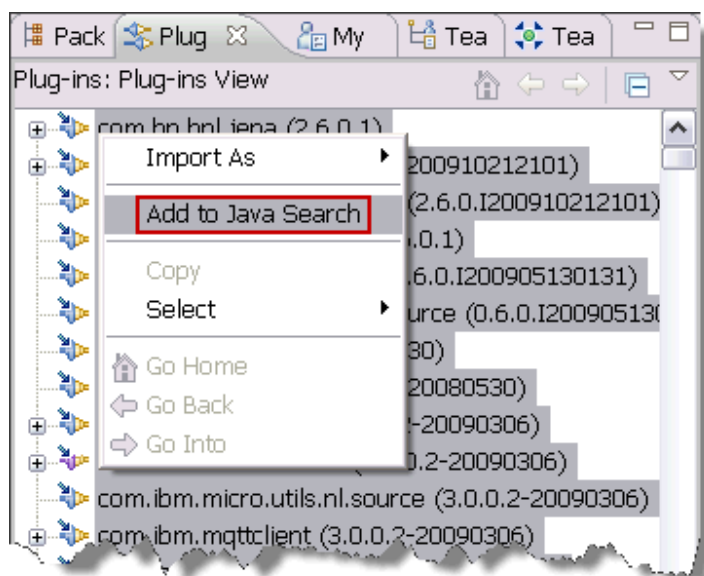
__a. On the left, select the **Plug-ins** view.



__b. From the view's context menu click **Select > All**.



- ___c. From the view's context menu select **Add to Java Search**. There is quite a bit of code. This operation could take a while.



1.3 Setup for Testing With Tomcat



Next, you will enable the Tomcat server for debugging and then you will test the setup by attaching the Eclipse Java debugger to the running server and hitting a breakpoint.

Later, you will setup to launch the server from Eclipse under Jetty. This will use a separate repository database from the Tomcat server. You will also use separate ports. This will give you a development test environment that is separate from your Tomcat test environment.

Finally, you will setup a Jetty launch using the same repository database and ports as the Tomcat server.

Testing with Tomcat has few advantages:



- Mimics a real deployment environment
- Teaches you how to install and configure your extension on the server
- Teaches you how to debug a running live server using Java tools in Eclipse

The primary disadvantage is a longer code, debug and fix cycle.

- ___1. Setup to enable access to the OSGi console when Tomcat is started.
- ___a. First, you need to start the server once to get the Jazz® web application deployed.
- ___b. Open a Windows Explorer and navigate to C:\RTC2002Dev\jazz\server and run the **server.startup.bat** file.

- ___c. After the server has finished starting (the "INFO: Server startup in nnnnn ms" message is displayed in the console), run the **server.shutdown.bat** file.
- ___d. Back in the Windows Explorer, navigate to
C:\RTC2002Dev\jazz\server\tomcat\webapps\jazz\WEB-INF.
- ___e. Edit the web.xml file with Wordpad (Notepad will not interpret the line endings correctly).
- ___f. Find the servlet definition with the id **bridge** (should be near the top).

```
<servlet id="bridge">
    <description>Equinox Bridge Servlet</description>
    <display-name>Equinox Bridge Servlet</display-name>
    <servlet-name>equinoxbridgeservlet</servlet-name>
    <servlet-class>com.ibm.team.repository.server.servletbridge.Jazz...
    <init-param>
        <param-name>enableFrameworkControls</param-name>
        <param-value>>false</param-value>
    </init-param>
    <!--
    <init-param>
        <param-name>commandline</param-name>
        <param-value>-console</param-value>
    </init-param>
    -->
    ...
```

- ___g. Note the commented out **init-param** near the snippet shown above. Uncomment that element so that it now looks like:

```
<servlet id="bridge">
    <description>Equinox Bridge Servlet</description>
    <display-name>Equinox Bridge Servlet</display-name>
    <servlet-name>equinoxbridgeservlet</servlet-name>
    <servlet-class>com.ibm.team.repository.server.servletbridge.Jazz...
    <init-param>
        <param-name>enableFrameworkControls</param-name>
        <param-value>>false</param-value>
    </init-param>
    <init-param>
        <param-name>commandline</param-name>
        <param-value>-console</param-value>
    </init-param>
    ...
```

- ___h. Save the web.xml file and close Wordpad.



After the server starts up when using the console enabling options you just set, the osgi prompt will not appear until after you hit enter in the console window. You can then enter osgi commands.



Also, when you run the shutdown.server.bat file when using the console enabling options you just set, you have to hit enter in the Tomcat console window after the server is stopped in order for the console window to close.

__2. Setup to run the server in debug mode.

- __a. Open Windows Explorer and navigate to C:\RTC2002Dev\jazz\server.
- __b. Open the server.startup.bat file with either Notepad or Wordpad.
- __c. Find the line that starts with the following. It is near the bottom of the file. Note that the line is much longer than this.

```
set JAVA_OPTS=-Djava.awt.headless=true -DSQLSERVER_JDBC="%SQLSERVER_JDBC%"
```

- __d. After the “-Djava.awt.headless=true” option, add the following options:

```
-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=3388
```

- __e. Be sure to keep all the other options and that the command remains one long line. Note that the port to attach the debugger to is 3388. Save the file and close the editor.

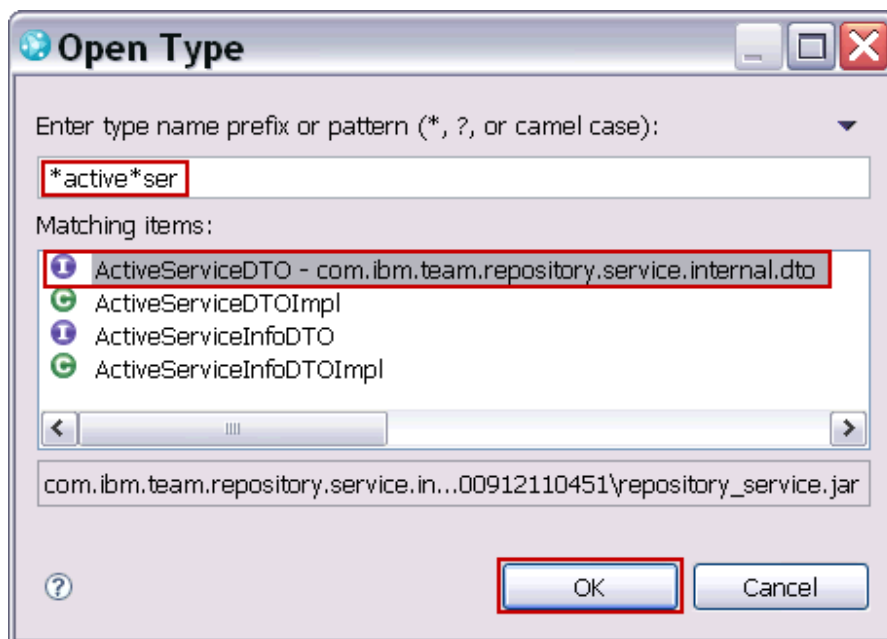


The port, 3388 in this case, could be different. It must be one that is not blocked by your firewall or in use by another application. The one suggested on jazz.net, 1044, was blocked on my laptop as I wrote this. I was able to find an unblocked port by running the server without the “address=<port>” sub-option on the “-Xrunjdwp” option. When I did that, an unblocked port was chosen randomly and was displayed at the top of the Tomcat console window. If 3388 does not work for you (the Tomcat window dies as soon as it opens) you can try the same work around to discover an open port.

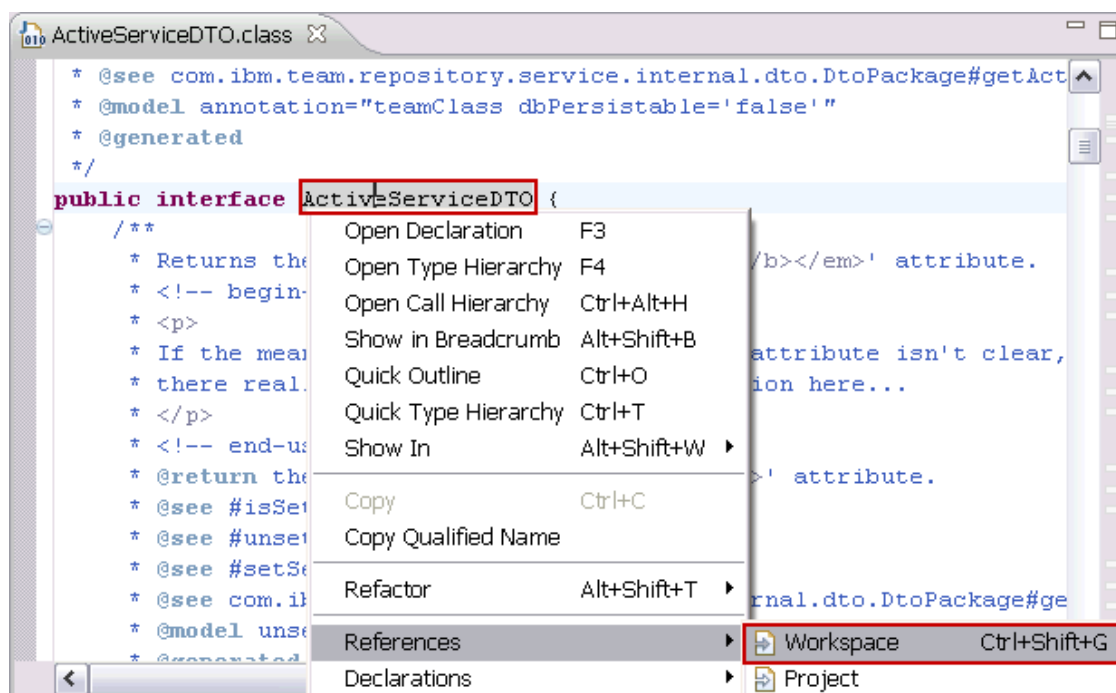
1.4 Test connecting the Eclipse debugger to Tomcat

- __1. Start the RTC server.
 - __a. Open Windows Explorer and navigate to C:\RTC2002Dev\jazz\server.

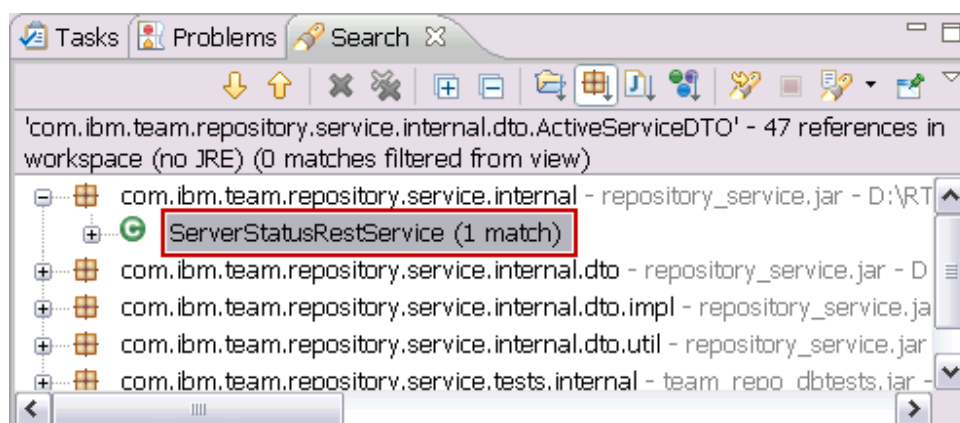
- ___b. Run the server.startup.bat file. Note that for some VMs, the server may pause right after the debug port is displayed. This is as expected. Some VMs, when started in debug mode, execution is suspended until the debugger attaches. You will do that in the following steps.
- ___2. Set a breakpoint to be used to verify the debugging connection.
 - ___a. Return to your RTC Eclipse client (in the **Plug-in Development** perspective you opened earlier) and from the menu bar select **Navigate > Open Type...**
 - ___b. In the Open Type dialog type `*active*ser` in the pattern field. Four types will appear. Select the **ActiveServiceDTO** interface as shown here and then click **OK**.



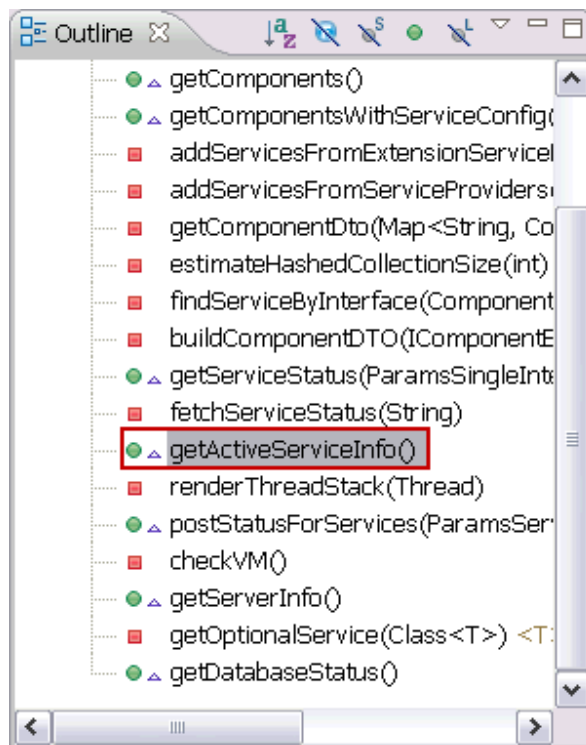
- ___c. When the Java editor opens on the class, the class name will be highlighted. Right click the class name and select **References > Workspace**.



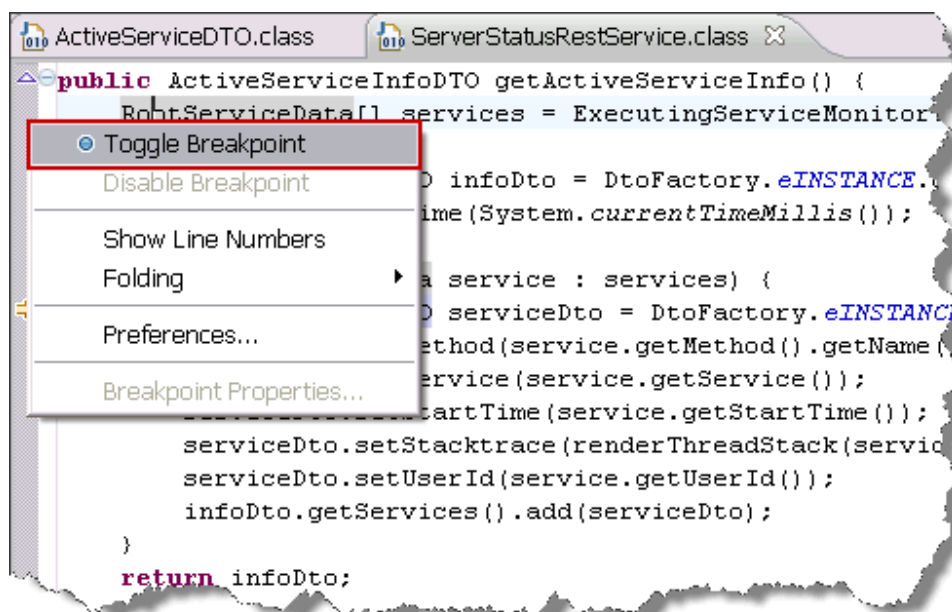
- ___d. The first entry in the **Search** results view is the one you want. Double click the **ServerStatusRestService** class to open an editor on it.



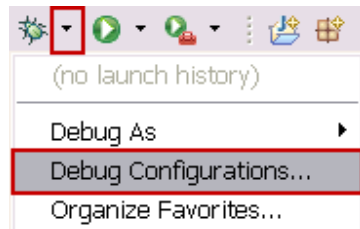
- ___e. The **Outline** view now shows the structure of the `ServerStatusRestService` class. In the Outline view, click the **getActiveServiceInfo()** method.



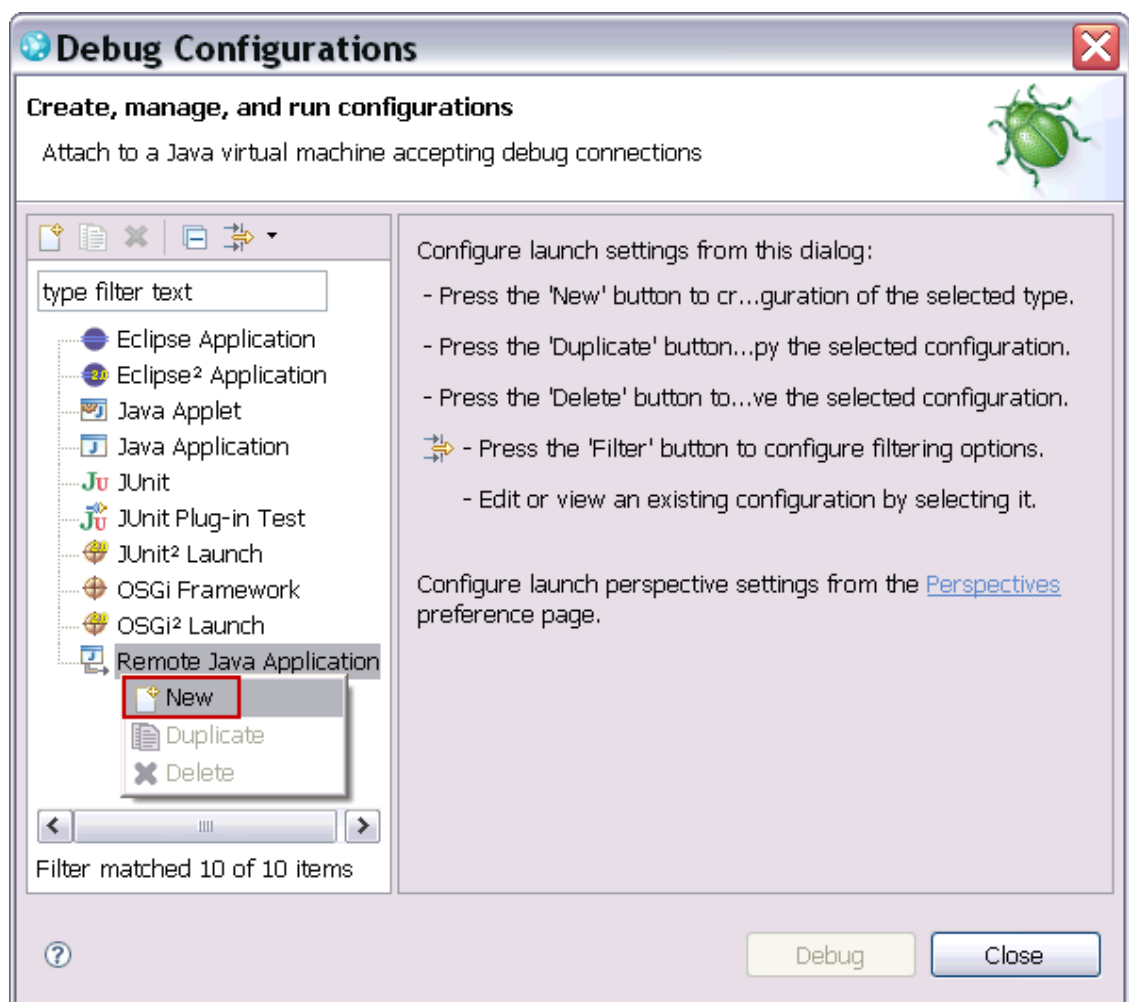
- ___f. The editor is now showing the `getActiveServiceInfo()` method. Set a breakpoint on the first line of the method. Right click in the gray area to the left of the first line to get the menu.



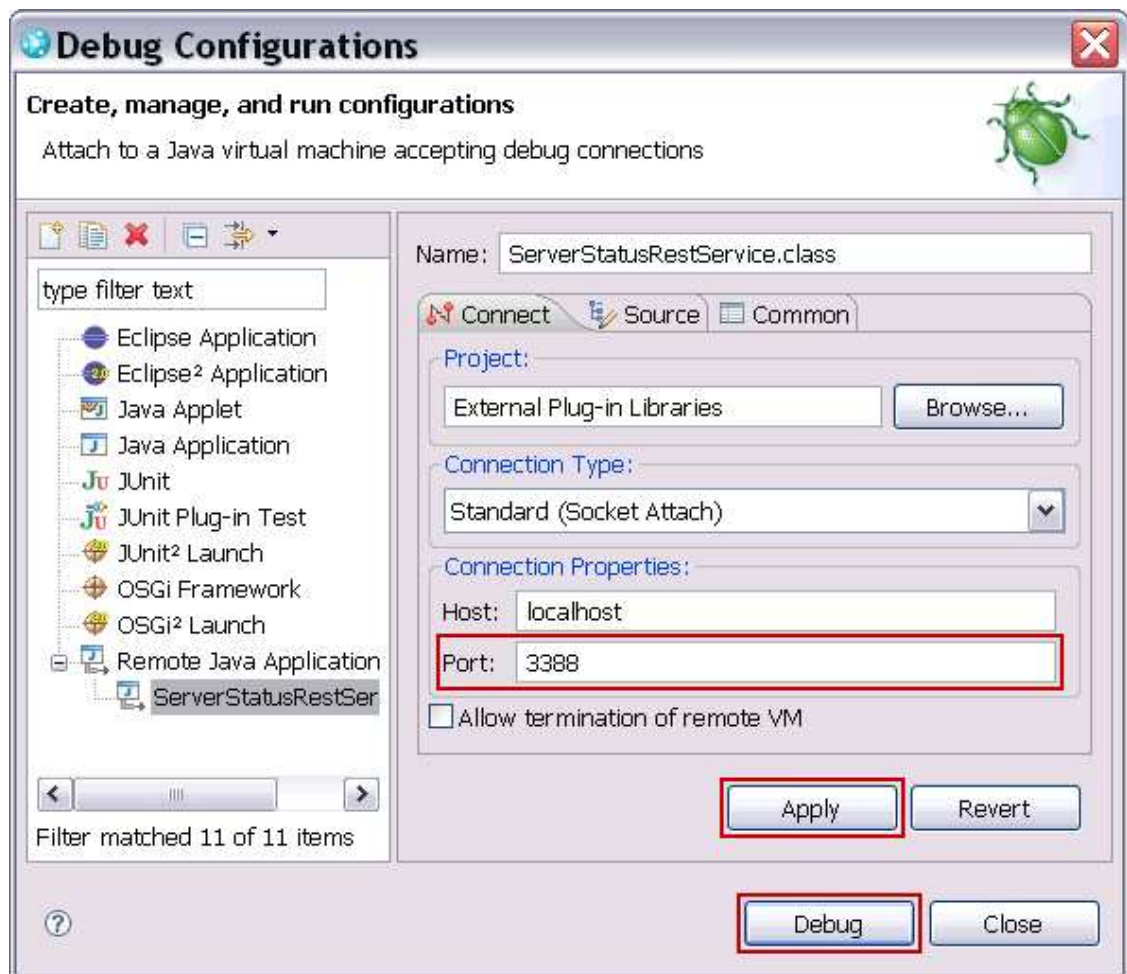
- ___3. Attach the Eclipse debugger to the RTC server.
- ___a. From the Debug toolbar icon dropdown select **Debug Configurations...**



- ___b. In the **Debug Configurations** dialog, right click **Remote Java Application** and then select **New** from the pop-up menu.

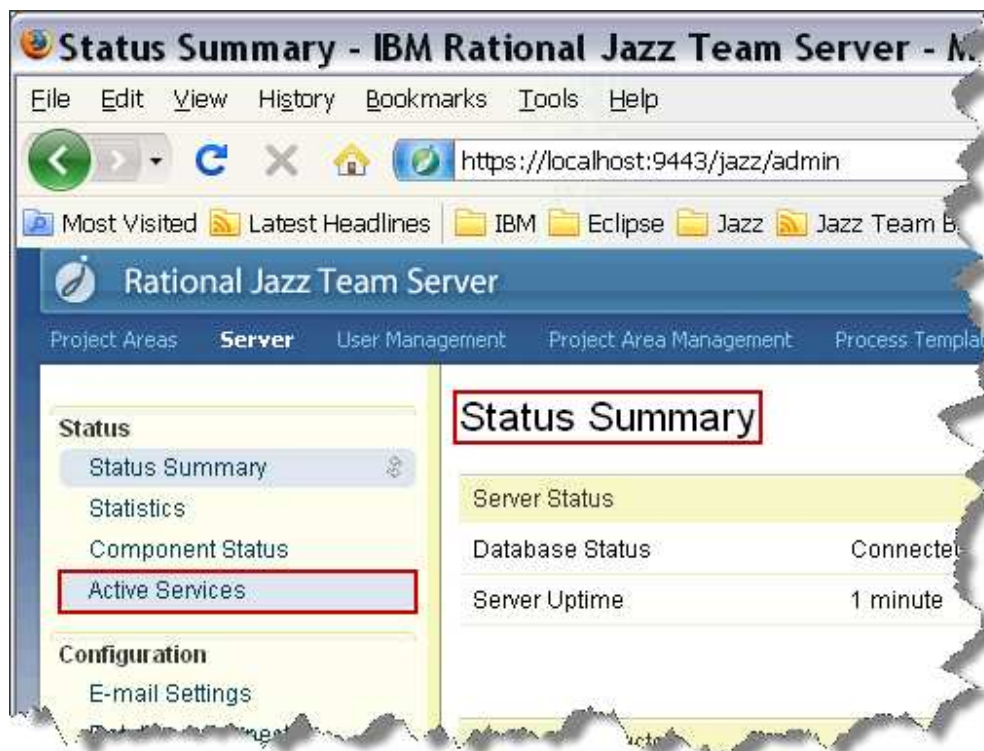


- __c. Set the values for the new debug configuration as shown here. Especially note the **Port** value. Click **Apply** and then click **Debug**.

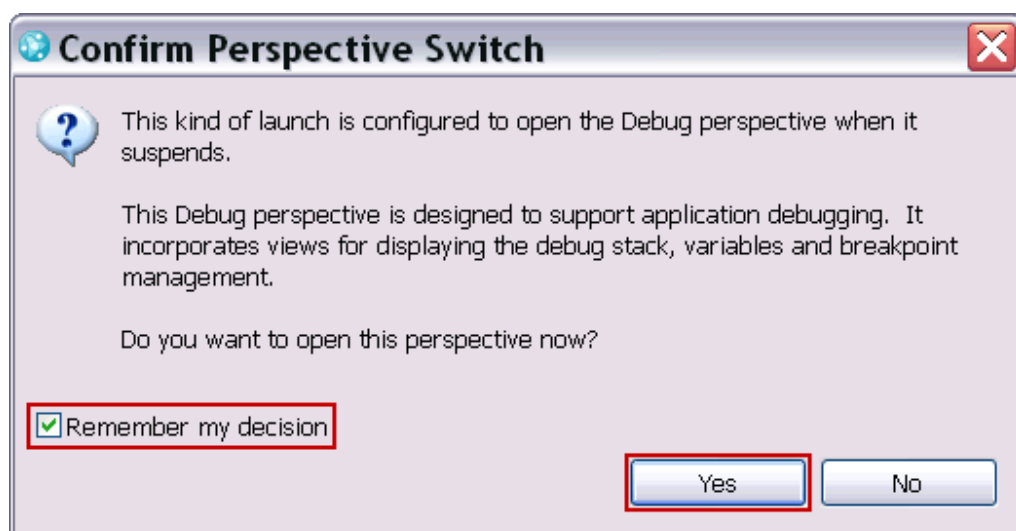


- __d. If your server had paused waiting for the debugger to attach, switch back to the Tomcat console window and you will notice that the server is now running.
- __4. Use the RTC Web UI to trigger the breakpoint.
- __a. Open your browser and enter the URL <https://localhost:9443/jazz/admin>. It may take longer than usual for the login page to appear since you are making the first request (and initial database connection) to a server running under debug. Also, you may need to add a security exception.
 - __b. Login with ADMIN as both **User ID** and **Password**.

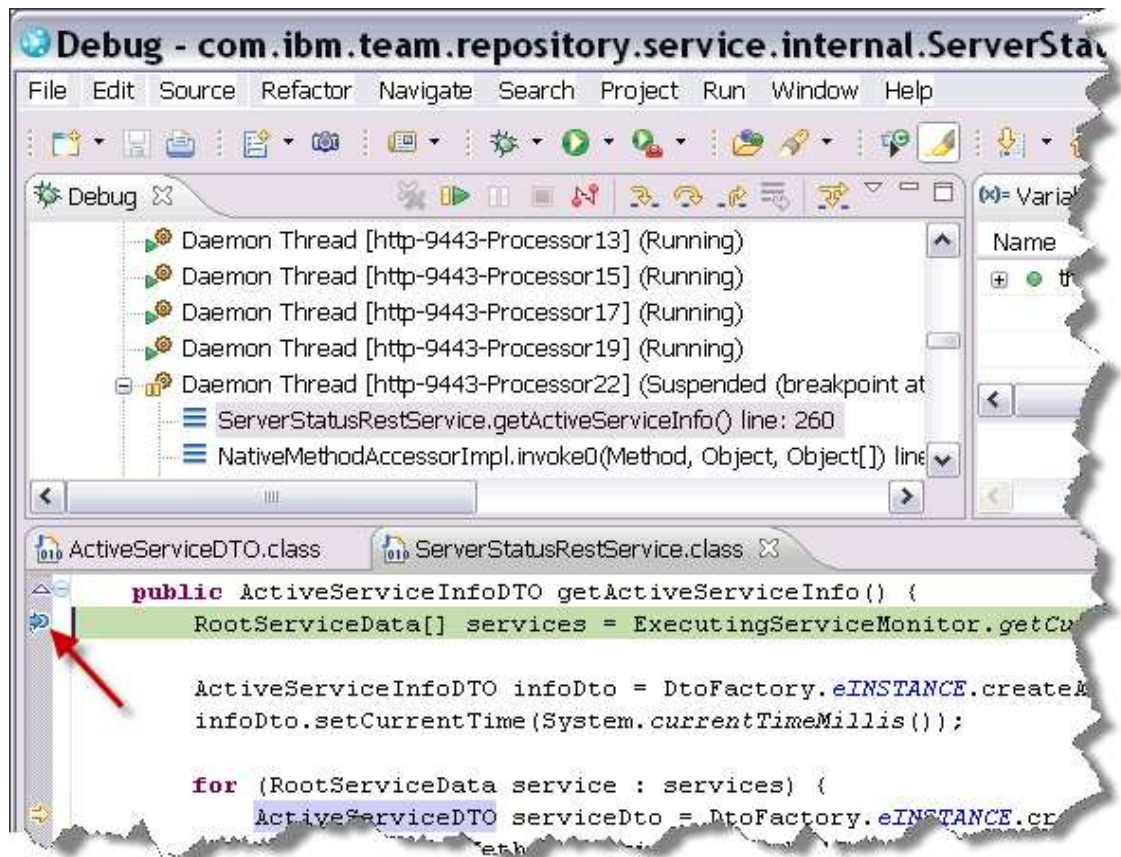
- ___c. When the **Status Summary** page appears, click the **Active Services** link on the left.



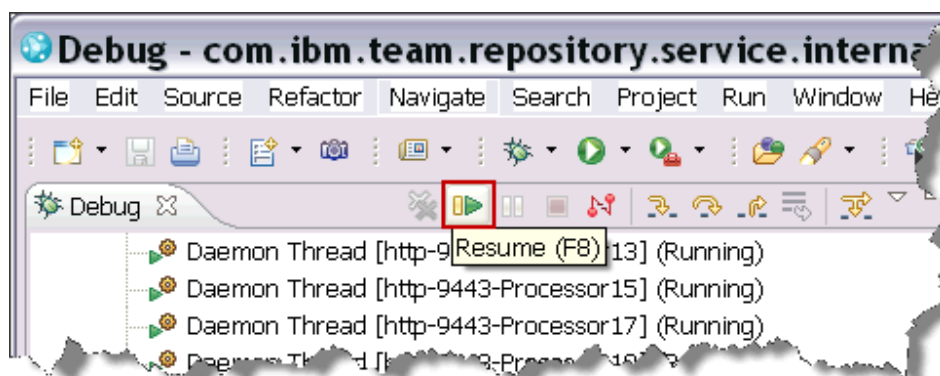
- ___d. The breakpoint will trigger and the RTC Eclipse client should come to the foreground (or flash in the Windows taskbar if minimized). If you are prompted to switch to the Debug perspective, click the **Remember my decision** checkbox if you wish, and then click **Yes**.



- __e. You will now be in the Debug perspective stopped at the breakpoint you set earlier.

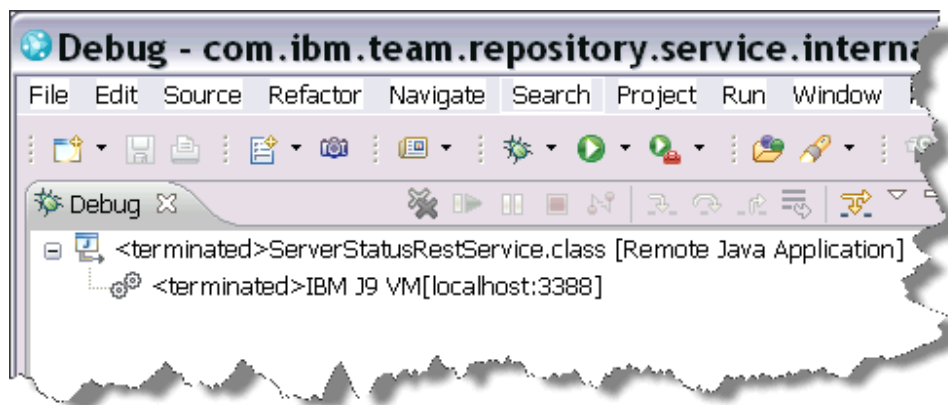


- __f. Click the **Resume** toolbar button to resume execution of the server.



- __g. Return to your browser and note that the **Active Services** page is now showing. Close your browser window.

- ___h. Return to the Windows Explorer and navigate to C:\RTC2002Dev\jazz\server and run the **server.shutdown.bat** file.
- ___i. Switch to the Tomcat window where the server is running. You may need to hit Enter after the message "INFO: Failed shutdown of Apache Portable Runtime" appears to completely shut the server down.
- ___j. Return to your RTC Eclipse client and note that the server process has terminated.



1.5 Setup for Jetty Based Server Launch



As mentioned earlier, you will now setup to launch the server from Eclipse under Jetty. This will use a separate repository database than the Tomcat server. You will also use separate ports. This will give you a development test environment that is separate from your Tomcat test environment.

Testing with Jetty has a couple advantages:



- Faster server startup to debug
- Faster code, debug and fix cycle including hot code replace

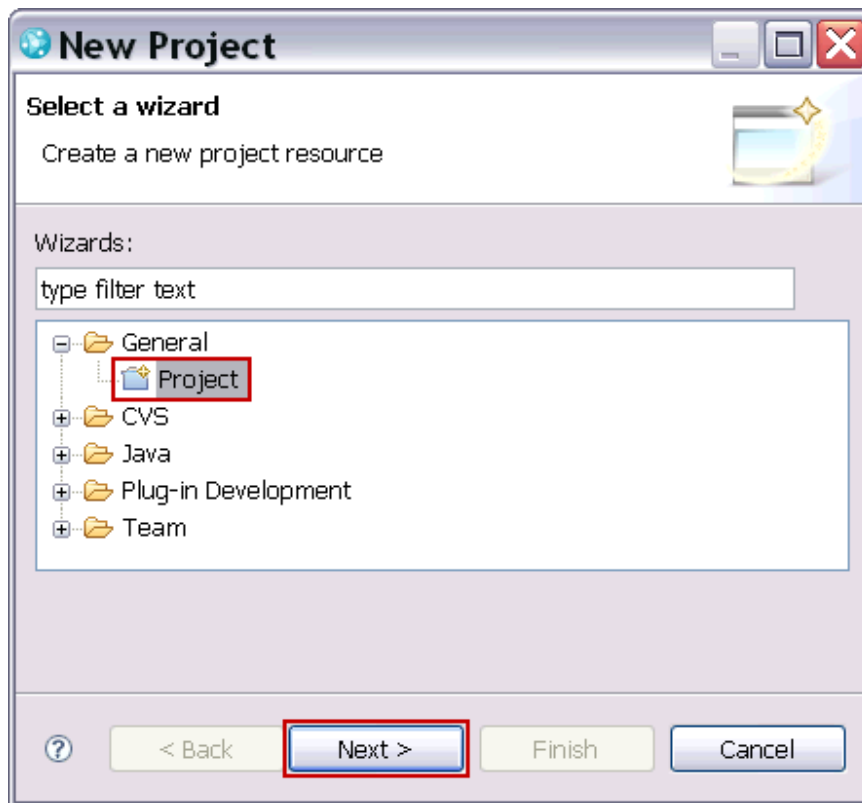
The primary disadvantage is the extra setup you will experience now.

- ___1. Create an Eclipse project to store launches and associated files.
 - ___a. Return to the **Plug-in Development** perspective. You may close any editors that are left open.

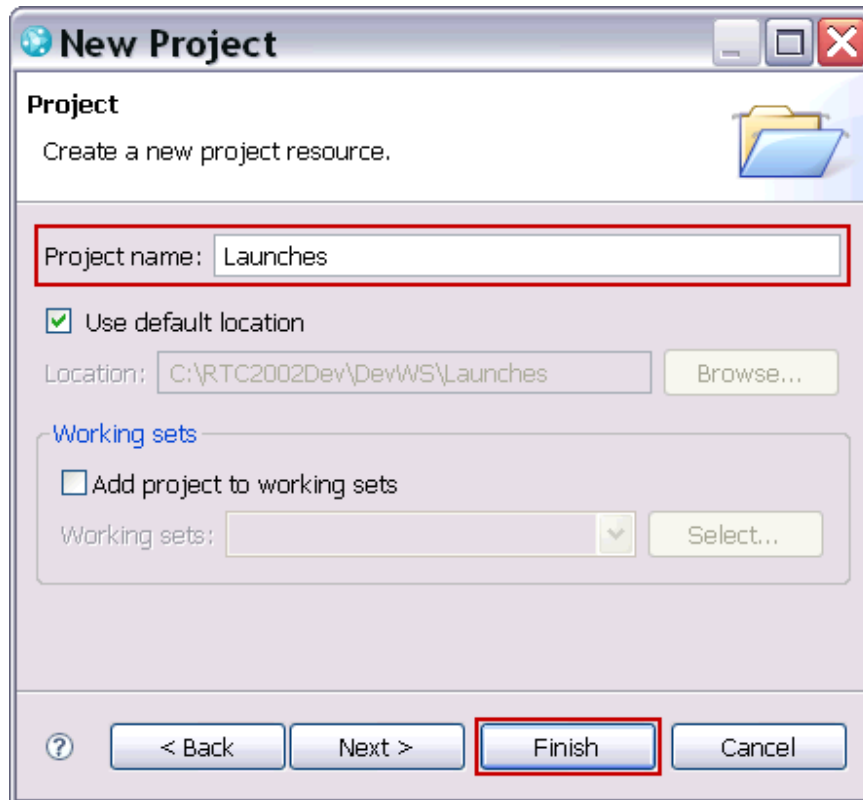


- ___b. From the menu bar, select **File > New > Project...**

- ___c. On the **New Project** wizard, select **General > Project** and click **Next**.

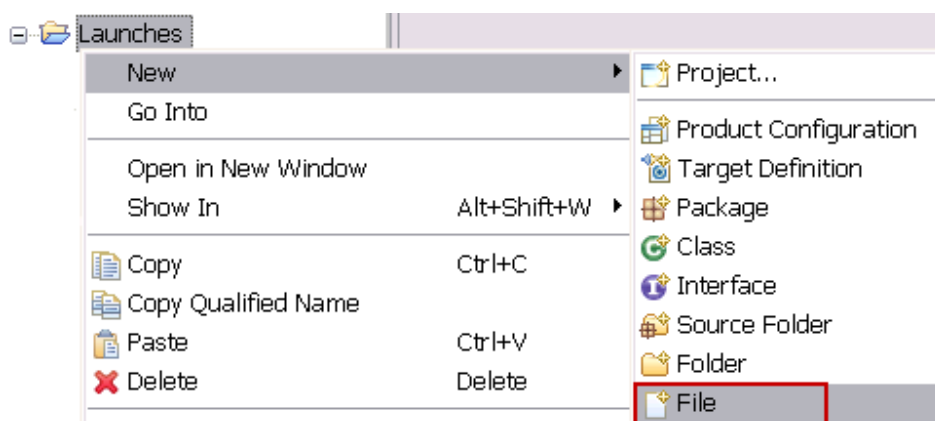


- ___d. On the second page, type `Launches` into the **Project name** field and click **Finish**. The project will now show up in your Package Explorer.

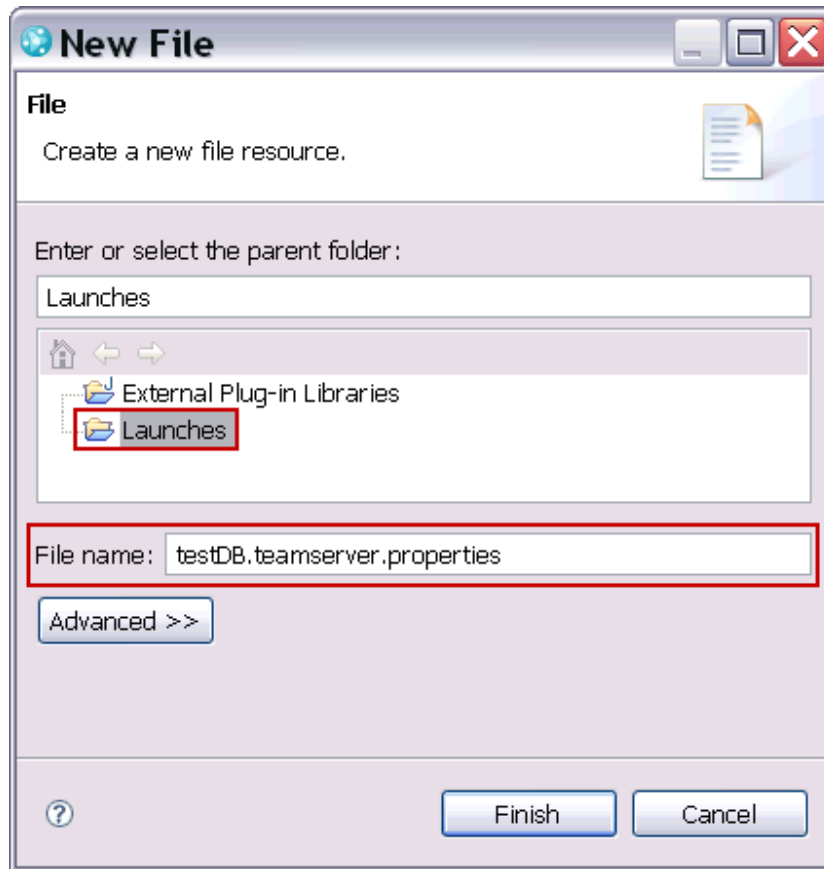


- ___2. Create a `teamserver.properties` file for your test repository Jetty launch.

- ___a. In the **Package Explorer** view on the left, right click the **Launches** project and from the menu select **New > File**.



- ___b. In the **New File** wizard, make sure the **Launches** project is selected as the parent folder, enter `testDB.teamserver.properties` in the **File name** field and then click **Finish**.



- ___c. In the editor that opens, enter these two lines. The first line defines where the repository database will be created. The second defines where the work item index should be built.

```
com.ibm.team.repository.db.jdbc.location=C:/RTC2002Dev/jettywork/testDB
com.ibm.team.fulltext.indexLocation=C:/RTC2002Dev/jettywork/workitemindex
```

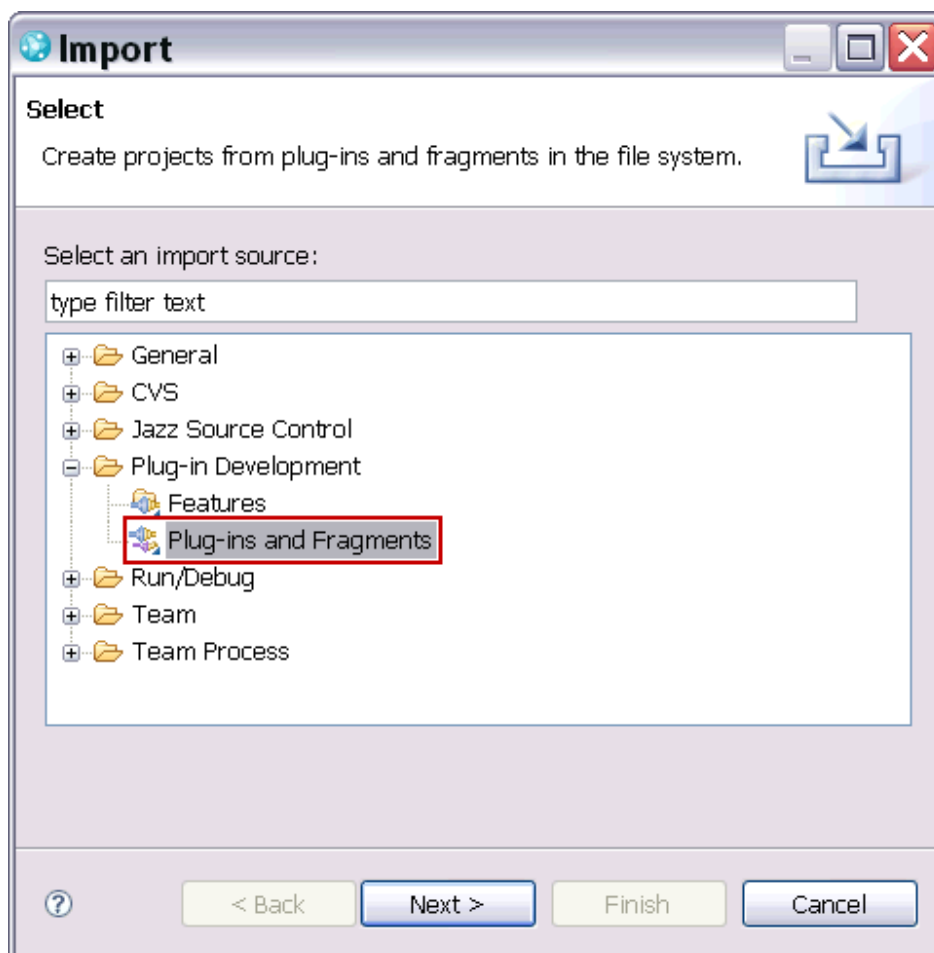
- ___d. Press **Ctrl + S** to save the file and then close the editor.

- ___3. Create a `log4j.properties` file for use with the server launched under Jetty.

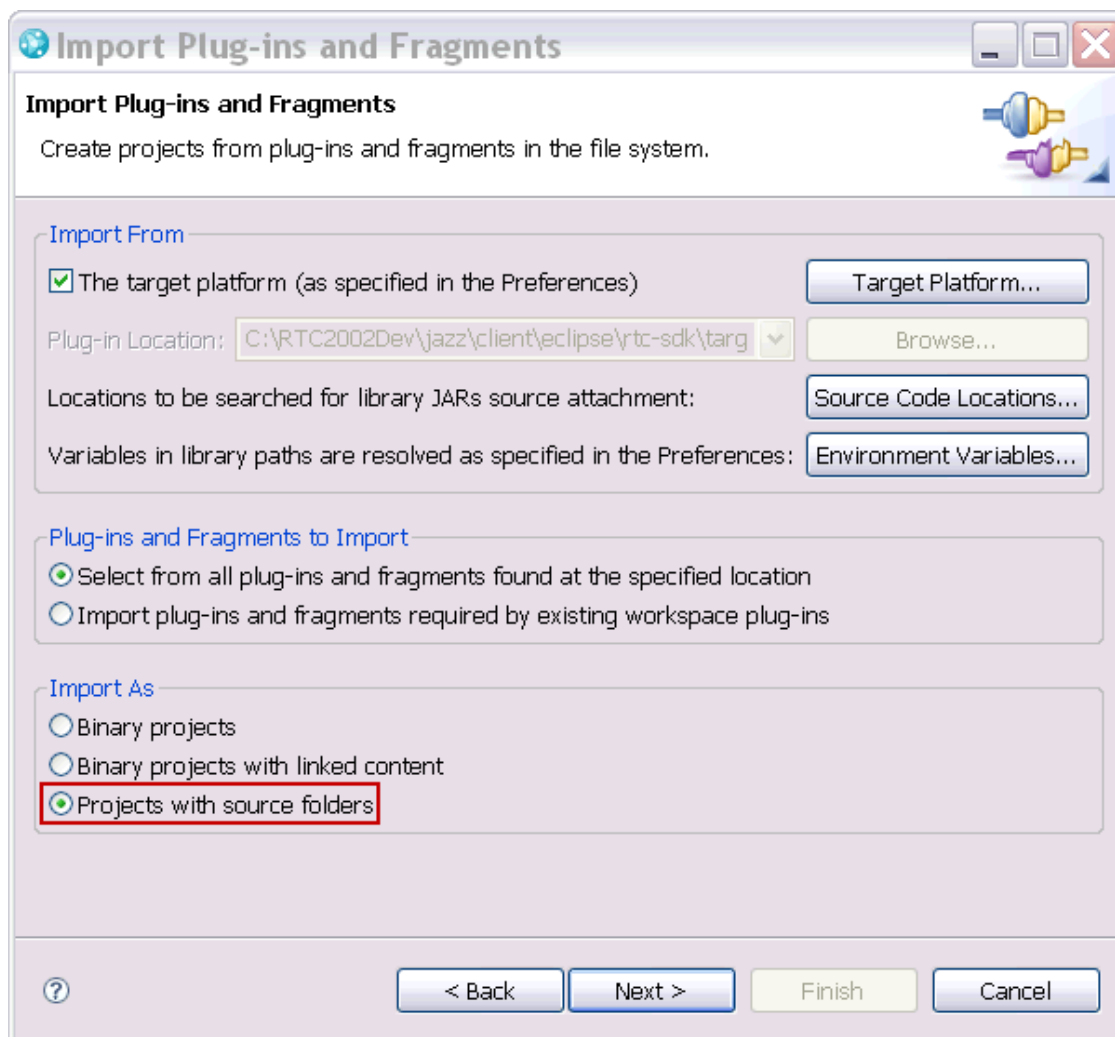
- ___a. Repeat step 2. Except this time, name the file `log4j.properties` and enter the following lines into the file.

```
# Default logging is for WARN and higher
log4j.rootLogger=WARN, stdout
# Log to stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} [%t] %5p %-50.50c - %m%n
log4j.logger.com.ibm.team.server.embedded=INFO
```

- ___4. Import the projects required to create the repository database.
- ___a. First, import the JUnit test plug-in that contains the creation code. From the menu bar, select **File > Import...** and then in the Import wizard, select **Plug-in Development > Plug-ins and Fragments** as shown here and then click **Next**.



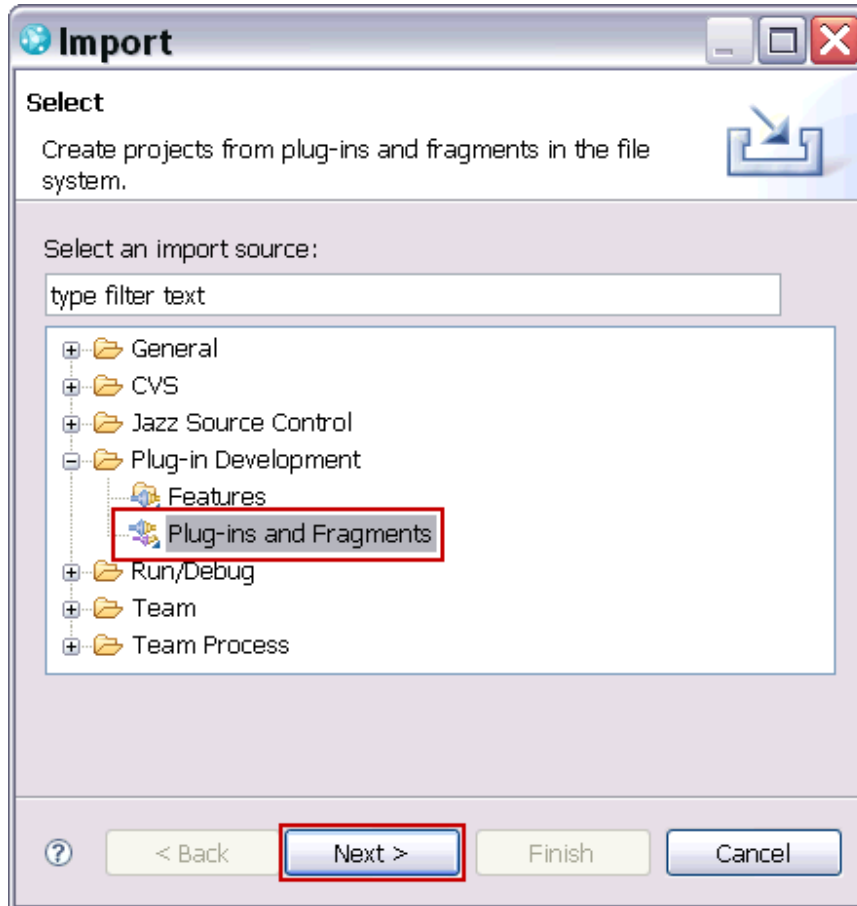
- ___b. On the second page of the wizard, make sure your selections match those shown here. The only one you should have to change is highlighted. Then, click **Next**.



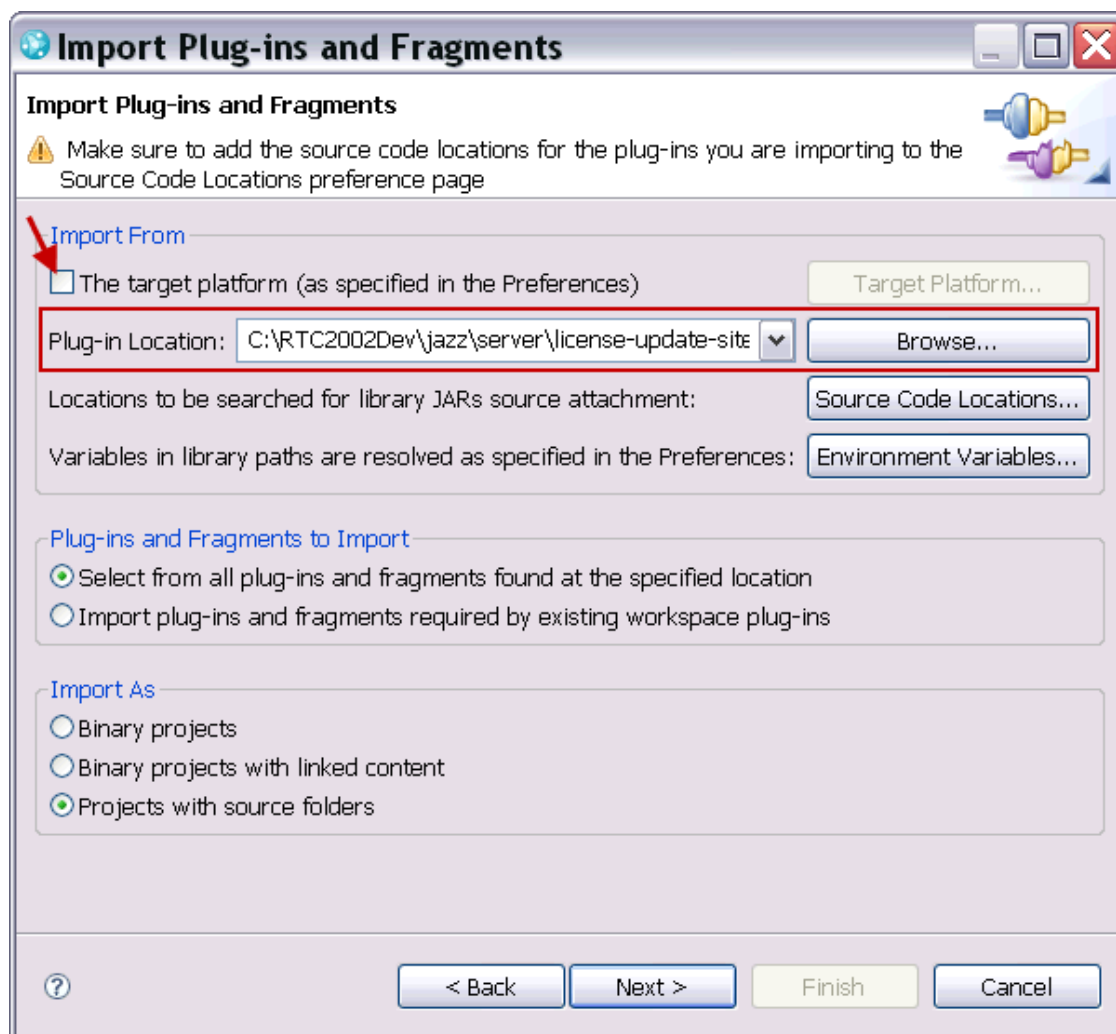
- ___c. On the third page of the wizard, enter `repository.service.tests` into the ID field. This will filter the plug-ins list. Select the **com.ibm.team.repository.service.tests** plug-in, click **Add -->** and then click **Finish**.



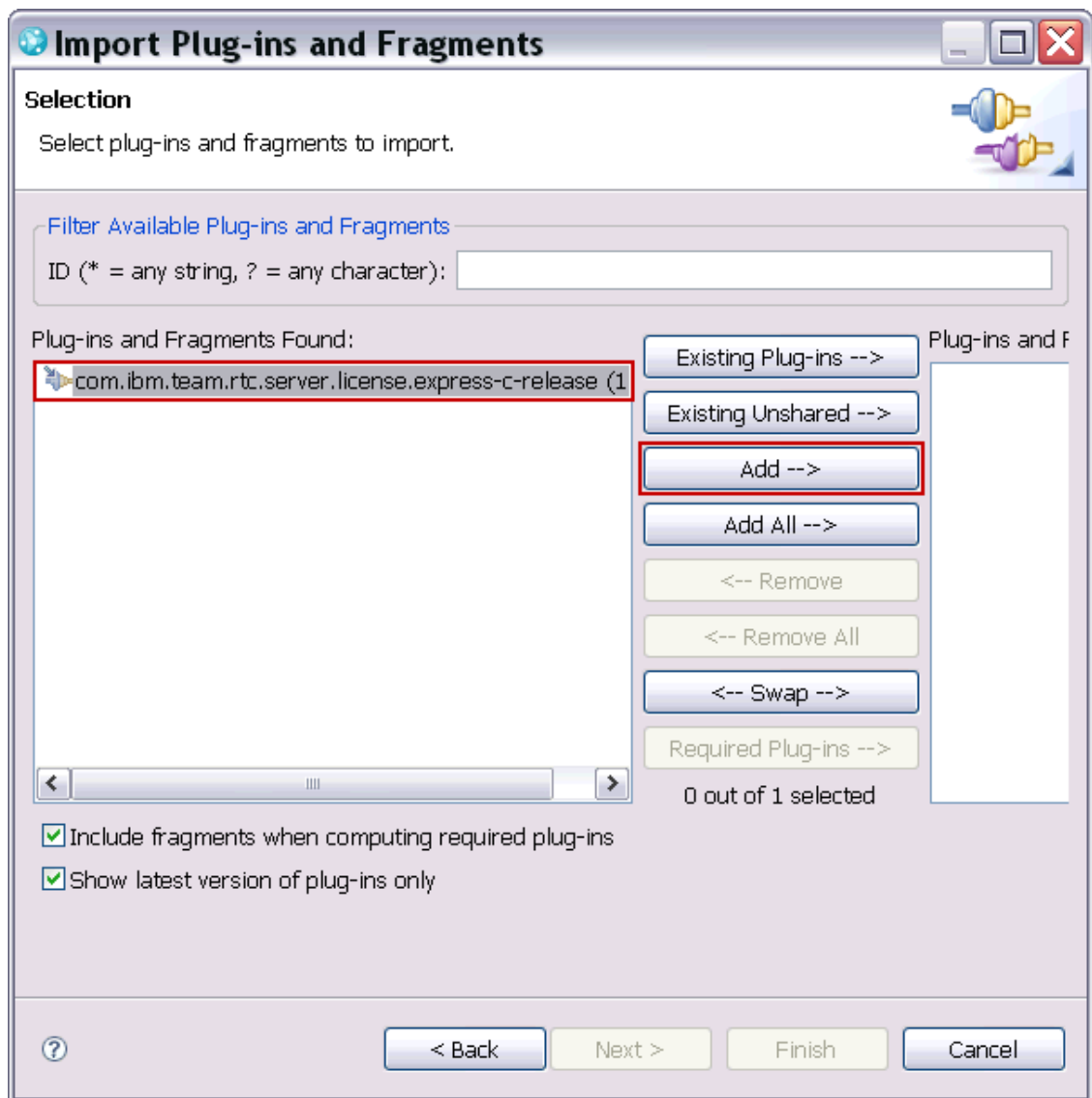
- ___d. Next, import the server license from the unzipped server installation. This will override the development time license you would otherwise be using in a Jetty launch with the permanent Express-C license. It is likely that the development license has expired. As before, from the menu bar, select **File > Import...** and then in the **Import** wizard, select **Plug-in Development > Plug-ins and Fragments** as shown here and then click **Next**.



- ___e. This time on the second page of the wizard, make sure your selections match those shown here and then click **Next**. The major difference from last time is the selection of a different place to import from. The **Plug-in Location** field should be set to (use the **Browse...** button to find it): C:\RTC2002Dev\jazz\server\license-update-site\plugins.

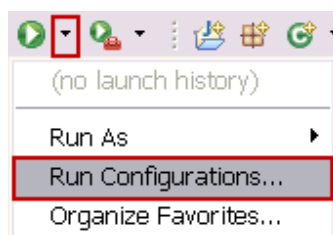


- ___f. On the third page of the wizard, select the **com.ibm.team.rtc.server.license.express-c-release** plug-in. Then click **Add -->** and finally click **Finish**.



- ___5. Create the development time repository database.

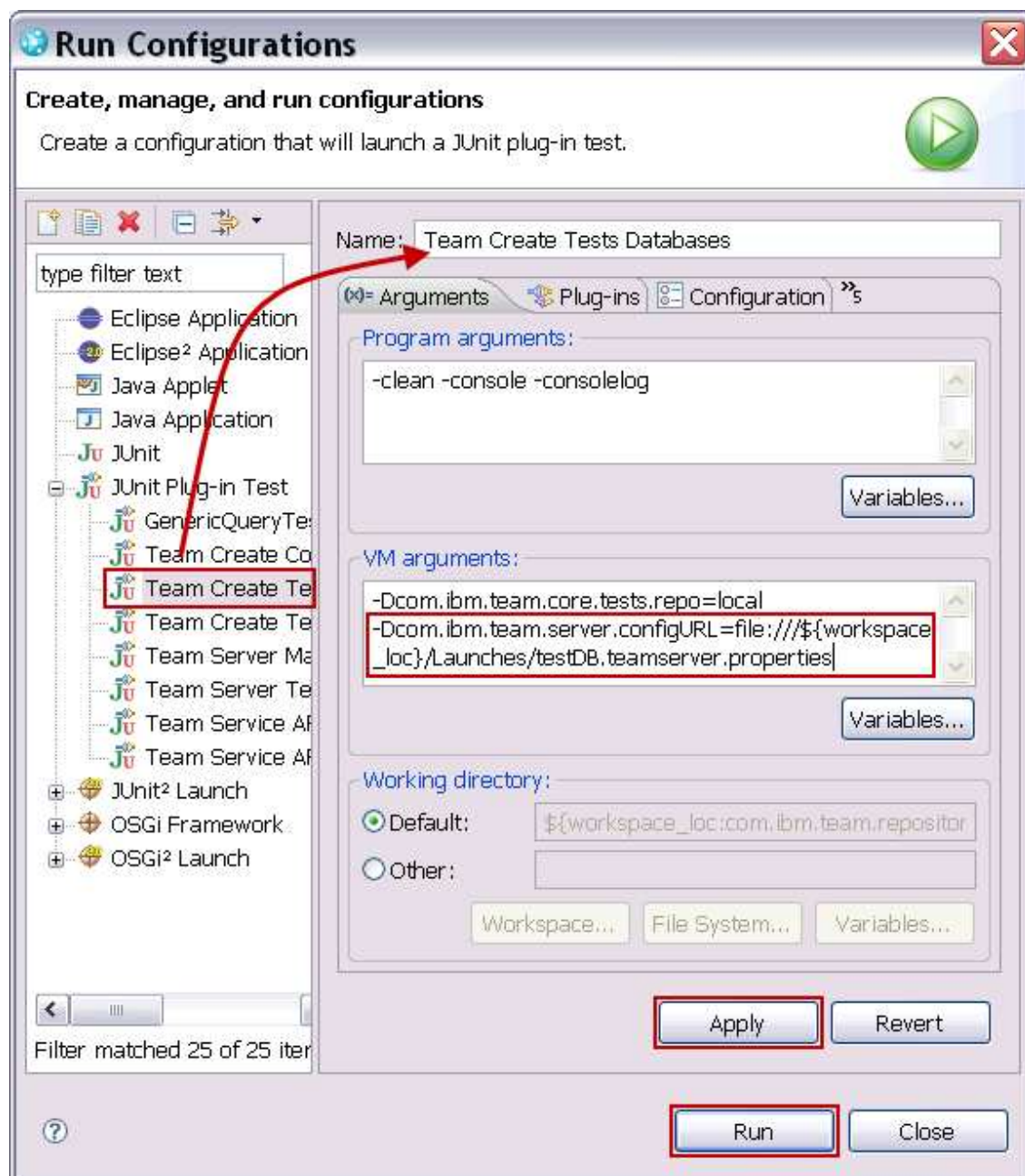
- ___a. Select **Run Configurations...** from the dropdown menu off the **Run** toolbar icon.



- __b. On the **Run Configurations** dialog, select **JUnit Plug-in Test > Team Create Tests Databases**, select the **Arguments** tab and enter the following as a second entry in the **VM Arguments** field (in the screen shot it appears to be more than one line but that is due to wrapping):

```
-Dcom.ibm.team.server.configURL=file:///${workspace_loc}/Launches/testDB.teamserver.properties
```

Then, click **Apply** and then click **Run**.

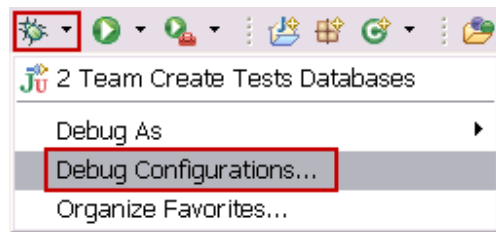


- __c. This may take a while to run. The **Console** view will appear and show quite a bit of output. The **JUnit** view will also be active. When the database creation is complete, the **JUnit** view will show success. Note that the Console view will show some exceptions. The important thing is that the JUnit view shows success.

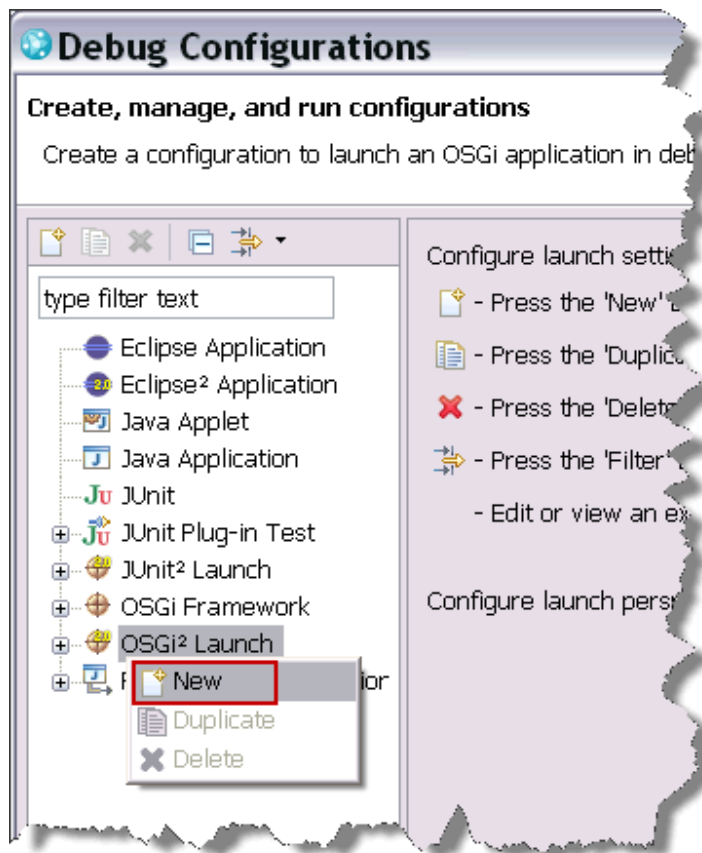


- __6. Create a launch configuration to launch the RTC server under Jetty.

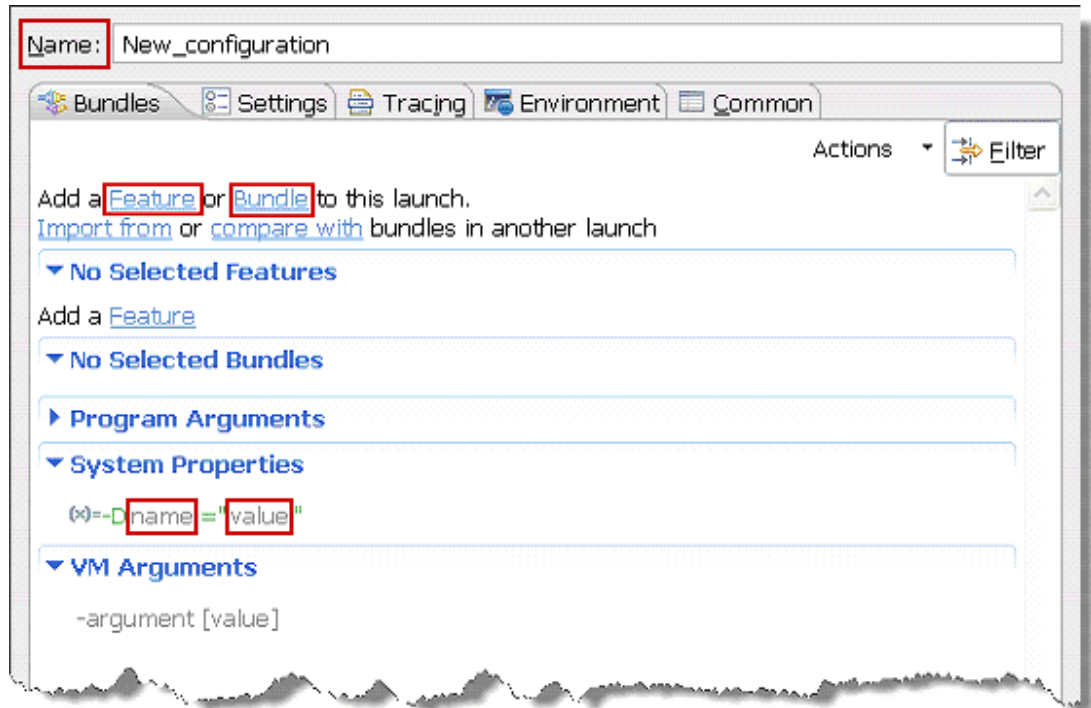
- __a. Select **Debug Configurations...** from the dropdown menu off the **Debug** toolbar icon.



- ___b. In the **Debug Configurations** dialog, right click **OSGi² Launch** and then select **New** from the menu.

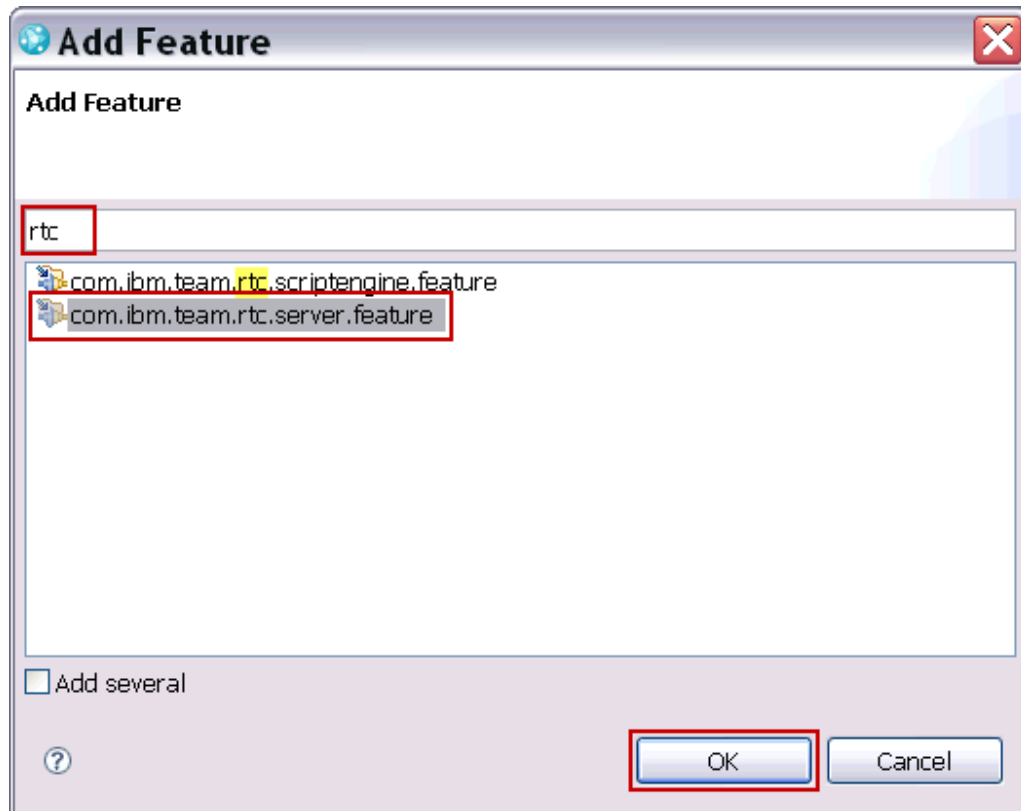


- __c. A new launch definition appears in the right hand pane in the **Debug Configurations** dialog. You will use the highlighted links and fields to add three features, one bundle and 6 system properties to the launch.



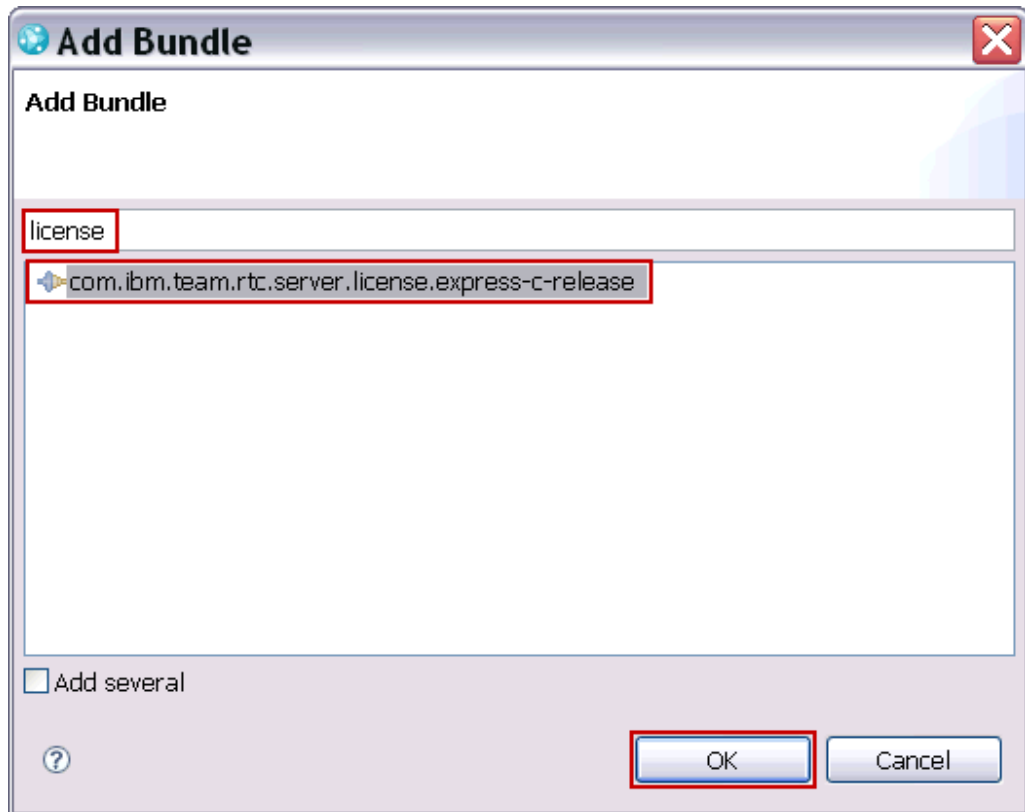
- __d. Enter Jetty Jazz Server into the **Name** field (at the top of the form).

- ___e. Click **Feature** as highlighted above. In the **Add Feature** dialog, type `rtc` in the filter entry field, select **`com.ibm.team.rtc.server.feature`** from the list and then click **OK**.



- ___f. Repeat step e but this time type `jetty` in the filter field and select **`com.ibm.team.server.embedded.jetty.jazz.feature`** from the list.
- ___g. Repeat step e once again but this time type `jetty` in the filter field and select **`com.ibm.team.server.embedded.jetty.base.feature`** from the list.

- __h. Click **Bundle** as highlighted above. In the **Add Bundle** dialog, type `license` in the filter entry field, select **`com.ibm.team.rtc.server.license.express-c-release`** from the list and then click **OK**.



- ___i. Here are the **System Properties** you will be adding. Also, note that not only are you specifying your teamserver.properties file here (the last property in the table), but additional properties could be added here that would override settings in that file.

Table 1: Jetty Server Launch System Properties

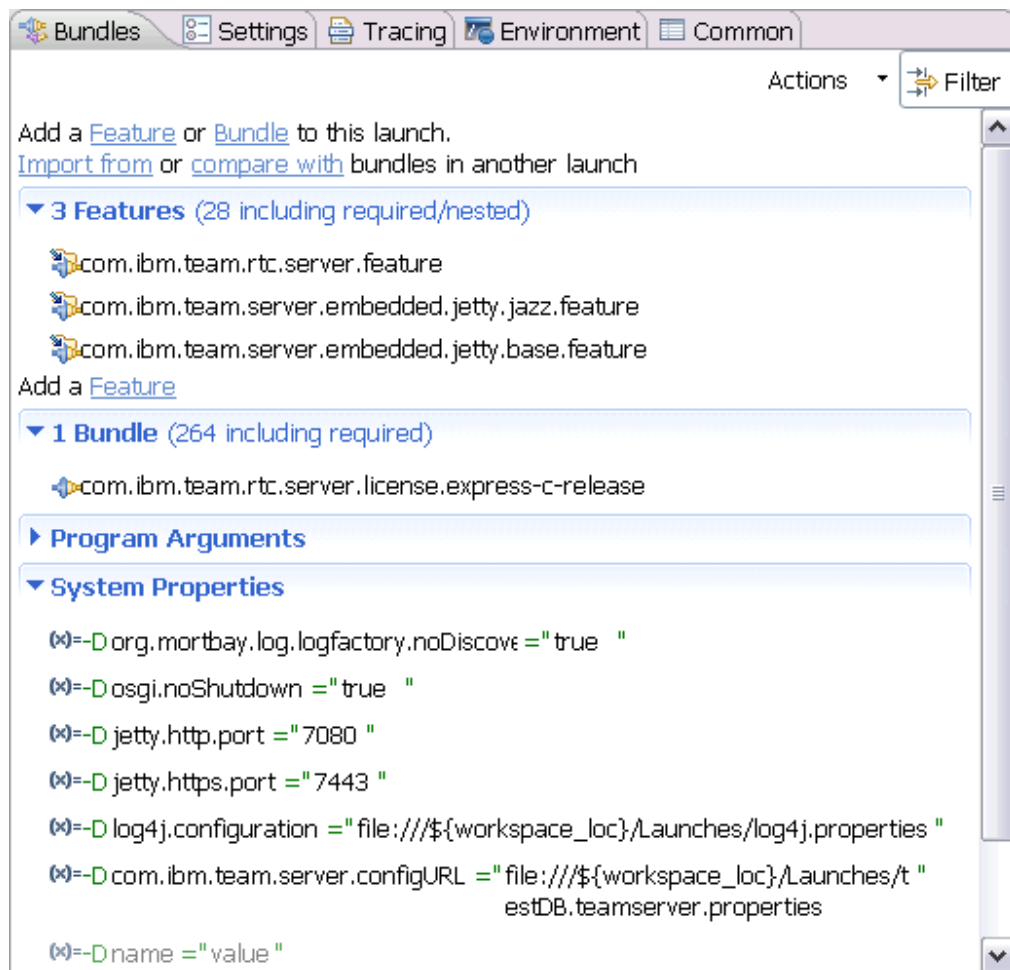
Name	Value
org.mortbay.log.logfactory.noDiscovery	true
osgi.noShutdown	true
jetty.http.port	7080
jetty.https.port	7443
log4j.configuration	file:///\${workspace_loc}/Launches/log4j.properties
com.ibm.team.server.configURL	file:///\${workspace_loc}/Launches/testDB.teamserver.properties

- ___j. For each entry click first on the **name** field and then the **value** field to enter the properties. Notice how a new empty entry appears as you add each one.

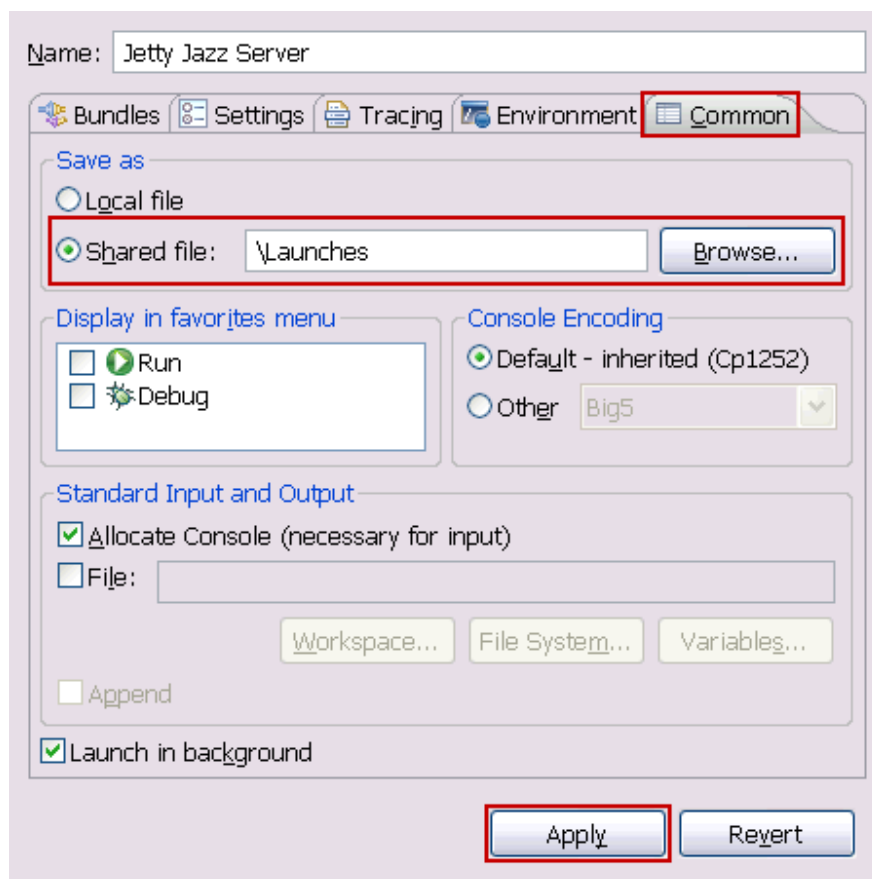
```
(x)=-Dorg.mortbay.log.logfactory.noDiscover ="true| "
```

```
(x)=-Dname ="value "
```

- ___k. After you are complete, the launch configuration's **Bundles** tab should look like this. You are almost, but not quite done.



- ___l. Click on the **Common** tab. Click the **Shared file** radio button and enter `\Launches` (the project you created earlier) into the field. You can use the **Browse...** button to find it. Then, click **Apply**. Now you have a configured launch and it is stored in your Launches project in case you want to be able to easily share it with someone else. Leave the **Debug Configurations** dialog open for the next section.

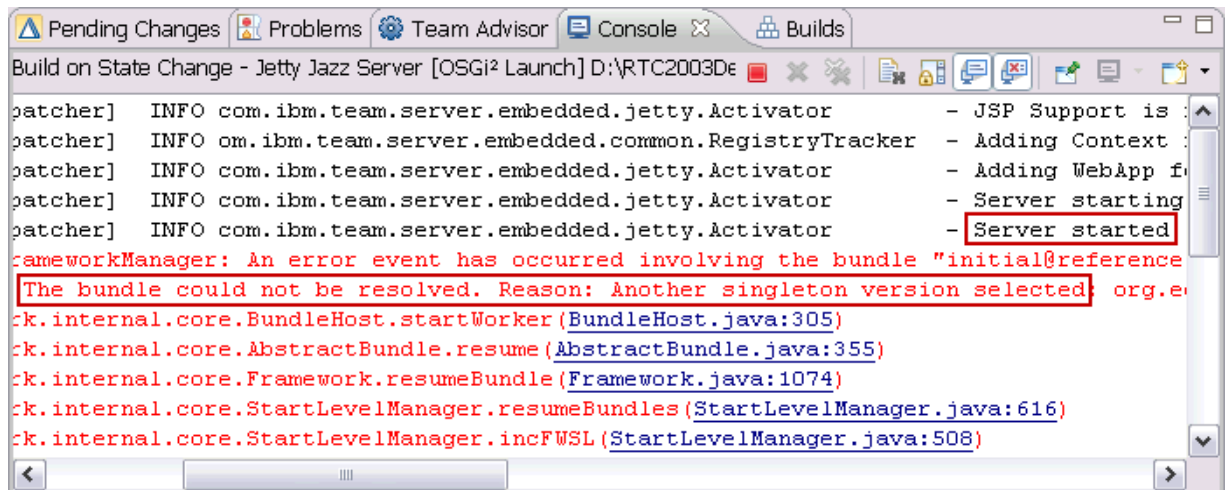


1.6 Test the Jetty Jazz Server Launch

- ___1. Launch the server.
- ___a. In the **Debug Configurations** dialog with the **Jetty Jazz Server** launch still selected on the left, click **Debug**.

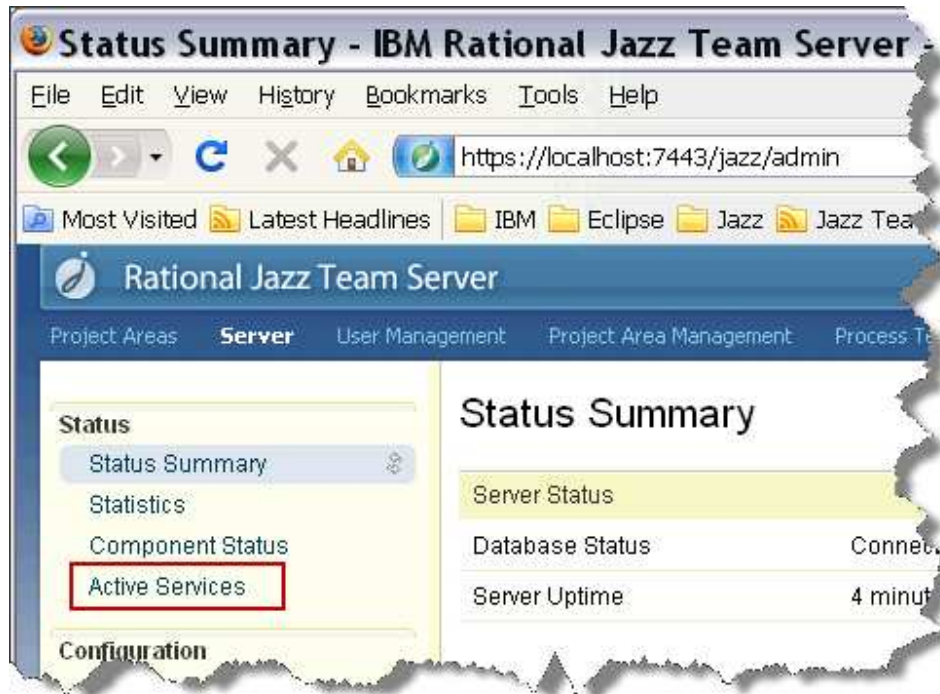
- __b. The **Console** view will take focus and log messages will start appearing there. Once you see a message that ends in **Server started**, the server is up and ready to be used.

Note: You may see an exception or two concerning the presence of two versions of an Eclipse plug-in being present. The messages contain the text “**The bundle could not be resolved. Reason: Another singleton version selected**” just before the exception’s stack trace. You can ignore these exceptions.

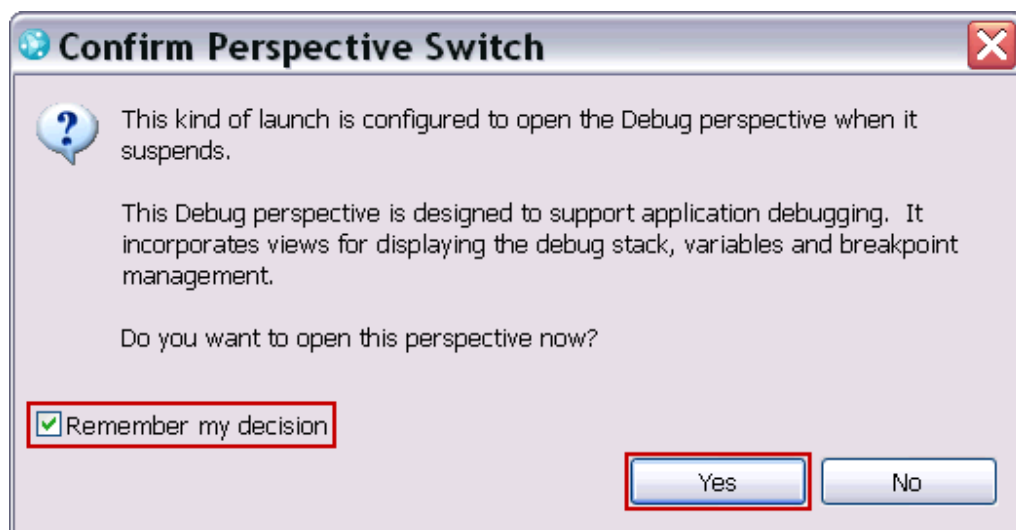


- __2. Connect with your browser.
- __a. Start your browser and navigate to this URL: <https://localhost:7443/jazz/admin>. You may need to add another security exception (note that the port is different).
- __b. Log in with ADMIN as both the **User ID** and **Password**.
- __c. If you switch back to your RTC Eclipse client, you will now notice many more log messages in the **Console** view. These will include entries about a successful connection to the repository database you created earlier.

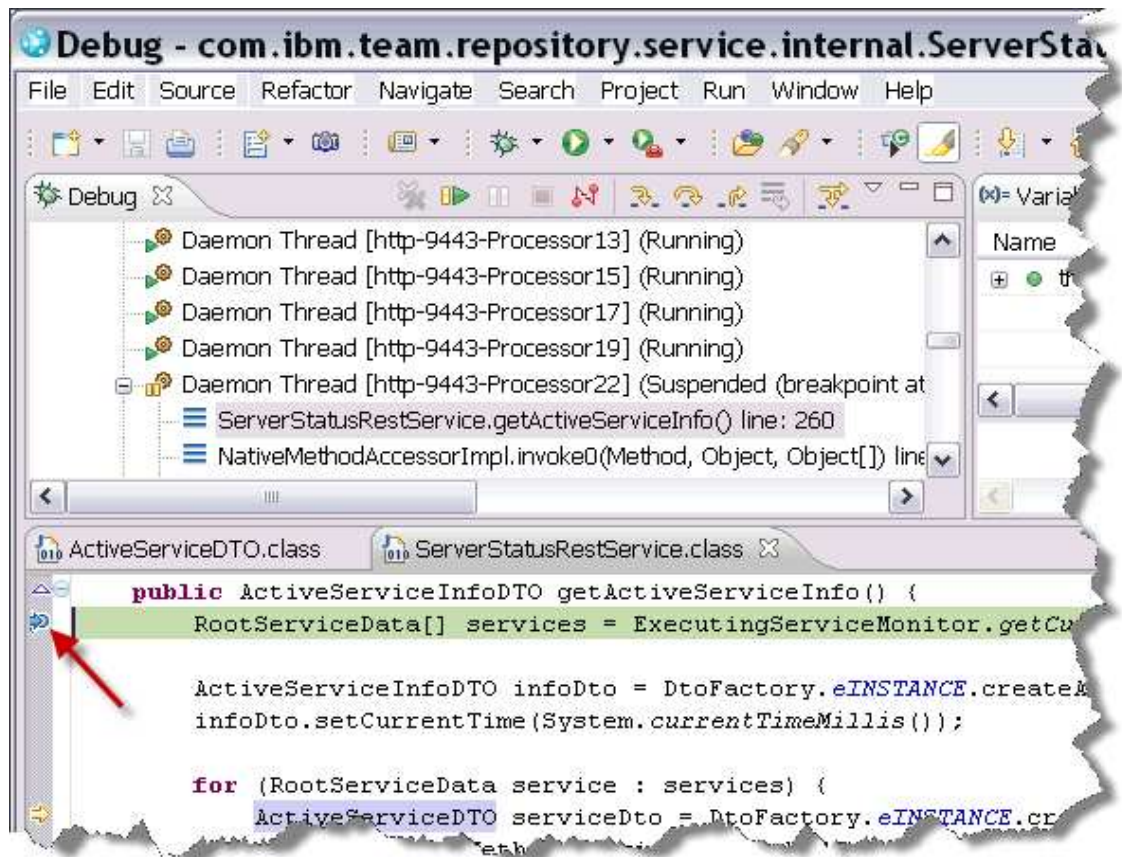
- __3. Trigger the breakpoint set earlier.
- __a. As before, click the **Active Services** link on the left.



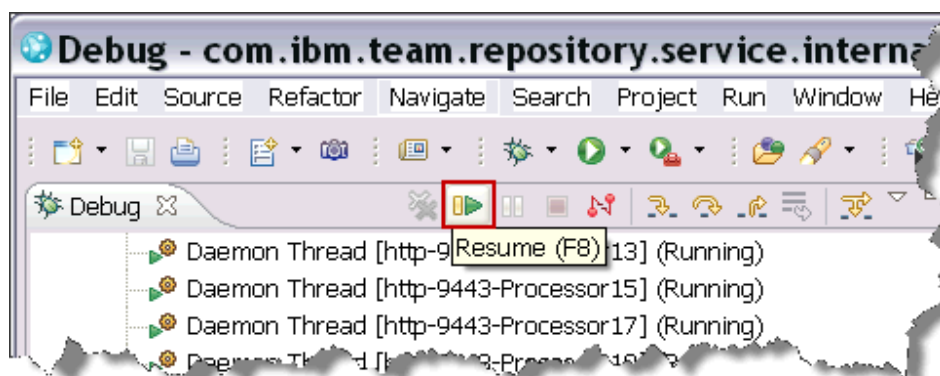
- __b. The breakpoint will trigger and the RTC Eclipse client should come to the foreground or flash in the Windows taskbar. If you are prompted to switch to the Debug perspective, click the **Remember my decision** checkbox if you wish, and then click **Yes**.



- __c. You will now be in the Debug perspective stopped at the breakpoint you set earlier.



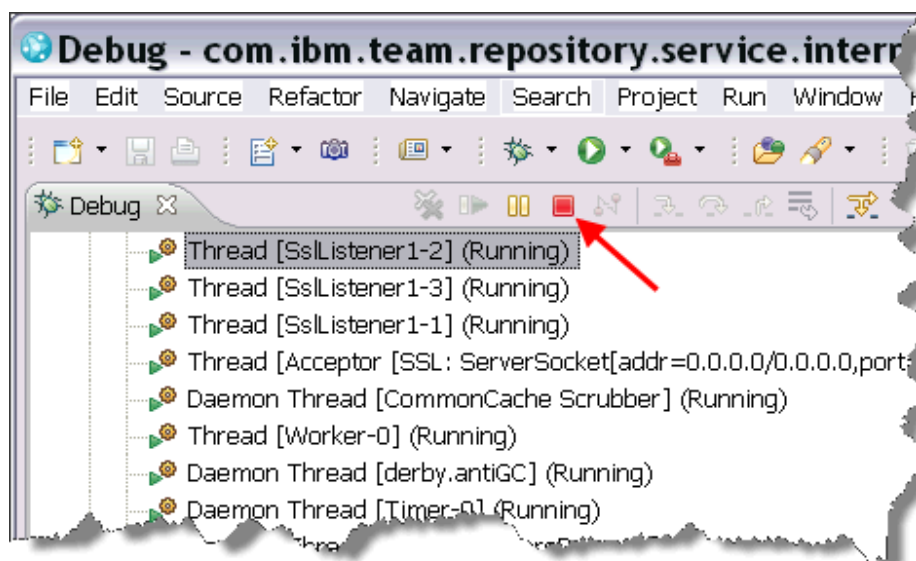
- __d. Click the **Resume** toolbar button to resume execution of the server.



- __4. Complete the test.

- __a. Return to your browser and note that the **Active Services** page is now showing. Close your browser window.

- __b. You can now return to the RTC Eclipse client and terminate the server by clicking the **Terminate** toolbar icon in the **Debug** view as shown here or in the **Console** view.



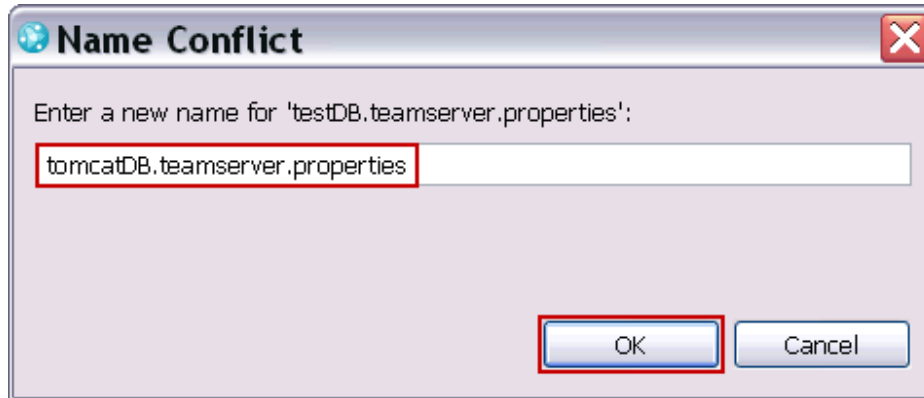
1.7 Setup Jetty Launch Using the Tomcat Repository



This scenario is occasionally useful. For example, you may have some interesting data built up in the Tomcat repository but you have a problem to debug in which you think you will want hot code replace.

- __1. Create a new teamserver.properties file.
- __a. In your RTC Eclipse client, return to the **Plug-in Development** perspective. You may close any editors that are left open.
- 
- __b. In the **Package Explorer** view on the left, expand your **Launches** project, right click the **testDB.teamserver.properties** file and from the menu select **Copy**.
- __c. Right click the **Launches** project and from the menu, select **Paste**.

- ___d. In the **Name Conflict** dialog, change the file name to `tomcatDB.teamserver.properties` and then click **OK**.



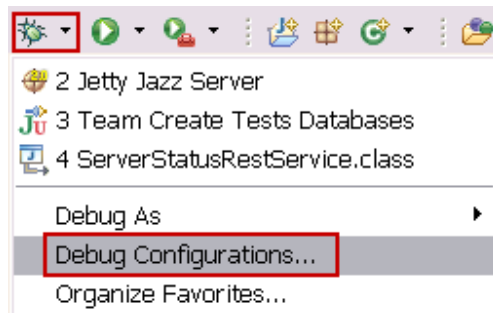
- ___e. Double click the new **tomcatDB.teamserver.properties** file and in the editor that opens change the two properties as shown here. Note the use of forward slashes and not back slashes.

```
com.ibm.team.repository.db.jdbc.location = C:/RTC2002Dev/jazz/server/repositoryDB
com.ibm.team.fulltext.indexLocation=C:/RTC2002Dev/jazz/server/workitemindex
```

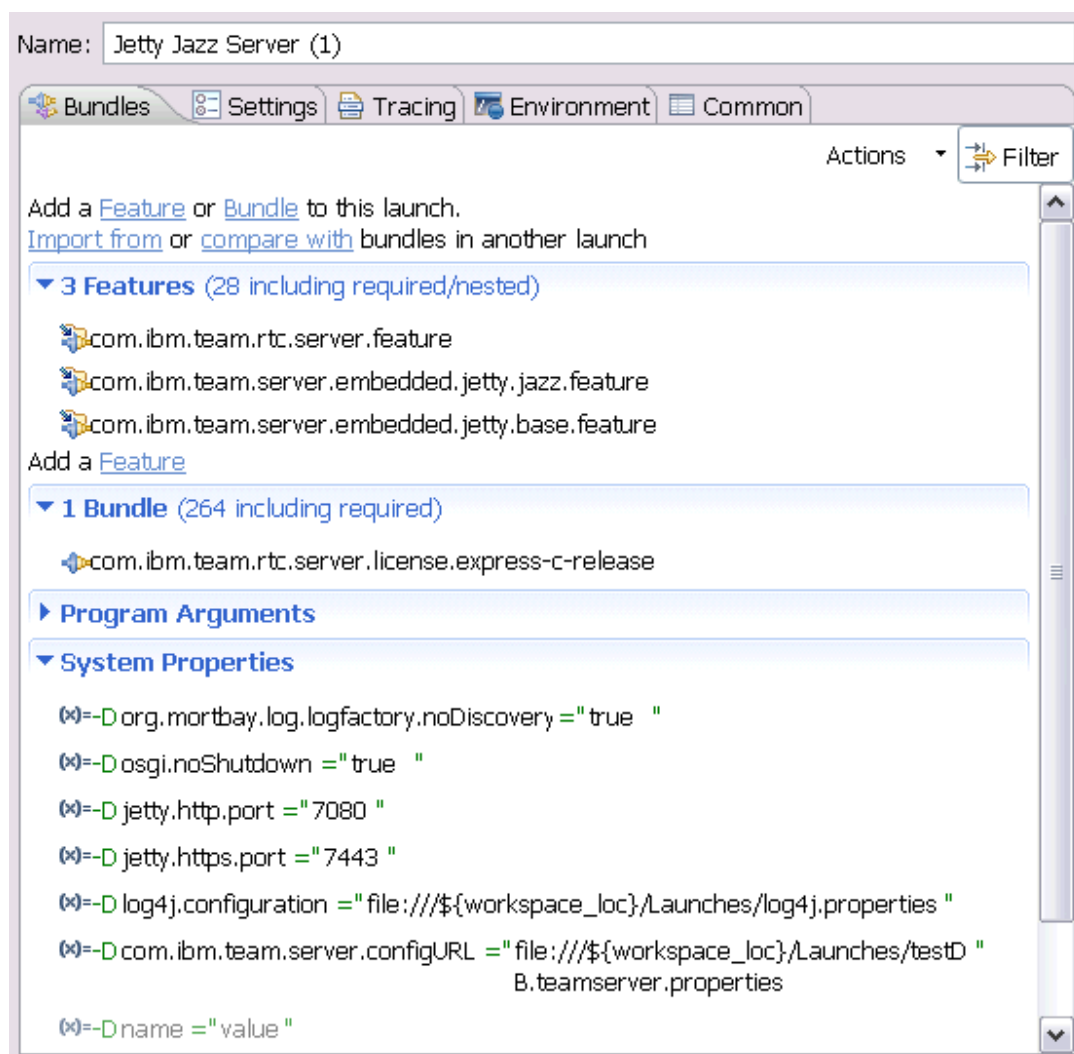
- ___f. Press **Ctrl + S** to save the file and then close the editor.

- ___2. Create a new launch configuration.

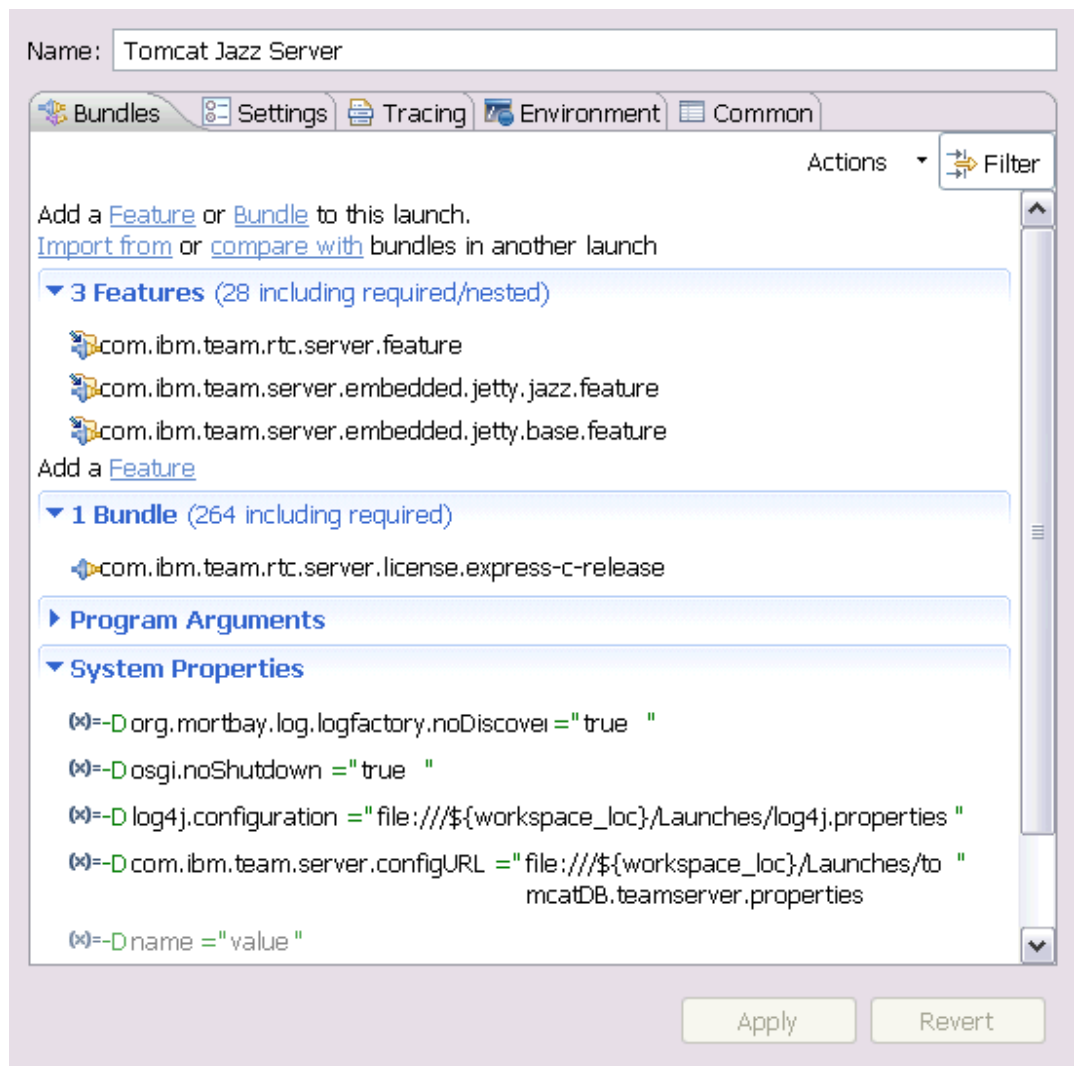
- ___a. Select **Debug Configurations...** from the dropdown menu off the **Debug** toolbar icon.



- __b. In the **Debug Configurations** dialog, expand the **OSGi² Launch** node, right click **Jetty Jazz Server** and from the menu, select **Duplicate**. You will now have a new **Jetty Jazz Server (1)** launch that looks like this.



- ___c. Change the **Name** to `Tomcat Jazz Server` and then select these **System Properties** and remove them: **jetty.http.port** and **jetty.https.port**. Removing these will make the server use the ports configured for the Tomcat server (probably 9080 and 9443). This will prevent both servers from starting at the same time. Finally, click on the value for the **com.ibm.team.server.configURL** property and change it to:
`file:/// ${workspace_loc} /Launches/tomcatDB.teamserver.properties`.
 This will point it to the properties file you created in the prior step. Click **Apply** and the launch configuration should now look like this. Leave the **Debug Configurations** dialog open for the next section.



1.8 Test the Tomcat Jazz Server Launch

- ___1. Launch the server.
- ___a. In the **Debug Configurations** dialog with the **Tomcat Jazz Server** launch still selected on the left, click **Debug**.

- __b. The **Console** view will take focus and log messages will start appearing there. Once you see a message that ends in **Server started**, the server is up and ready to be used.

Note: As before, you may see an exception or two concerning the presence of two versions of an Eclipse plug-in being present. The messages contain the text “**The bundle could not be resolved. Reason: Another singleton version selected**” just before the exception’s stack trace. You can ignore these exceptions.

- __2. Try steps 2 through 4 of section 1.6 again, except this time use port 9443 rather than 7443 when connecting to the server with your browser. It should all behave the same.

1.9 Setup to Debug an RTC Eclipse Client

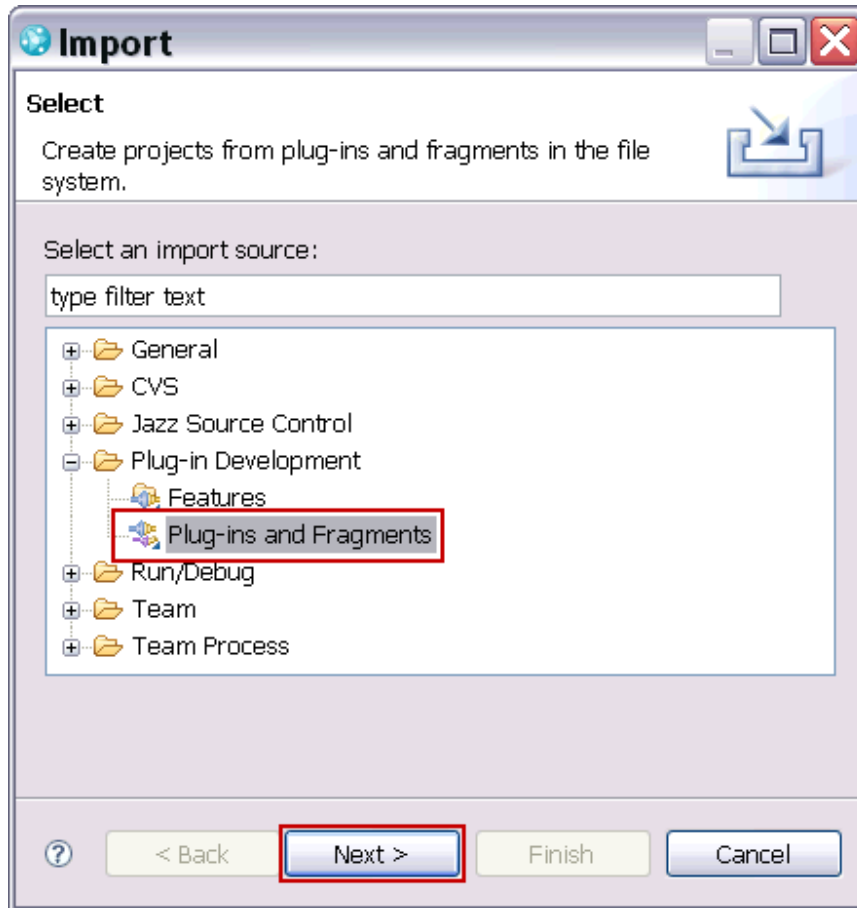


Up to this point you have only been debugging RTC servers. You will sometimes want to extend and debug RTC Eclipse clients too. You will create a launch for that in this section and then test it in the next.

- __1. Import the RTC branding plug-in. Note that once RTC defect 104769 (<https://jazz.net/jazz/web/projects/Rational%20Team%20Concert#action=com.ibm.team.workitem.viewWorkItem&id=104769>) is fixed; this step should not be required.

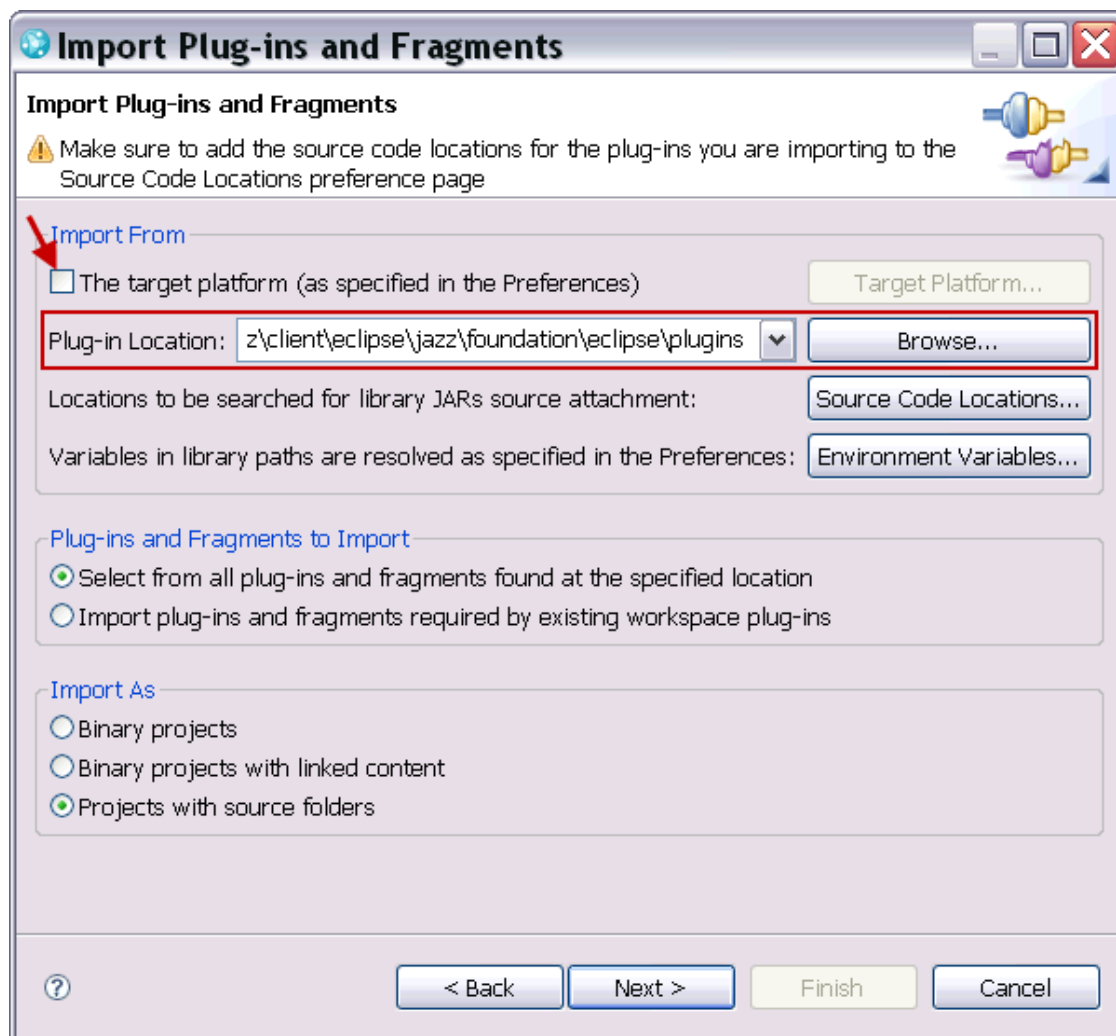
Update: Defect 104769 was fixed in RTC 2.0.0.2 iFix3. If you are using that release or later, you can skip step 1 (and its sub-steps). Continue with step 2.

- ___a. As before, from your RTC Eclipse client menu bar, select **File > Import...** and then in the **Import** wizard, select **Plug-in Development > Plug-ins and Fragments** as shown here and then click **Next**.

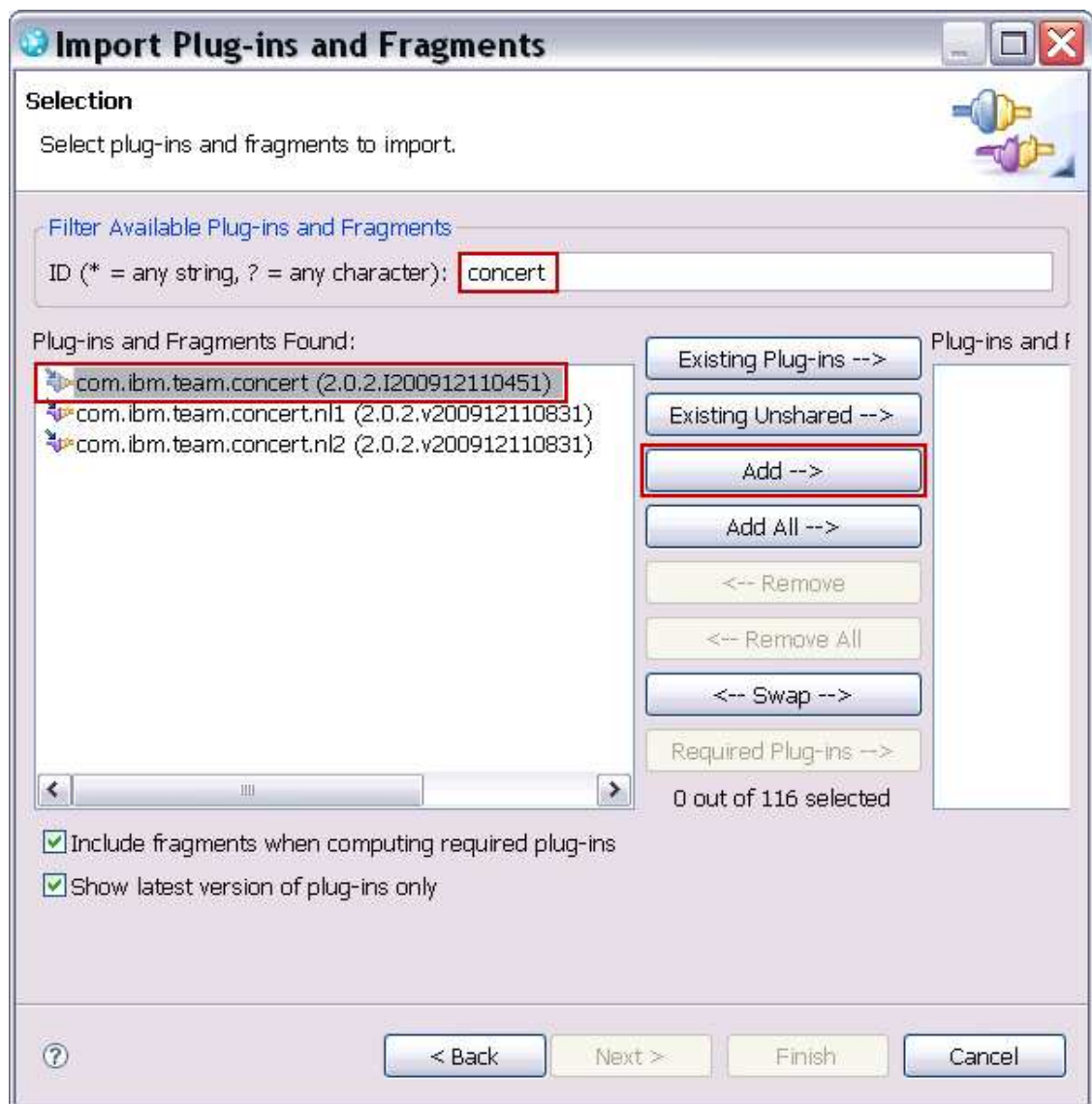


- ___b. This time on the second page of the wizard, make sure your selections match those shown here and then click **Next**. The major difference from last time is the selection of a different place to import from. The **Plug-in Location** field should be set to (use the **Browse...** button to find it):

C:\RTC2002Dev\jazz\client\eclipse\jazz\foundation\eclipse\plugins.

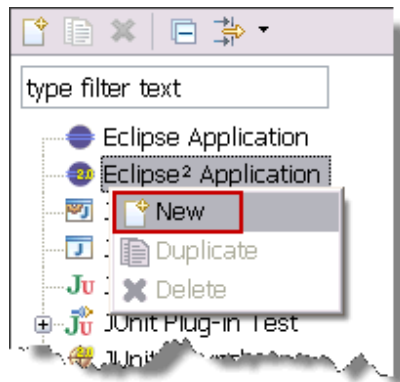


- ___c. On the third page of the wizard, type `concert` into the **ID** filtering field, select the **com.ibm.team.concert** plug-in, click **Add -->** and then click **Finish**.

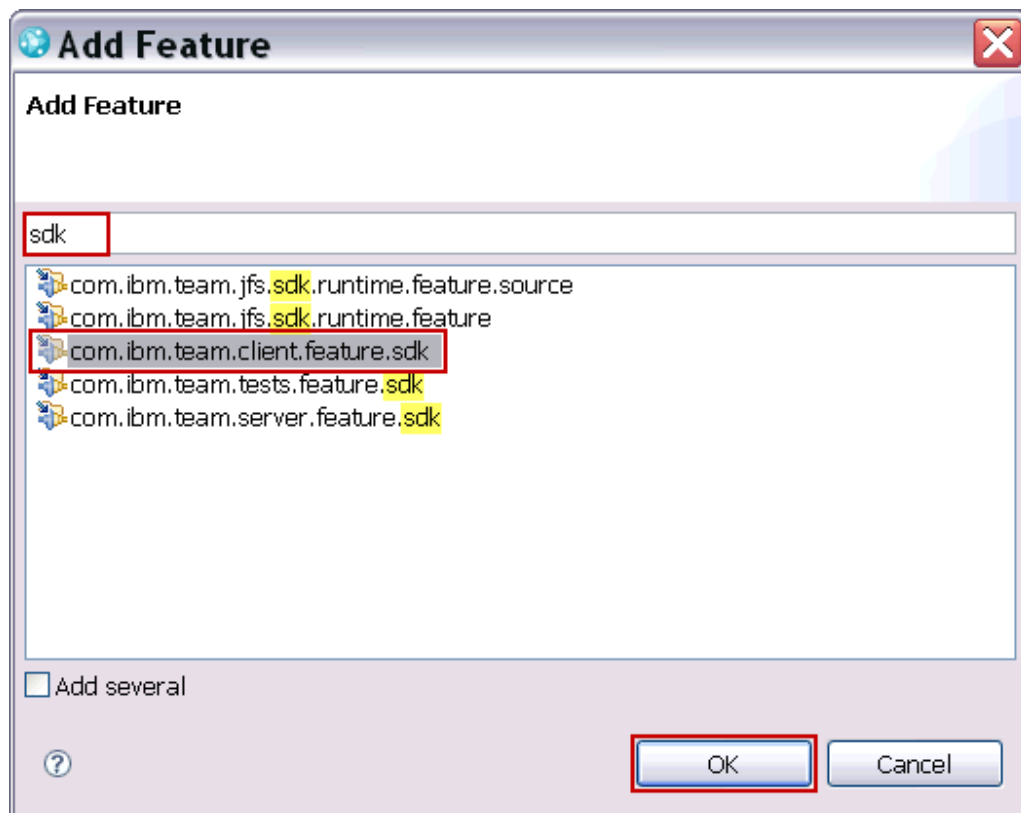


- ___2. Create the launch.
- ___a. In your RTC Eclipse client, once again select **Debug Configurations...** from the **Debug** tool bar icon's dropdown menu.

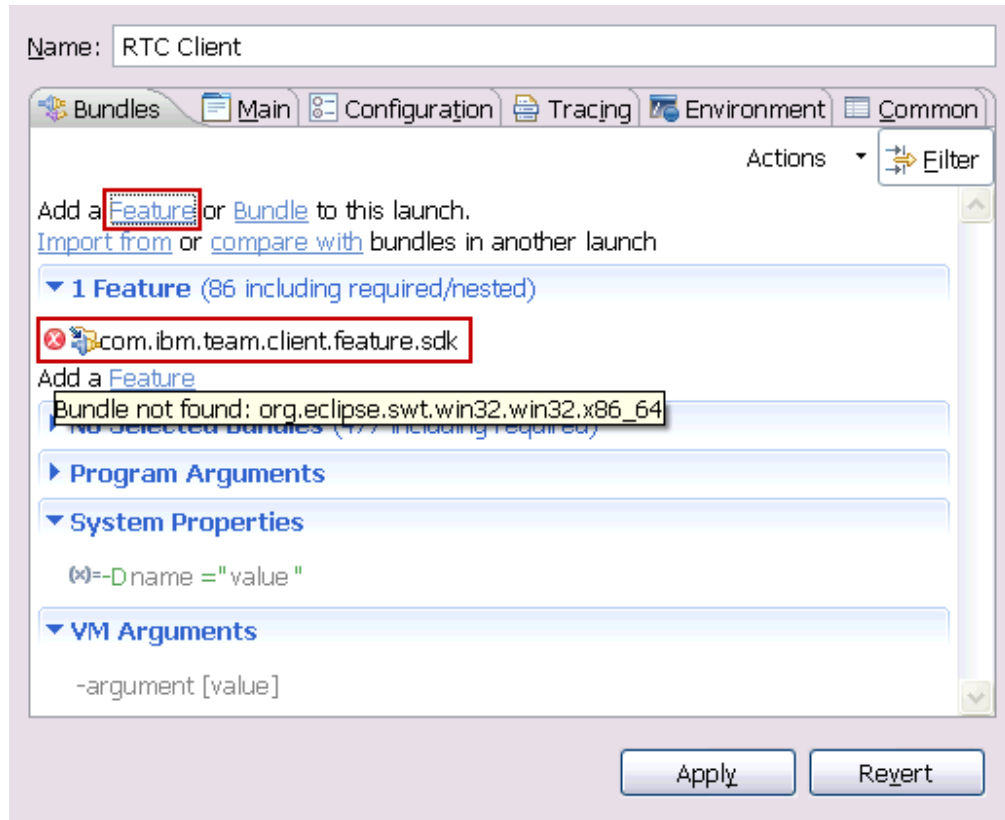
- ___b. In the **Debug Configurations** dialog, right click **Eclipse² Application** and from the menu, select **New**.



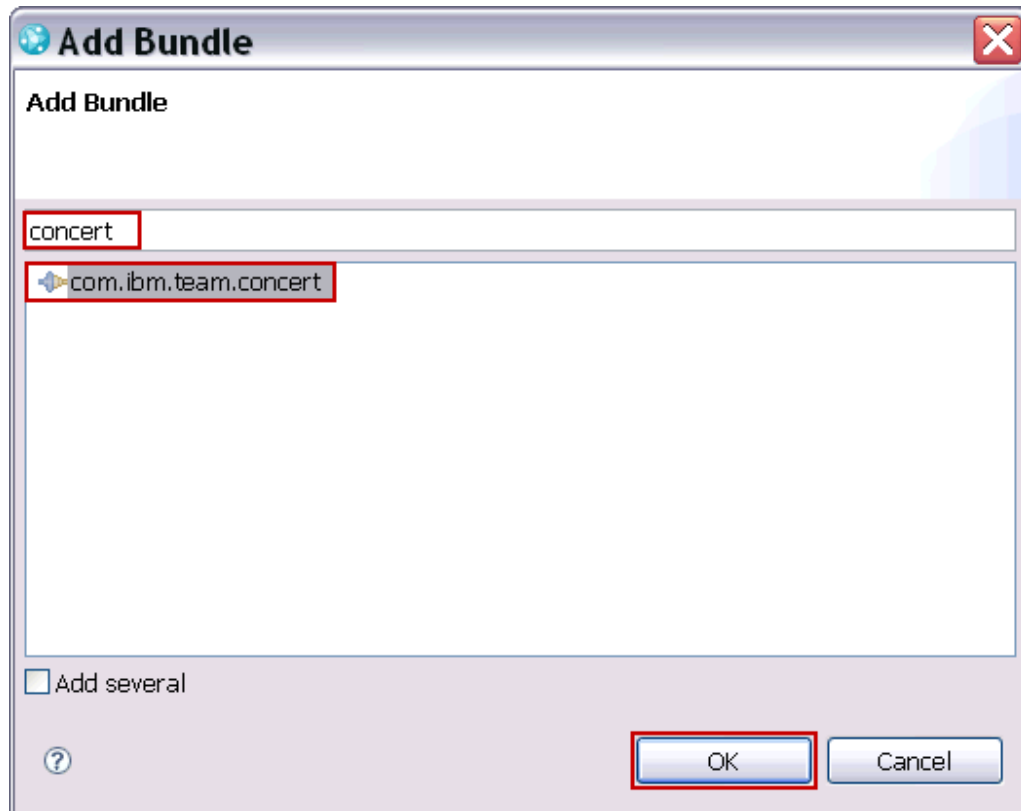
- ___c. A new debug configuration is created and there are a few things you need to change. First change the **Name** to **RTC Client**. Second, click the **Feature** link and then in the **Add Feature** dialog, type **sdk** in the filter field, select **com.ibm.team.client.feature.sdk** from the list and then click **OK**.



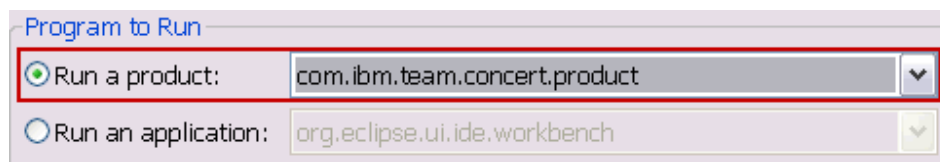
- ___d. The **Bundles** tab of your new launch will now look like this. Note the error message. You can ignore it. The feature based launches support does not currently consider the operating system, windowing system and architecture (os, ws and arch) constraints on included plug-ins and fragments. The bundle it can not find is mentioned by one of the required features that was pulled in but is not really needed at runtime nor is it shipped with RTC.



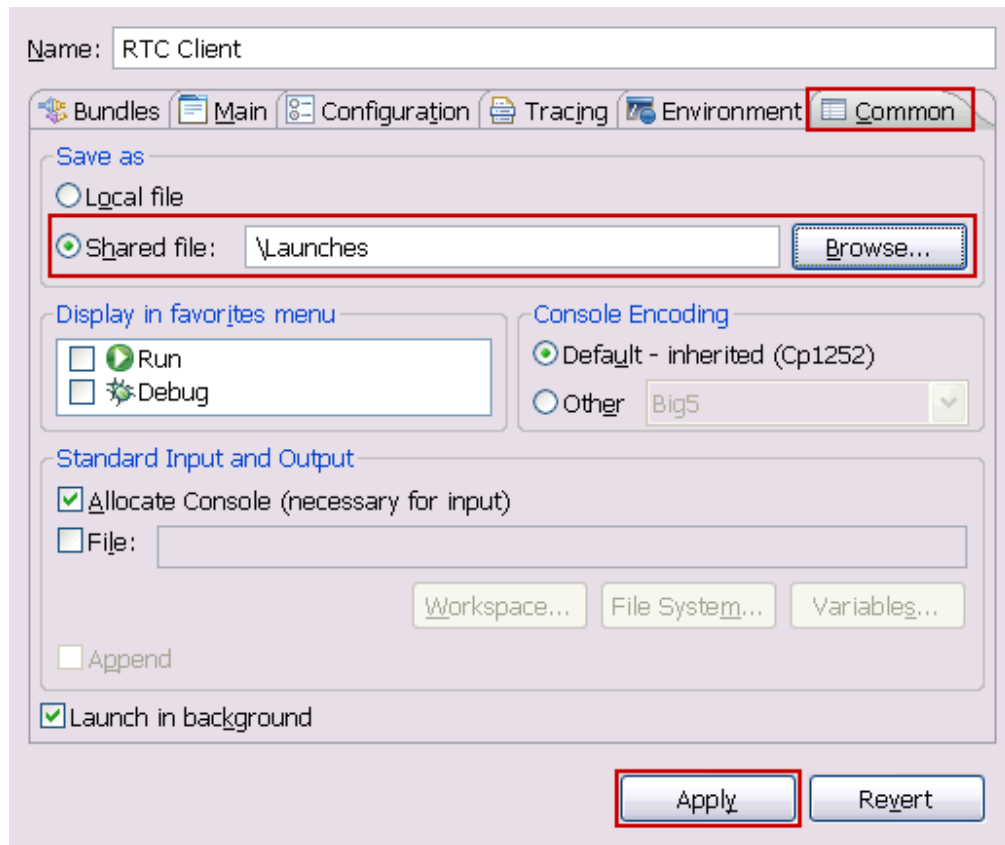
- __e. Click the **Bundle** link and in the **Add Bundle** dialog, type `concert` in the filter field, select `com.ibm.team.concert` from the list and then click **OK**.



- __f. Click the **Main** tab and change the product to run to `com.ibm.team.concert.product`.



- ___g. Click the **Common** tab and then click the **Shared file** radio button and enter `\Launches` (the project you created earlier) into the field. You can use the **Browse...** button to find it. Then, click **Apply**. Now you have a configured launch and it is stored in your Launches project in case you want to be able to easily share it with someone else. Leave the **Debug Configurations** dialog open for the next section.



1.10 Test Your RTC Client Launch

- ___1. In the **Debug Configurations** dialog click **Debug** on the lower right.
- ___2. The RTC Eclipse client will launch and you can use it as you normally would. If you hit a client side breakpoint, your original RTC Eclipse client will surface to handle the debugging.
- ___3. If you launch one of your servers under debug as before, you can create repository connections from your launched client to your launched server and debug both sides of your connection.
- ___4. Close the RTC Eclipse client you just launched under debug.

1.11 Setup for the Remaining Labs

- __1. For the remaining labs, you will be testing primarily using Jetty launches. You will use the Tomcat server once to learn how to deploy the server side extensions to a real server. You will make good use of the Tomcat server, however. The `ExtensionsAndIntegrationsLabCodeRepository-yyyymmdd.tar` file you have contains an RTC repository. The repository has one project area, stream and workspace for this workshop (it may also contain artifacts for other workshops). The stream contains a baseline of the code for each lab. The workspace currently contains the code for lab 2. Right now, you are going to import this repository into the Tomcat server's database. In subsequent labs, you will connect to the project area from the RTC Eclipse workspace you have configured in this lab to obtain the code.
- __a. From the Windows Explorer, copy the tar file to the **C:\RTC2002Dev\jazz\server** folder.
- __b. Make sure the server is stopped.
- __i. If it is not, double click the **server.shutdown.bat** file to stop the server.
- __ii. Switch to the Tomcat window where the server is running. You may need to hit **Enter** after the message "INFO: Failed shutdown of Apache Portable Runtime" appears to completely shut the server down.
- __c. Open a command prompt in the **C:\RTC2002Dev\jazz\server** folder.
- __d. Type the command `repotools -import fromFile=ExtensionsAndIntegrationsLabCodeRepository-yyyymmdd.tar` and then hit enter. Replace `yyyymmdd` with the actual date stamp in the file name.
- __e. Type the command `repotools -rebuildTextIndices` and then hit enter.
- __f. If you are continuing on to lab 2 right now, double click the **server.startup.bat** file to start the server.



You have completed lab 1. You now have a complete develop and debug environment for extending RTC. You have created 4 launch configurations (they can be used for run in addition to debug) which you can use as templates for other launches. You will do some of that in upcoming labs.

Lab 2 Create a Simple Build on State Change Operation Participant



Lab Scenario

You have been assigned to create a new work item save operation participant (or follow-up action). When a Story is changed to the Implemented state, the project's integration build will be run. If the build can not be started, the work item save is stopped.

Note that that follow-up actions run after an operation. There is a similar construct that runs before an operation called an operation advisor (or precondition). They use a different extension point and implement a different interface but are constructed in the same manner.

If your RTC server is not running, start it now

(C:\RTC2002Dev\jazz\server\server.startup.bat).

As part of creating this operation participant, you will also be creating a new Jazz component. It is sometimes possible to create a participant without creating a new component, however; in this case, you will need a component because:

- the participant will be requesting services from other components,
- therefore, the participant must declare dependency on those other components, and
- in order to declare the dependency, the participant itself must be part of a component.

Components generally have 5 parts (each implemented as an Eclipse plug-in project):



- Common – contains interfaces, constants, etc that are common to both the client and server
- Service – contains the server side service implementations
- Client library – contains the client side libraries – these are Java api that can be used in plain Java applications outside the OSGi environment in which Jazz clients and servers typically run.
- Rich client UI – Eclipse or Visual Studio UI components
- Web UI – Extensions to the Jazz web UI for the component

None of these are strictly required to make a Jazz component. In this lab there will be a common, service and rich client UI (Eclipse). For more information on the architecture of a Jazz component see:

<http://jazz.net/library/LearnItem.jsp?href=content/docs/platform-overview/index.html>. For information on how to create more complex components, see: <http://jazz.net/wiki/bin/view/Main/ComponentDevelopment>.

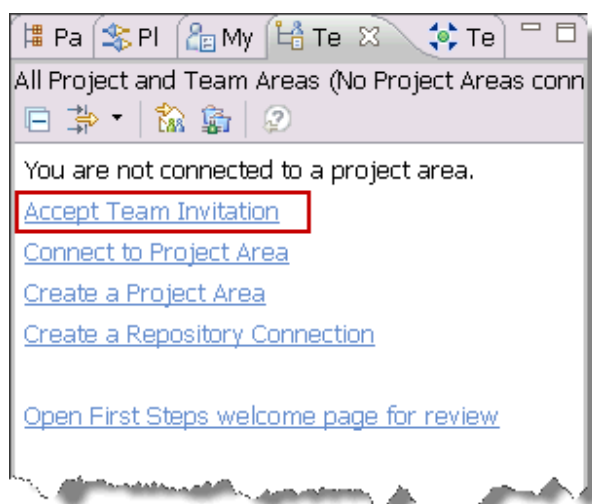
2.1 Create a Basic Server Side Service

- __1. If your RTC development environment is not open, navigate to `C:\RTC2002Dev\jazz\client\eclipse` in the Windows explorer and double click **eclipse.exe**. If prompted to select an Eclipse workspace, select the same one you created in lab one. If you are in a classroom environment where lab one was done for you, select the Eclipse workspace as directed by your instructor. If the **Plug-in Development** perspective is not open, open it now by selecting **Window > Open Perspective > Other... > Plug-in Development** from the menu bar.



If you are running this as part of a formal workshop or are using a VM image, it is possible that the next step of this section has been completed for you. If there is already a repository connection for the ADMIN user and a connection to the “Extension and Integration Workshops” project area in the Team Artifacts view, you can skip to step 3, “Load the lab two code.”

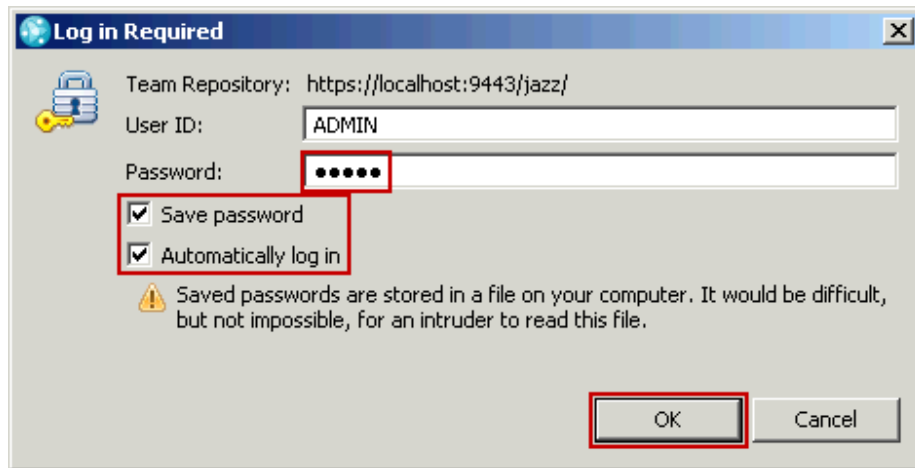
- __2. Connect to the project area.
- __a. On the left, switch to the **Team Artifacts** view and click the **Accept Team Invitation** link.



- __b. In the **Accept Team Invitation** wizard, enter the following in the text field and then click **Finish**.

```
teamRepository=https://localhost:9443/jazz
userId=ADMIN
userName=ADMIN
projectAreaName=Extension and Integration Workshops
```

- ___c. When prompted for a password, enter `ADMIN`. Also, check the **Save password** and **Automatically log in** check boxes. Then click **OK**.

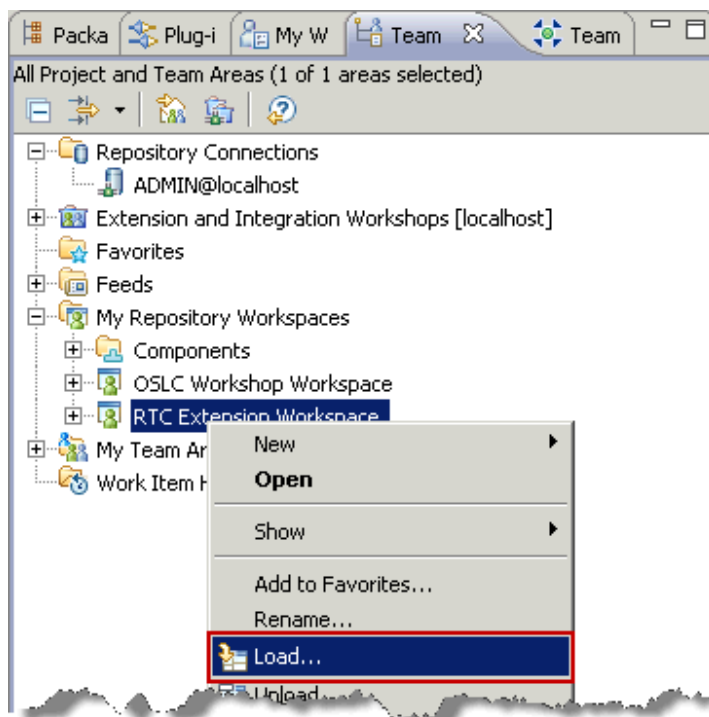


- ___d. If prompted with a **Repository Connection Certificate Problem**, select the **Accept this certificate permanently** radio button and then click **OK**.

- ___e. Close the project area editor that opens.

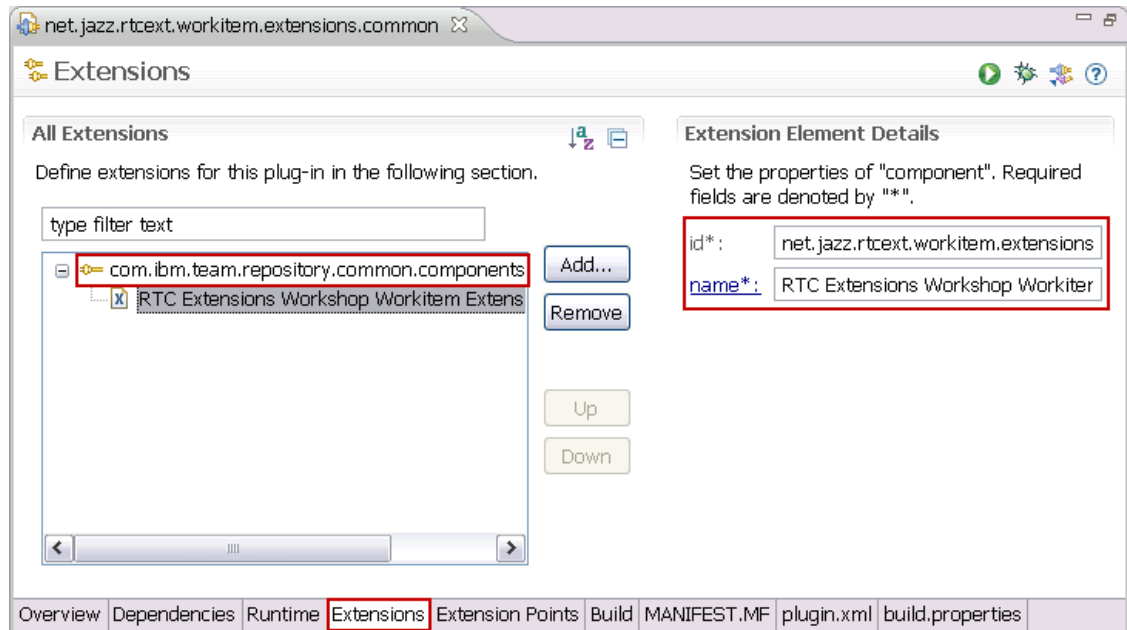
- ___3. Load the lab two code.

- ___a. In the **Team Artifacts** view, expand the **My Repository Workspaces** node, right click the **RTC Extension Workspace** and then select the **Load...** action from then menu.



- ___b. In the **Load Repository Workspace** wizard, click **Finish**.
 - ___c. Verify that there are now three new Eclipse projects in your **Package Explorer** view. Two of these projects define the common (**net.jazz.rtcext.workitem.extensions.common**) and service (**net.jazz.rtcext.workitem.extensions.service**) parts of your component as defined above. In the rest of this lab you will learn about the various parts of this initially simple participant. The third project (**RTC Extension Lab Code License**) contains the license agreement for the sample code you are using in this workshop.
 - ___d. You will also notice in the Package Explorer view, or the **Pending Changes** view, that there are incoming change sets and baselines. Do not accept them. You will make use of them in later labs. If the Pending Changes view is not open, select **Window > Show View > Other...** from the menubar, type `pending` into the filter field and then double click the **Pending Changes** entry.
- ___4. Understanding the common plug-in Eclipse project.
- ___a. If you are just creating operation participants, the common project is usually pretty simple. It defines the component and other items (constants in this case) that are needed by both the server and client side portions of your component. At this time, you only have the server side portion, so the common project is not strictly needed, but in a future lab, you will add the client side portion.
 - ___b. In the **Package Explorer** view, expand the tree for the common project (**net.jazz.rtcext.workitem.extensions.common**) and double click the **plugin.xml** file. The editor that opens presents information from not only the plugin.xml file but also the build.properties and META-INF/MANIFEST.MF files. The content reflects standard Eclipse plug-in practices, for example, including `qualifier` as the last element of the plug-in **Version** on the **Overview** tab (see http://help.eclipse.org/galileo/topic/org.eclipse.pde.doc.user/tasks/pde_version_qualifiers.htm).

- ___c. The most interesting part for your purposes is found on the **Extensions** tab. There is an instance of the **com.ibm.team.repository.common.components** extension point. It uses the id `net.jazz.rtcext.workitem.extensions` and the name `RTC Extensions Workshop Workitem Extensions`. This entry defines your component. Since it uses a repository common extension point, this plug-in also declares a dependency on the `com.ibm.team.repository.common` plug-in on the **Dependencies** tab.



- ___d. Back in the **Package Explorer** view, expand the **src/net.jazz.rtcext.workitem.extensions.common** source package and then double click the **IComponentDefinitions.java** file. This file contains constants that pertain to the component as a whole. In this case there is just a constant for the component's id.

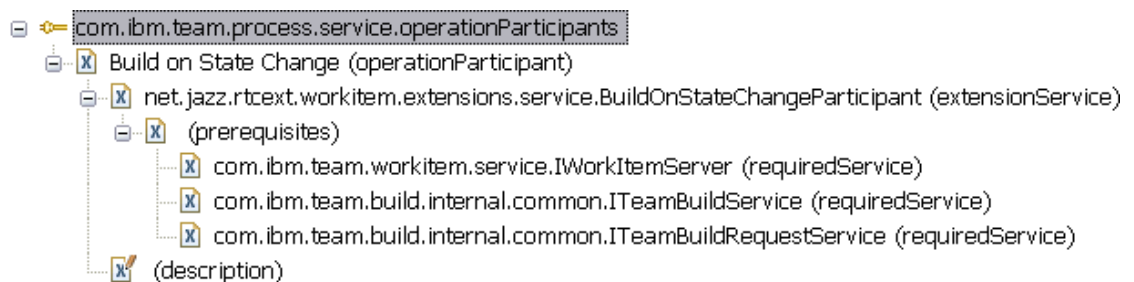
```
/**
 * The component id is used to identify the component to Jazz. It is also
 * used by service definitions to identify which component they belong to.
 */
public static String COMPONENT_ID = "net.jazz.rtcext.workitem.extensions";
```

- ___e. Once again in the **Package Explorer** view, in the same package, double click the **IBuildOnStateChangeDefinitions.java** file. This file contains constants that are particular to the build on state change participant. Right now, it contains just the id for the participant. This will change in future labs.

```
/**
 * The extension id is used to identify the operation participant to Jazz.
 * It is also included in instantiations of the participant in process
 * definitions.
 */
public static String EXTENSION_ID =
    "net.jazz.rtcext.workitem.extensions.service.buildOnStateChange";
```

___5. Understanding the service plug-in Eclipse project.

- ___a. In the **Package Explorer** view, expand the tree for the service project (**net.jazz.rtext.workitem.extensions.service**) and double click the **plugin.xml** file. Once again, there is a set of standard Eclipse plug-in definitions. Also, the most interesting part is once again on the **Extensions** tab. On the left side, you see an instance of the **com.ibm.team.process.service.operationParticipants** extension point. All server side operation participants are defined using this extension point. In the following steps, you will explore most of the nodes in this tree. Note that the tree is a structural editor for the xml that comprises the definition. The raw xml can be seen on the **plugin.xml** tab of the editor. The text in parenthesis on each line is the name of the xml element for that line.



- ___b. Select the **Build on State Change (operationParticipant)** element on the left then the right side of the editor will look like this. The **class** and **operationId** attributes are the two most critical attributes. The class is the Java code that implements the service (more on that soon) and the operationId identifies the Jazz operation for which the participant is valid. In this case, the work item save operation. The **id** attribute identifies this participant definition and is the same as the constant `IBuildOnStateChangeDefinitions.EXTENSION_ID`. You will add a schema in a future lab.

Extension Element Details	
Set the properties of "operationParticipant". Required fields are denoted by "*".	
id*:	net.jazz.rtext.workitem.extensions.service.buildOnStateChange
class*:	net.jazz.rtext.workitem.extensions.service.BuildOnE Browse...
name*:	Build on State Change
operationId:	com.ibm.team.workitem.operation.workItemSave
schema:	Browse...

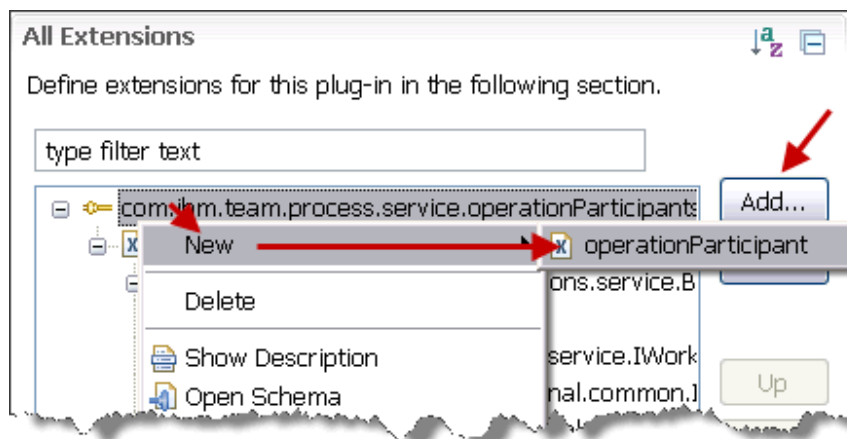
- ___c. Select the **net.jazz.rtext.workitem.extensions.service.buildOnStateChangeParticipant (extensionService)** element on the left and the right side of the editor will look like this. Note that this element is optional. It is only required if the participant will require services from other components. The value in the **componentId** field should look familiar. It is the id given to the component in the common plug-in's plugin.xml file. This ties the participant to the component. When defining an operation participant, the **implementationClass** attribute, is typically set to the same class as the class attribute in the last step and that is the case here. This single class serves as both the participant and a basic service implementation through which the required services will be found. As you will soon see, this is much easier than it sounds.

Extension Element Details

Set the properties of "extensionService". Required fields are denoted by "**".

componentId*:
 implementationClass*:

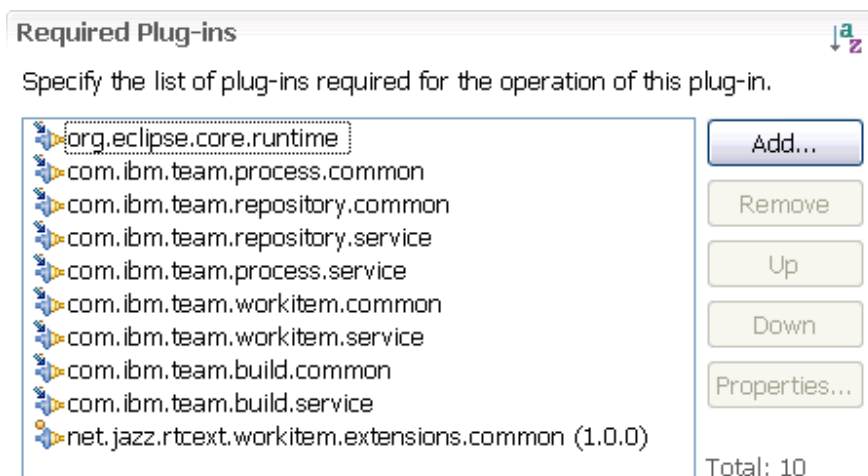
- ___d. If you select the **(prerequisites)** node, you will see that it has no attributes.
- ___e. Skip over the children of the (prerequisites) node for a moment and select the **(description)** node. On the right, you will see the description of the operation participant.
- ___f. Up to now all the work you would need to create this definition is possible from this one place using the **Add...** button and the **New >** cascade menu from the various element's pop-up menus.



- ___g. Unfortunately, this is not the case for the children of the (**prerequisites**) node. You can edit the nodes that are there, but to add a new (**requiredService**) node, you need to edit the xml on the plugin.xml tab. The syntax is pretty simple. Here you see three required services. You will see how these services are used by the participant later.

```
<prerequisites>
  <requiredService
    interface="com.ibm.team.workitem.service.IWorkItemServer"/>
  <requiredService
    interface="com.ibm.team.build.internal.common.ITeamBuildService"/>
  <requiredService
    interface="com.ibm.team.build.internal.common.ITeamBuildRequestService"/>
</prerequisites>
```

- ___h. As you may have guessed, this service plug-in has many more plug-in dependencies than the common plug-in. There are dependencies on process for the operation participant extension itself and on other components for the services the participant will use. Here they are from the **Dependencies** tab.



- ___6. Understand the code within the service plug-in Eclipse project

- ___a. Back in the **Package Explorer** view, expand the **src/net.jazz.rtext.workitem.extensions.service** source package and then double click the **BuildOnStateChangeParticipant.java** file. This file contains the participant implementation. There are several interesting parts to this class. First, note the class javadoc comment. The first paragraph repeats the description you saw in the plug-in.xml file. The remaining text is critical to understand for anyone implementing operation participants, that is:

- It is critical to understand that operation participants are managed as singletons by the process component. Therefore, their methods, most notably the run method must be reentrant. Operation participants must not rely on any instance state variables (i.e. non-static fields).

-While rare, it is occasionally the case that the complexity of the operation to be performed and the number and interactions of methods and their data interdependencies will present a case where the use of instance state variables is highly desirable. In this case, another class will need to be defined and an instance of that class created for each invocation of the run method. The run method can then delegate the operation to the instance of this second class. This second class can use instance state variables for its implementation.

- ___b. Next, note the declaration of the class. The class implements the **com.ibm.team.process.advice.runtime.IOperationParticipant** interface. All operation participants implement this interface. It defines the run method. The class also extends the **AbstractService** class. Only participants whose extension definition in the plugin.xml file contains the optional **extensionService** element have to extend this class. Recall that you needed the extensionService element to declare the prerequisite services. Even though the AbstractService class is indeed abstract, there are no abstract methods left that this class has to implement. This class will, however, use methods from AbstractService to locate the prerequisite services.

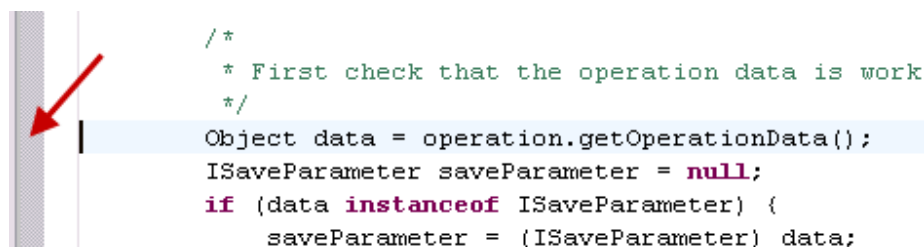
```
public class BuildOnStateChangeParticipant extends AbstractService implements
    IOperationParticipant {
```

- ___c. Note that a default constructor is required for an operation participant but not explicitly defined here. The default constructor added by the Java compiler is typically sufficient for an operation participant.
- ___d. Take a look at the **run** method javadoc comment. Note that the participant is called for each work item save operation but only if the participant has been configured a project area or team area's work item save operation behavior. You will see that configuration later. The rest of the comment describes each parameter in detail. This initial implementation only makes use of the **operation** parameter.
- ___e. Note the first comment block in the body of the run method. The point here is that there are often several checks your code will make in order to decide if there is action to take. In deciding which order to check them, take into account the cost of the check (put more expensive checks later) and the likely hood that the check will make your code decide there is nothing to do (put more likely to fail checks earlier). Ideally, you want fast and likely to fail checks first and slower less likely to fail checks later. Of course, sometimes you will be faced with slow likely to fail or fast unlikely to fail checks and it will be a bit more difficult to decide on an ordering. The order of checks here is:
- ___i. Is the data passed to the participant really for a work item save operation? This should always pass but it is a best practice to make this check first.

- __ii. Has the state id (the workflow state) changed? Note that the case of saving a new work item is handled in these lines. In the case of a new work item, the **oldState** (the full state data of the work item, not the workflow state) will be null. And in the last line, note that **Identifier<T>#equals(null)** always returns false and the overall test will pass so that one could have the work item type's initial state be the target state.

```
IWorkItem oldState = (IWorkItem) saveParameter.getOldState();
if (oldState != null) // New work item check.
    oldStateId = oldState.getState2();
if ((newStateId != null) && !(newStateId.equals(oldStateId))) {
```

- __iii. Is the work item of the type in which the participant is interested? Right now the work item type id is hard coded to the Story type from the Scrum template. You will change that later.
- __iv. Is the work item now in the state (workflow state) in which the participant is interested? Right now the work item state id is hard coded to the Story type's Implemented state (it does not look like it with the word tested at the end, but it is). You will change that later.
- __f. If all those checks pass, a build request is made by calling the participant's **build** method. You will look at that next. Note that the build definition id is also hard coded. That will also change later.
- __g. Conceptually, the build method is pretty simple. There are two lines (using the team build service) to find the build definition and two lines (using the team build request service) to request a build for that definition. The key element at this point is the comment between the two sets of lines. Notably, that there are things that can go wrong here that are not being handled. That will be corrected in the next lab.
- __h. So there you have a pretty simple participant that boils down to a few simple status checks in the run method and four lines of code to request a build. There is one more thing to do before leaving this editor. That is, set a breakpoint at the first line of the run method. You will step through it several times in this lab. Double click in the margin next to the first line of the run method to set the breakpoint. A small blue circle will appear after you double click.

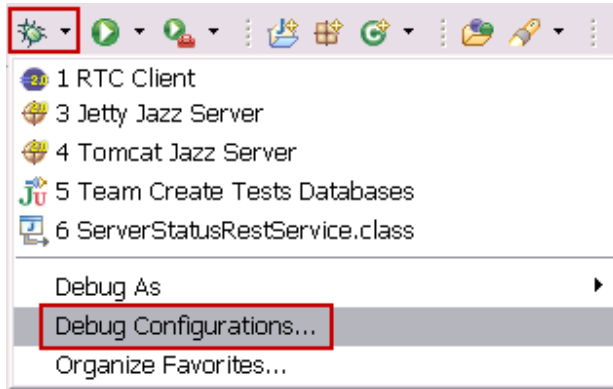


```
/*
 * First check that the operation data is work
 */
Object data = operation.getOperationData();
ISaveParameter saveParameter = null;
if (data instanceof ISaveParameter) {
    saveParameter = (ISaveParameter) data;
```

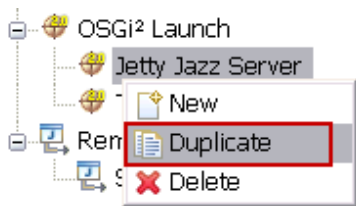
2.2 Launch the Server for Debug Using Jetty

___1. Create the launch configuration.

___a. From the **Debug** toolbar dropdown, select **Debug Configurations...**

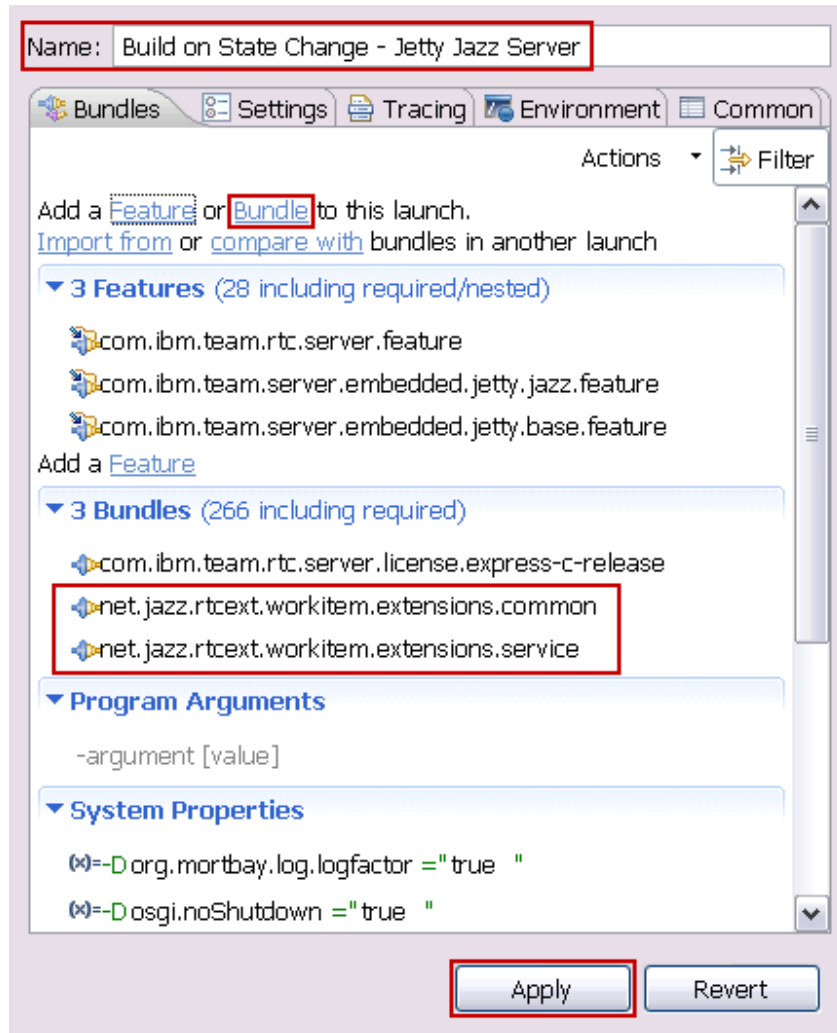


___b. In the **Debug Configurations** dialog, expand the **OSGi² Launch** tree and right click the **Jetty Jazz Server** configuration and then from the popup menu, select **Duplicate**. Note that you are not changing the existing launch but creating a copy of it. You should keep the original launch around unchanged to use as a known working base from which to create other launch configurations.



___c. Change the **Name** of the new configuration to **Build on State Change - Jetty Jazz Server**.

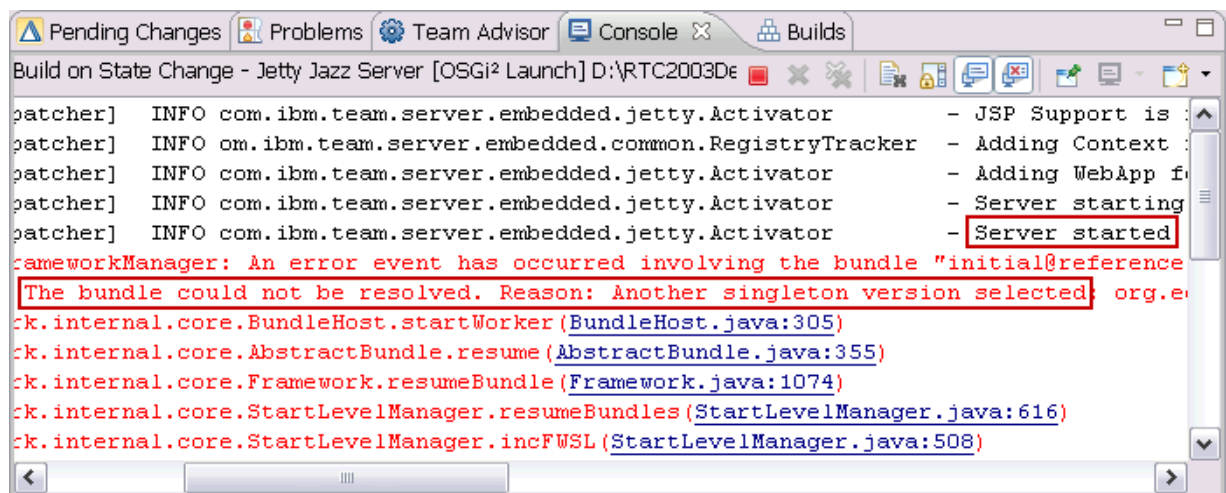
- ___d. Add your participant's two bundles to the configuration. Click on the **Bundle** link and in the **Add Bundle** dialog, type `rttext` in the filter field, select the common plug-in and then click **OK**. Repeat, but select the service plug-in this time. Your launch configuration should look like this.



- ___e. Click **Apply** to save your changes but do not close the dialog.
- ___2. Launch the server.
- ___a. Click **Debug** at the bottom of the **Debug Configurations** dialog.

- __b. The **Console** view will take focus and log messages will start appearing there. Once you see a message that ends in **Server started**, the server is up and ready to be used.

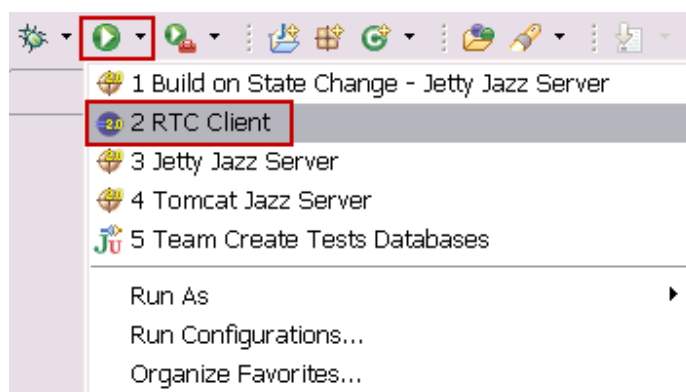
Note: You may see an exception or two concerning the presence of two versions of an Eclipse plug-in being present. The messages contain the text “**The bundle could not be resolved. Reason: Another singleton version selected**” just before the exception’s stack trace. You can ignore these exceptions.




- __c. The next time you want to debug this server configuration, you will be able to click a shortcut to it on the dropdown of the Debug toolbar icon. You will not need to open the **Debug Configurations** dialog.

2.3 Launch an RTC Client and Connect to the Server

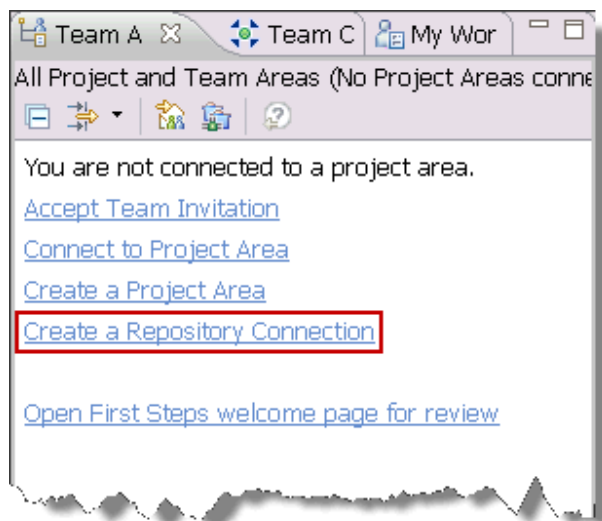
- __1. Launch the RTC Client.
- __a. From the dropdown menu of the **Run** toolbar icon, select **RTC Client**. Note that you are just running the client and not debugging. The same launch configuration can be used for both. You will debug a client in a future lab.



__b. The RTC Eclipse client will start up and should look familiar. If the **Welcome** screen is showing, minimize it via this () button near the top or right of the window.

__2. Connect to the debug server.

__a. You will be in the **Work Items** perspective and the **Team Artifacts** view will be on the left. In the Team Artifacts view, click the **Create a Repository Connection** link.



- ___b. In the **Create a Jazz Repository Connection** wizard, set the **URI** to `https://localhost:7443/jazz` and the **User ID** and **Password** fields to `ADMIN`. Note that it is a '7' and not a '9' in the URI. Then, click **Finish**.

Create a Jazz Repository Connection

Jazz Repository Connection

Create a new Jazz repository connection.

Location

URI: * `https://localhost:7443/jazz`

Name: `localhost`

Authentication

User ID: * `ADMIN`

Password: •••••

☒ Remember my password

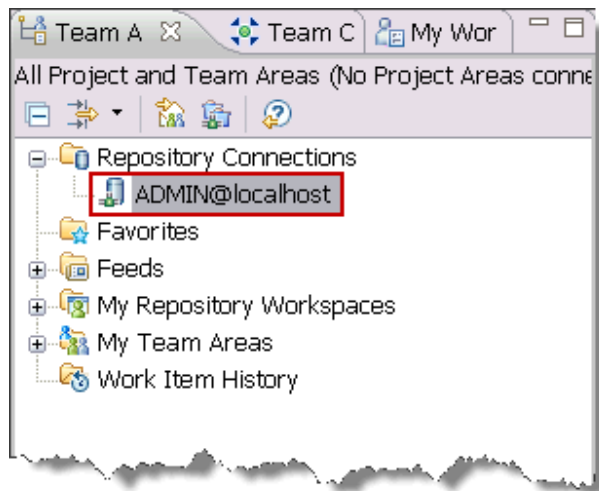
☒ Automatically log in at startup

Advanced

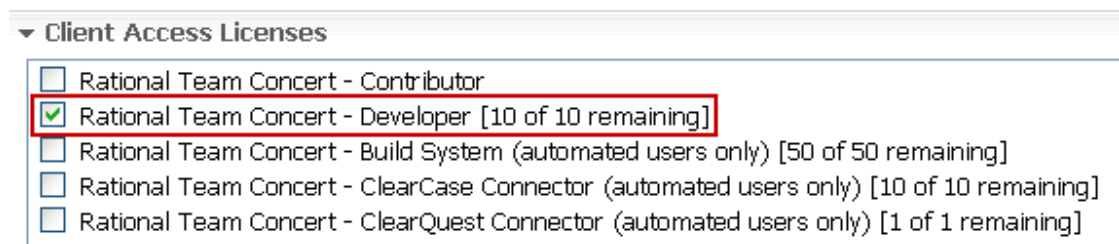
Connection Timeout (in seconds): `480`

Finish **Cancel**

- ___c. You will now have a repository connection in your **Team Artifacts** view.

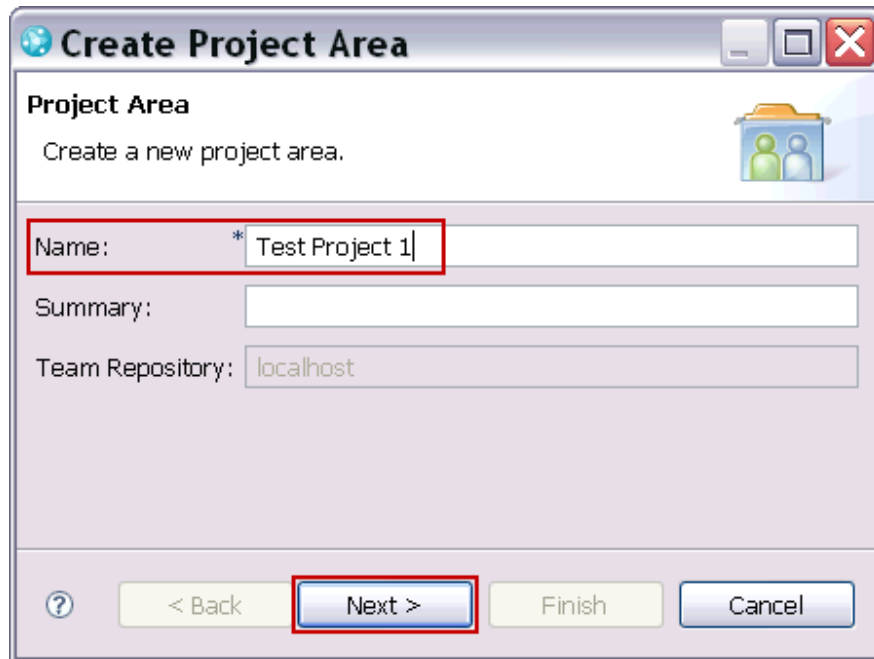


- ___d. Right click your repository connection and from the pop-up menu select **Open My User Editor**. In the user editor that opens, find the **Client Access Licenses** section at the lower right and make sure the **Rational Team Concert – Developer** checkbox is selected. Save and close the editor. You will use the ADMIN user id for several operations that require a developer client access license.



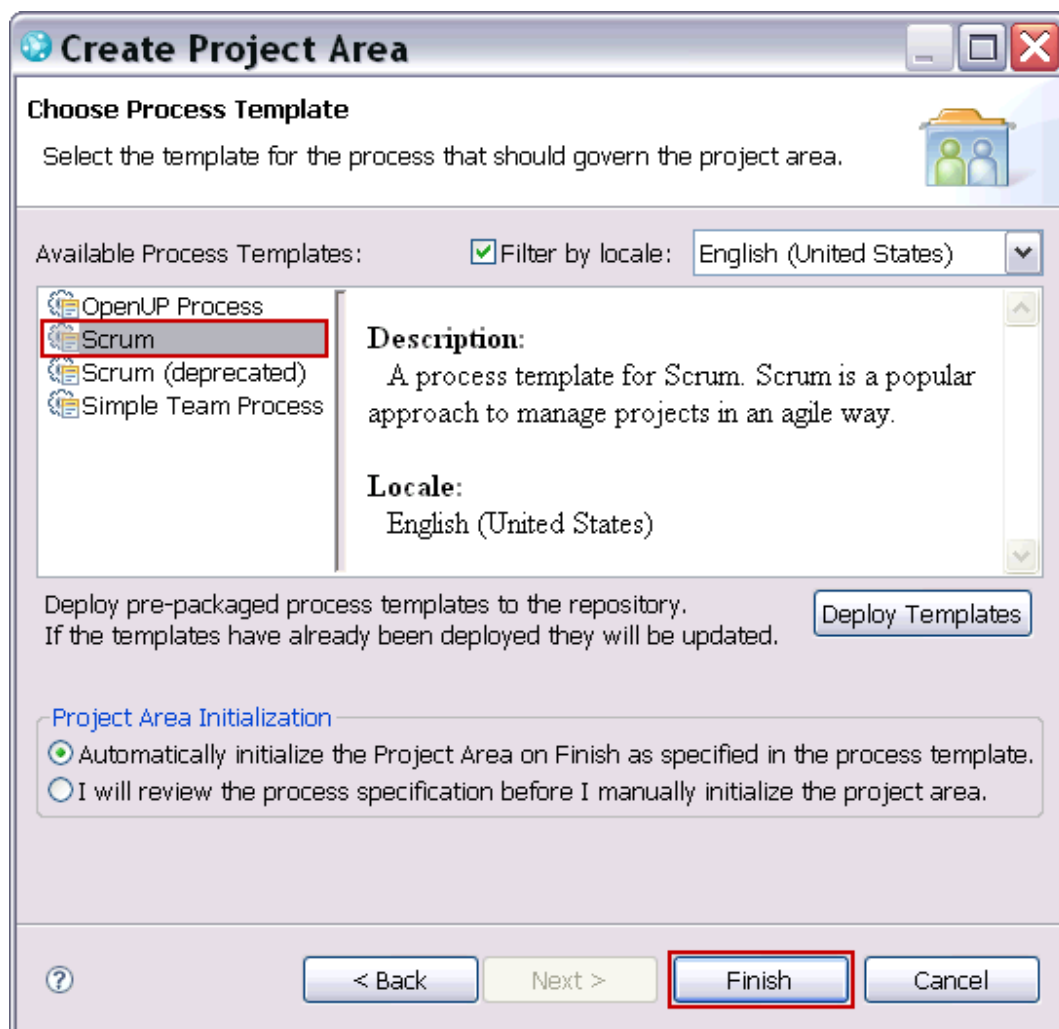
2.4 Edit the Process to Use the Participant

- ___1. Create a project area.
 - ___a. Right click your repository connection and from the pop-up menu select **New > Project Area**. In the **Create Project Area** wizard, set the **Name** to `Test Project 1` and click **Next**.



The screenshot shows a Windows-style dialog box titled "Create Project Area". The dialog has a title bar with standard minimize, maximize, and close buttons. Below the title bar, the text "Project Area" is displayed, followed by the instruction "Create a new project area." and a small icon of a folder with two people. The main area of the dialog contains three input fields: "Name:" with a red box around it containing the text "* Test Project 1", "Summary:" with an empty text box, and "Team Repository:" with a text box containing the value "localhost". At the bottom of the dialog, there is a row of buttons: a help button (question mark in a circle), a "< Back" button, a "Next >" button with a red box around it, a "Finish" button, and a "Cancel" button.

- ___b. On the second page of the wizard, click **Deploy Templates** button. This operation may take a bit of time. When it completes, you will be on the next page of the wizard. Select **Scrum** on the left and then click **Finish**. When the operation completes and the project area editor opens, leave the editor open for the next couple steps.



- ___2. Add ADMIN as a member of the project area.
- ___a. On the **Overview** tab of the project area editor, expand the **Members** section and click **Add...**

▼ Members

Roles determine a user's permissions as well as any preconditions and follow-up actions that are run for project and team operations. The roles assignments below are also valid in all the project's team areas. Unless configured otherwise, all users in the repository play the 'default' role.

Name	Process Roles

Add...

Create...

Remove

Process Roles...

- ___b. In the **Add Team Members** wizard, Type A into the **Enter user name** field and then click **Search**. Then, select **ADMIN** in the **Matching users** list, click **Select** (moves ADMIN to Selected users) and then click **Next**.

Add Team Members

Users
Select the users to be added to the team.

Enter user name:
(Enter a space to search on word boundaries or * for a full wildcard search)

A Search

Matching users (1 user found):

ADMIN

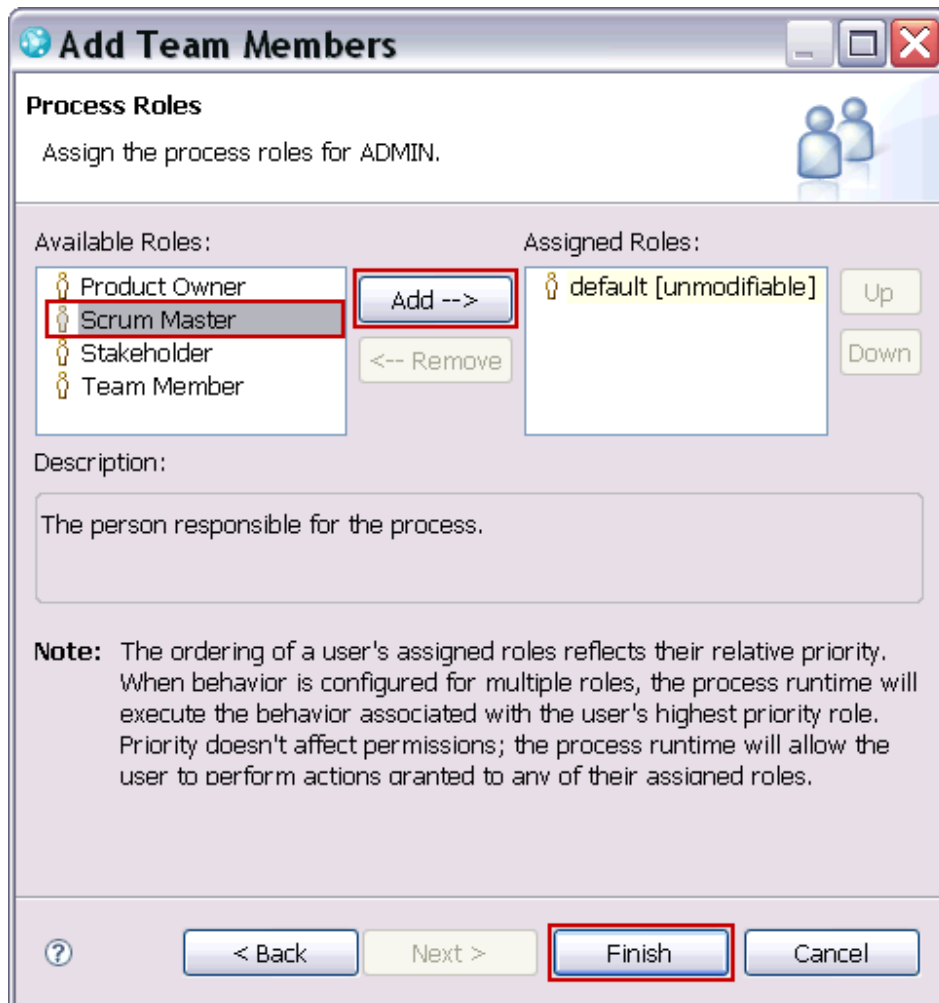
Email: ADMIN - ID: ADMIN

Selected users:

Select Deselect

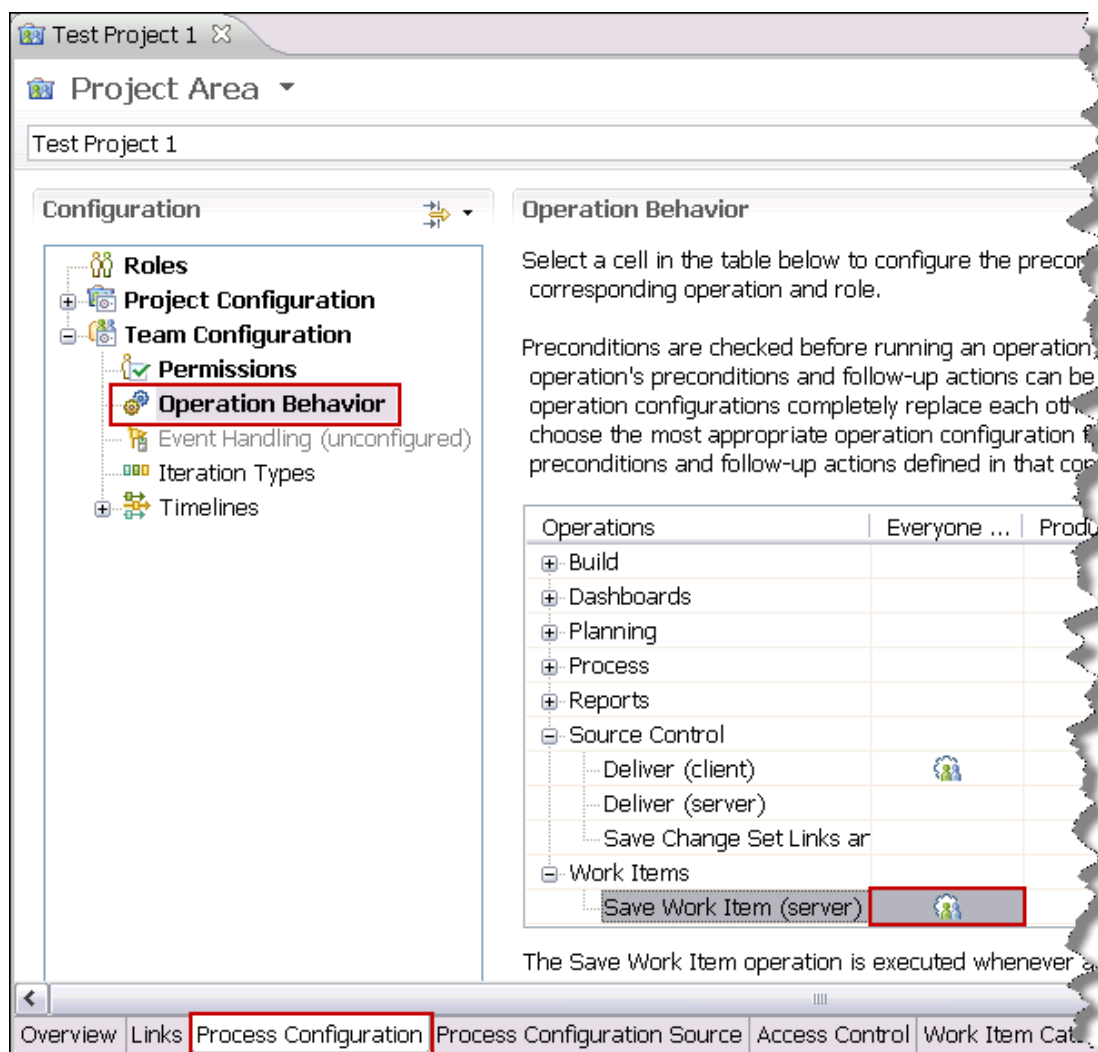
< Back Next > Finish Cancel

- ___c. On the second page of the wizard, select **Scrum Master** on the left, click **Add -->** (moves the selection to the right) and then click **Finish**.



- ___d. Back on the project area editor's **Overview** page, click **Save** (at the upper right) but leave the editor open for the next step.

- ___3. Add the build on state change participant to the work item save operation.
- ___a. Switch to the **Process Configuration** tab and then on the left, expand the **Team Configuration** tree then select **Operation Behavior**. Then, on the right, scroll down to the **Work Items > Save Work Item (server)** operation and select the **Everyone (default)** column next to it as shown here.



- __b. Scroll down to find the **Follow-up actions** section on the right. Initially, the list will be empty. Click **Add...** then on the **Add Follow-up Actions** dialog, select **Build on State Change** (your new participant!) and click **OK**. Build on State Change will now be in the list and when it is selected, the window will look like the following image. Finally, click **Apply changes** and then click **Save** at the upper right of the editor.

☒ Preconditions and follow-up actions are configured for this operation

☐ Final (ignore customization of this operation in child team areas)

Preconditions (6 available):

Name: Build on State Char ☐ Fail if not installed

Description:

When the specified work item type changes to the specified state, the specified build will be requested.

No specific configuration options

Follow-up actions (1 available):

Build on State Change

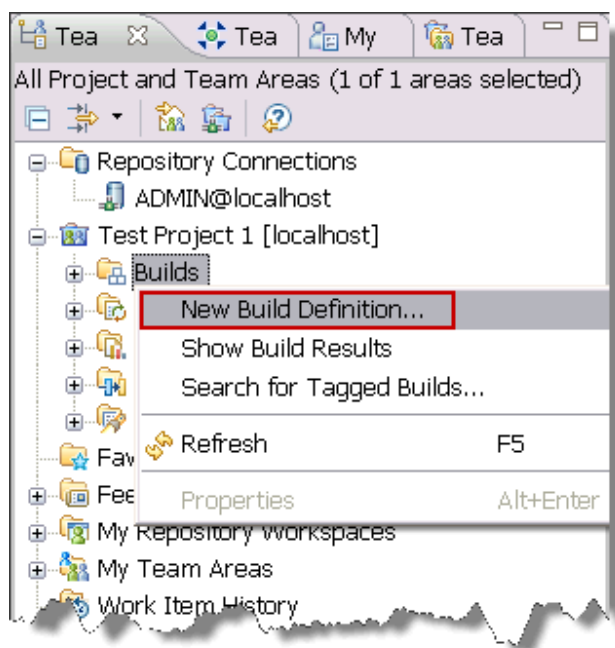
Apply changes **Revert to last applied state**

- __c. You may now close the project area editor and any other editors that may still be open.

2.5 Trigger the Participant

- __1. Create the "our.integration.build" build definition. You just need a simple build definition to test the participant. The build does not need to run properly. The participant just needs to make requests for it.

- ___a. In the **Team Artifacts** view, expand the **Test Project 1** node, right click **Builds** and then click **New Build Definition...**



- ___b. In the **New Build Definition** wizard, make sure **Create a new build** is selected and then click **Next**. On the second page of the wizard, change the **ID** to `our.integration.build`, make sure **Ant - Jazz Build Engine** is selected and then click **Finish**.

New Build Definition

General Information

Choose an ID, description, and build template for the new definition.

ID:

Description:

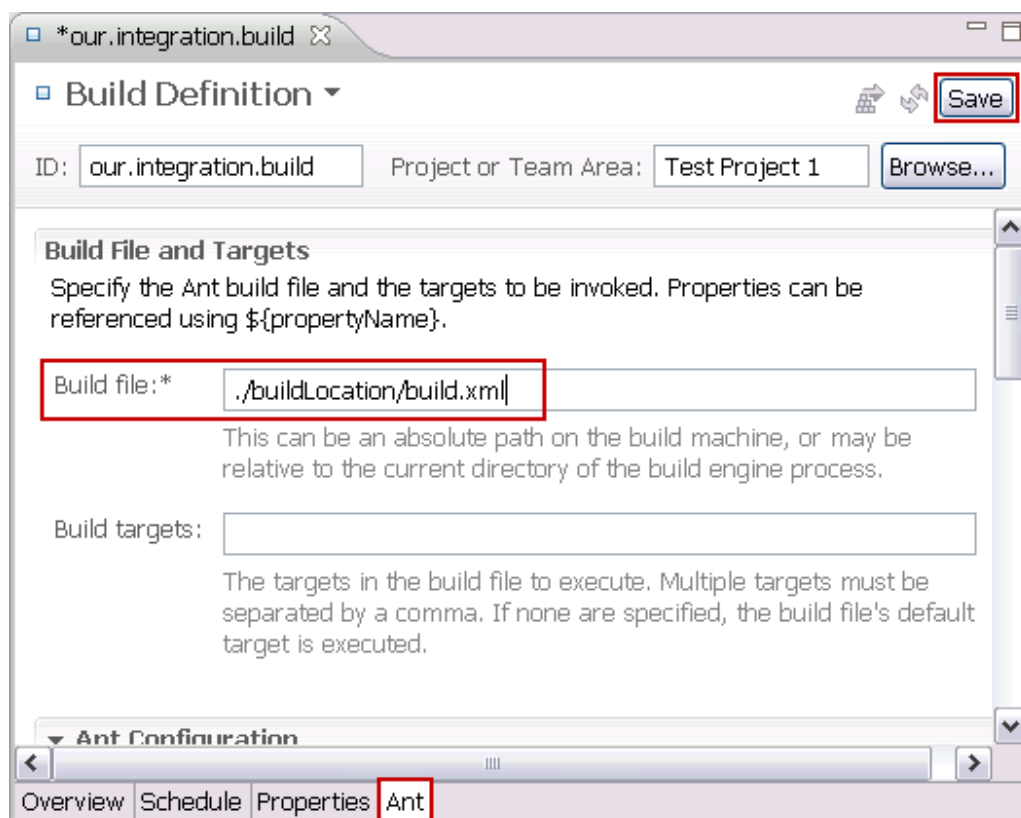
Available build templates:

- ☒ Ant - Jazz Build Engine
- ☐ Command Line - Jazz Build Engine
- ☐ Generic
- ☐ Maven - Jazz Build Engine

Description:

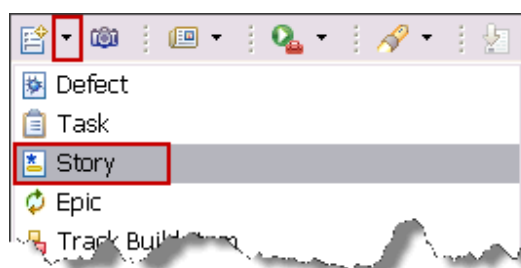
? Help < Back Next > **Finish** Cancel

- ___c. In the build definition editor that opens, switch to the **Ant** tab, and enter a path for the **Build file** and then click **Save**. You may now close the editor. Note that the build file does not exist and any path will work for the current purpose. If you wish, you can use the **Build file** path shown (./buildLocation/build.xml). Also note that a default build engine is created at this time and is associated with your new build definition. This actually is important. If there was no build engine for your build definition, the participant's request for a build would fail.



- ___2. Create a Story work item.

- ___a. Click the dropdown menu arrow next to the **New Work Item** toolbar icon and then click **Story**.



- ___b. In the new work item editor that opens, set the two required fields and shown here and then click **Save** in the upper right corner.

* <11:17:08>: <untitled> ✕

Story <11:17:08> ▾

Summary: * As a user, I want to build stuff

Details

Type: Story ▾

Filed Against: * Test Project 1 ▾


Story Points: 0 pts ▾

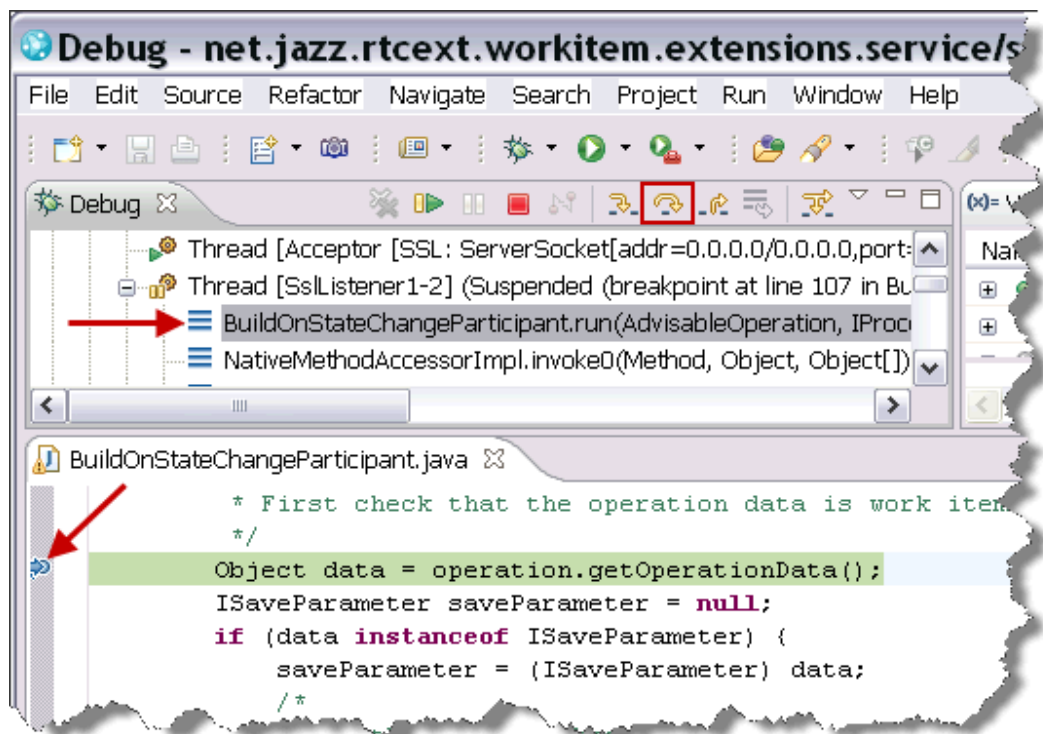
Progress: No Work


Progress: -- Estimated: --

Project Area: Test Project 1

Save

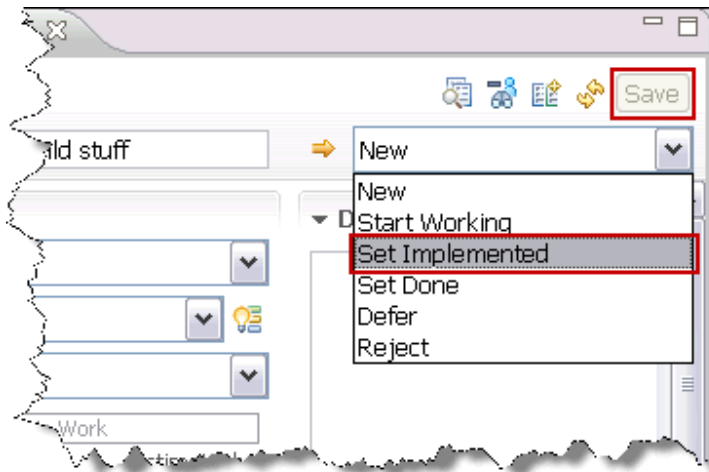
- ___c. The breakpoint you set earlier is now hit. The RTC Eclipse client in which you were studying the code will now surface (if asked about switching to the debug perspective, click **Yes**). If it does not surface, you probably minimized it earlier. In this case, it will be flashing in the Windows taskbar. Click it in the taskbar to surface the debugger. You should see something like this. Step through the run method using the **Step Over** button or F6. The check for the target state will fail and the run method will exit without requesting a build. After that check fails, be sure to click the resume button ().





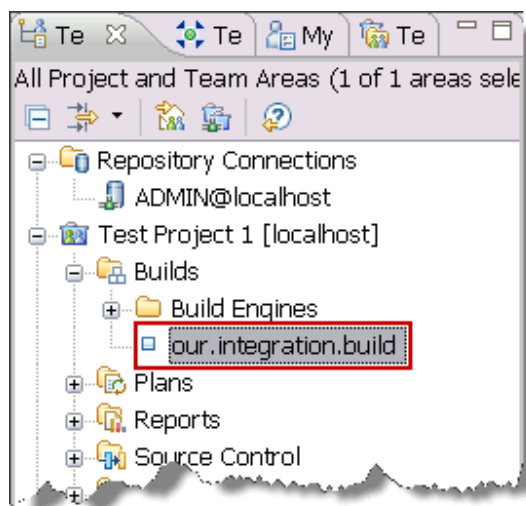
- ___d. Switch back to the launched RTC Eclipse client where you created the work item. Your work item will be successfully saved, and will be in the **New** state. If it shows a failure due to timeout, close the editor without saving, recreate the Story and when the breakpoint hits, just use the resume button ().

___3. Move the Story to the Implemented state.

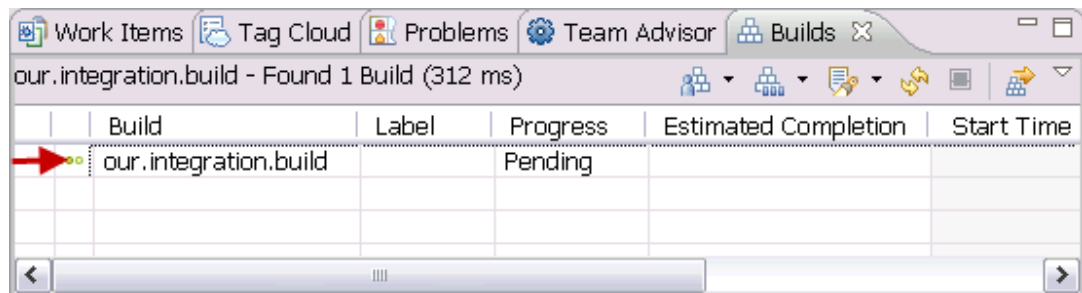
- ___a. At the upper right portion of the work item editor, select **Set Implemented** and then click **Save**.



- ___b. Once again the breakpoint is hit and your debugger surfaces (or you need to click it in the Windows taskbar). Step through the code again. When you get to the call to the build method, use the Step Into button (). You can then step through the four lines that request the build and then click the resume button ().
- ___c. Switch back to the launched RTC Eclipse client where you created the work item. Your work item will be successfully saved. In the **Team Artifacts** view, double click the **our.integration.build** build definition.

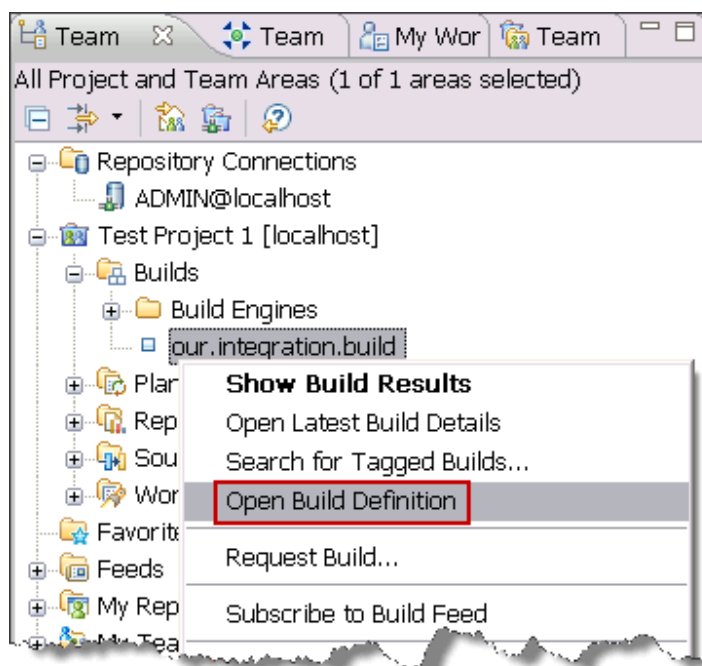


- __d. The **Builds** view opens showing the build request the participant just submitted.

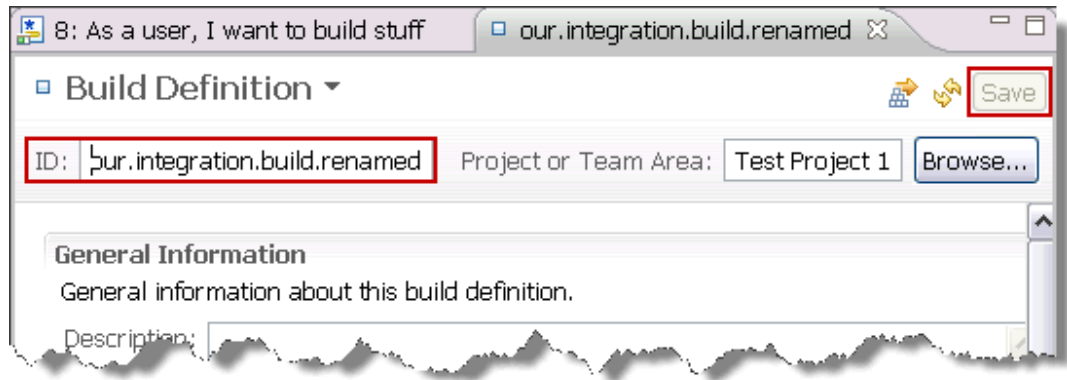


2.6 Rename Build Definition and Try Again

- __1. Rename the build definition.
- __a. In the **Team Artifacts** view, right click the **our.integration.build** build definition and then click **Open Build Definition**.

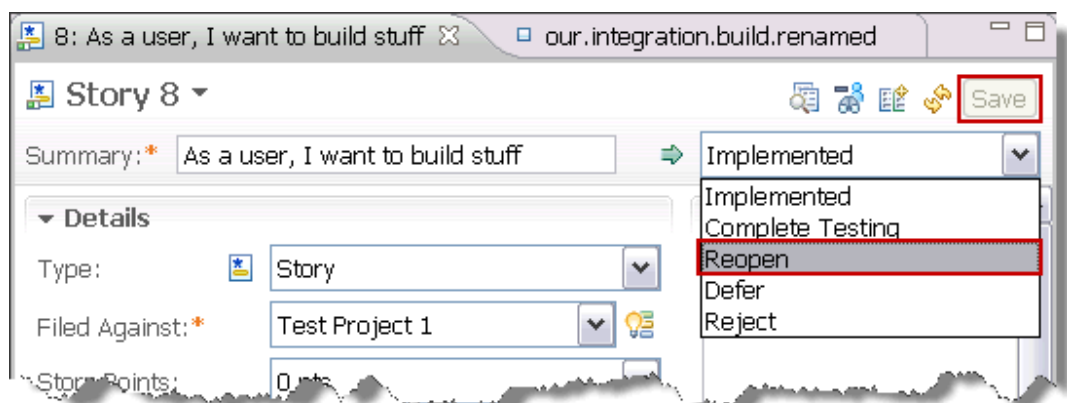


- ___b. In the build definition editor change the **ID** to `our.integration.build.renamed` and then click **Save**. Do not close the editor as you will want to rename it back soon.

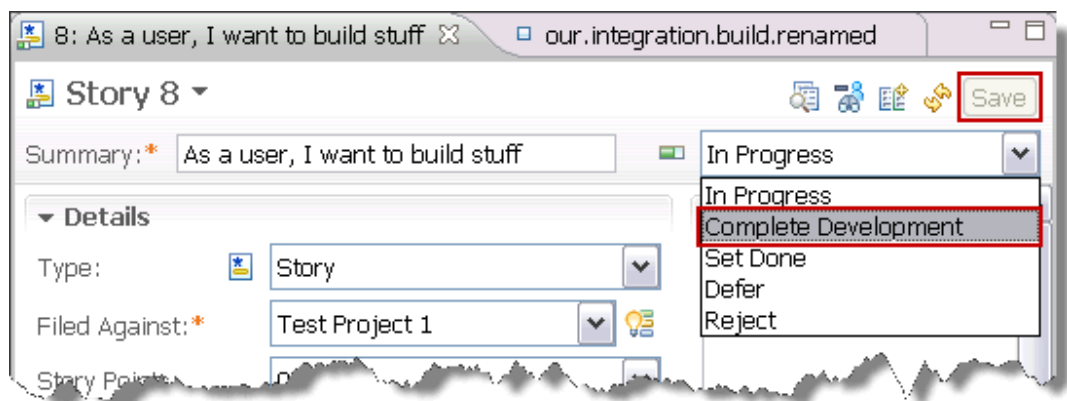


- ___2. Move the story to the Implemented state again.

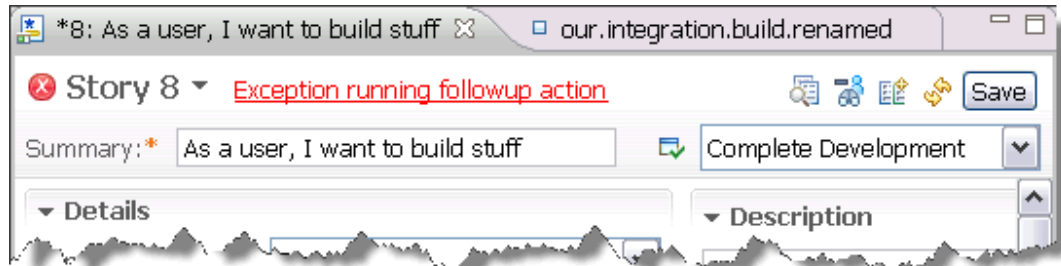
- ___a. Switch back to the work item editor and select **Reopen** from the state dropdown and then click Save. When the debugger surfaces, just click the resume button (). You are not to the interesting bit yet.



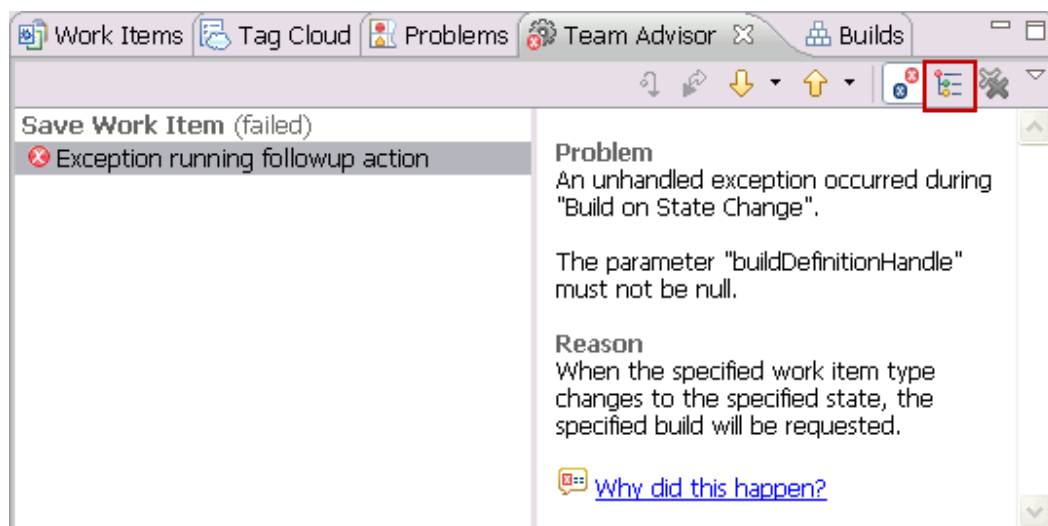
- ___b. Again in the work item editor, select **Complete Development** from the same dropdown and click **Save** again.



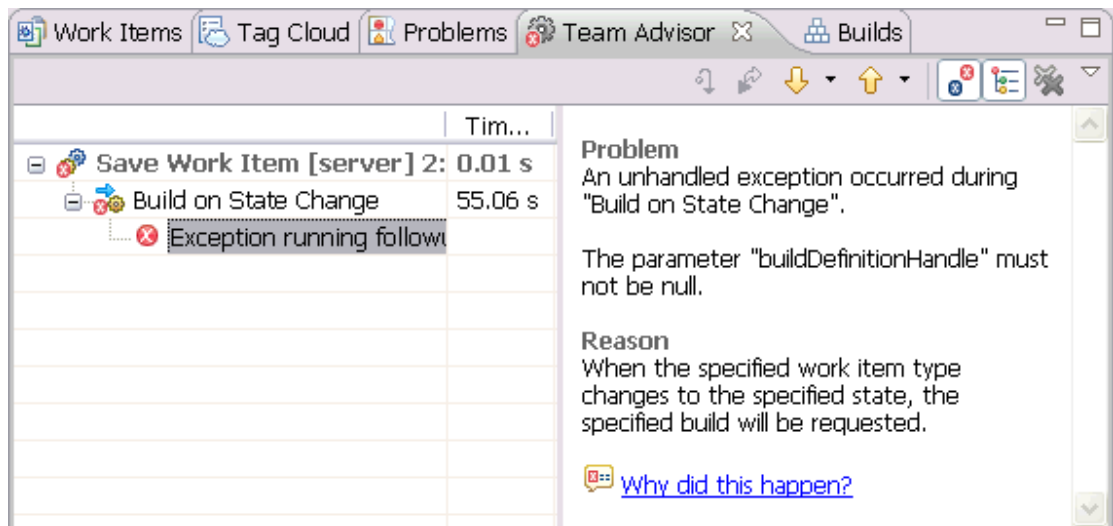
- ___c. This time, when the debugger surfaces, use the step over button to get to the build method call and then use the step into button. Step through the build method and note the major difference this time. The call to get the build definition returns null and the request of the build throws an exception. Click the debugger's resume button. Then switch back to your work item editor and note the red at the top, "Exception running followup action". It is actually a link to the Team Advisor view. Click it now.



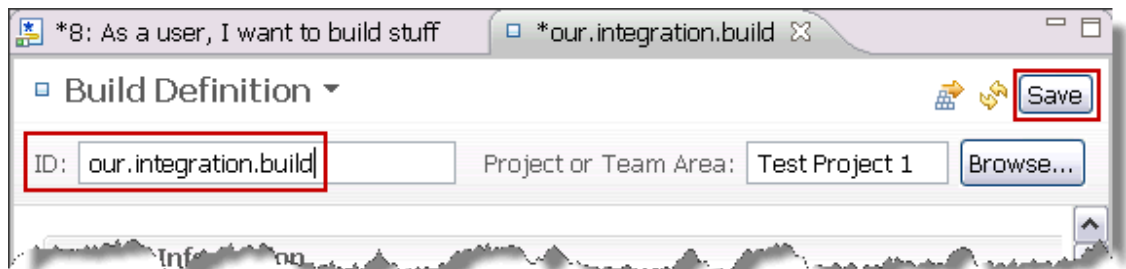
- ___d. The **Team Advisor** view appears with more information on the error. Click the **Show Detail Tree** button.



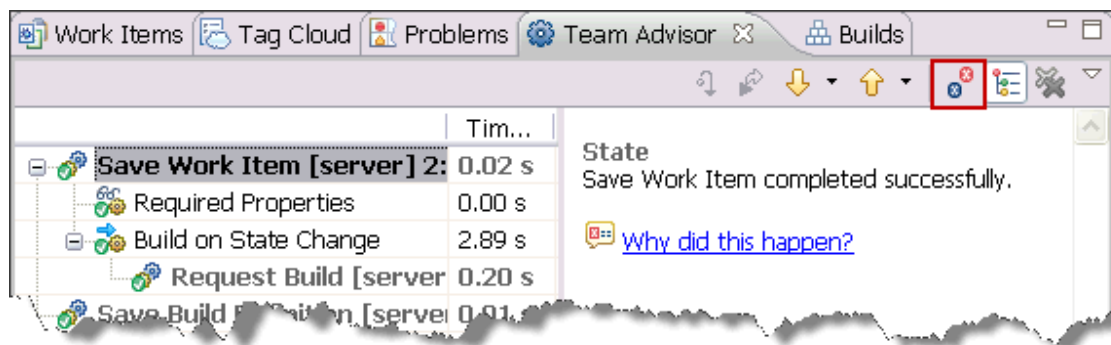
- ___e. The left side of the view changes to show the structure of the error condition. Click the nodes on the left to see what information is available. It is clear that better information would be helpful. For example, a messages stating that the participant was looking for a particular build definition but could not find it would make it much easier to fix the problem. In the next lab, you are going to work on this.



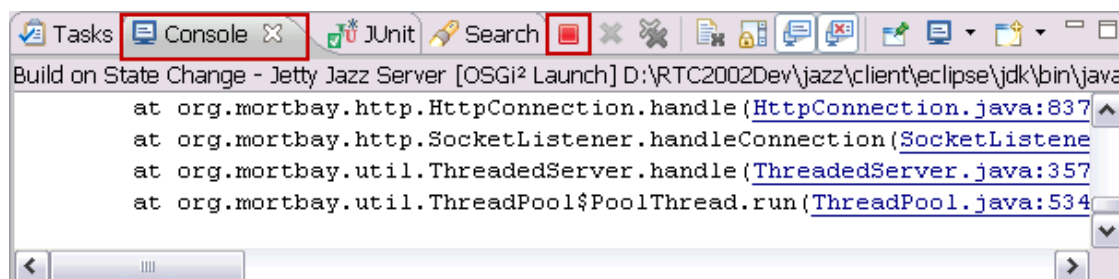
- ___f. Switch back to the build definition editor and change the ID back to `our.integration.build` and click **Save**.



- ___g. Switch back to the work item editor and click **Save**. When the debugger surfaces, you can step into the build method again or just hit resume. Once you do resume, the work item save should complete okay. Return to the work item editor to confirm this. If you go to the **Team Advisor** view and turn off the **Show Failures Only** filter (see highlight below), you can browse the results of this successful operation. Also, if you refresh the **Builds** view, you will now see two pending build requests.



- ___3. Close down the launched client and server.
- ___a. Close the launched RTC Eclipse client where you were working with the Story and build definition.
- ___b. Back in the original RTC Eclipse client, go to the **Console** view and click the **Terminate** icon.



You have completed lab 2. You now have your first functional but not entirely perfect server side operation participant. In future labs, you will improve the error handling and make the work item type, state and build id configurable.

Lab 3 Add Error Handling



Lab Scenario

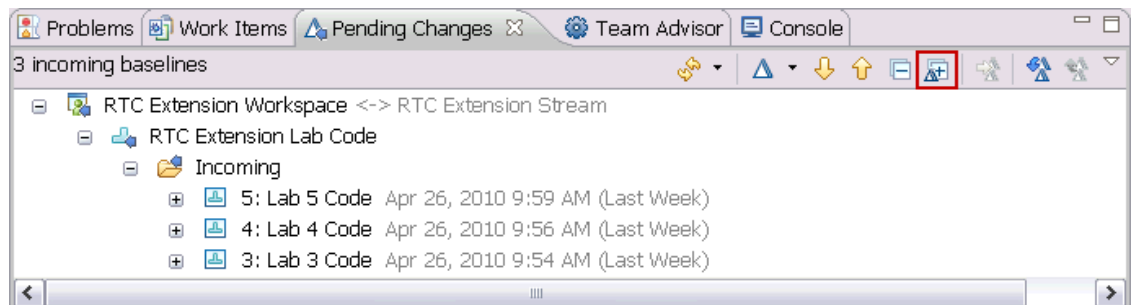
You have fulfilled the initial requirement, but you didn't really think that would be all, did you? The scrum masters like the behavior but find the messages reported on a failure confusing. You are baffled by this. They seem obvious to you, but you just roll your eyes and head back to your cube to get to work.

If your RTC server is not running, start it now

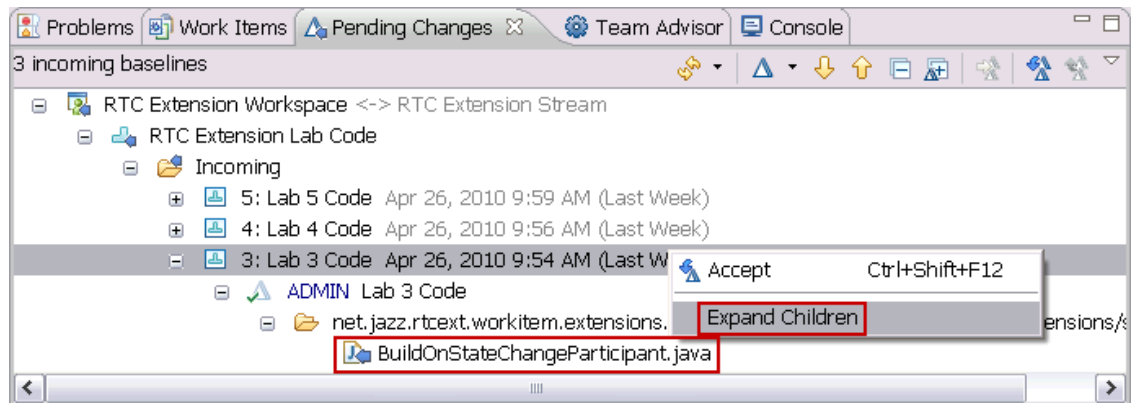
(C:\RTC2002Dev\jazz\server\server.startup.bat).

3.1 Understanding Error Handling Code

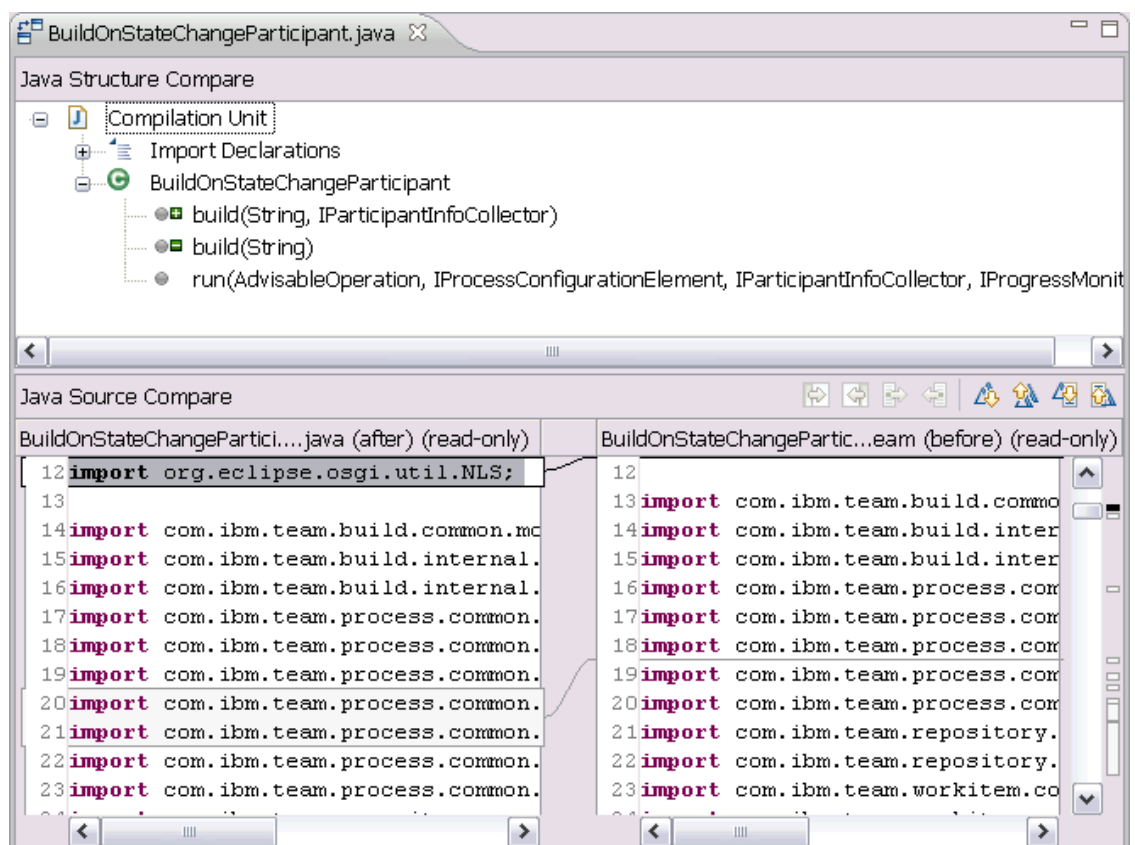
- __1. If your RTC development environment is not open, navigate to C:\RTC2002Dev\jazz\client\eclipse in the Windows explorer and double click **eclipse.exe**. If prompted to select an Eclipse workspace, select the same one you used in lab two. If the **Plug-in Development** perspective is not open, open it now by selecting **Window > Open Perspective > Other... > Plug-in Development** from the menu bar.
- __2. Browse and load the Lab 3 code.
 - __a. In the **Pending Changes** view, click the **Expand to Change sets** icon. This will show 3 incoming baselines as shown here.



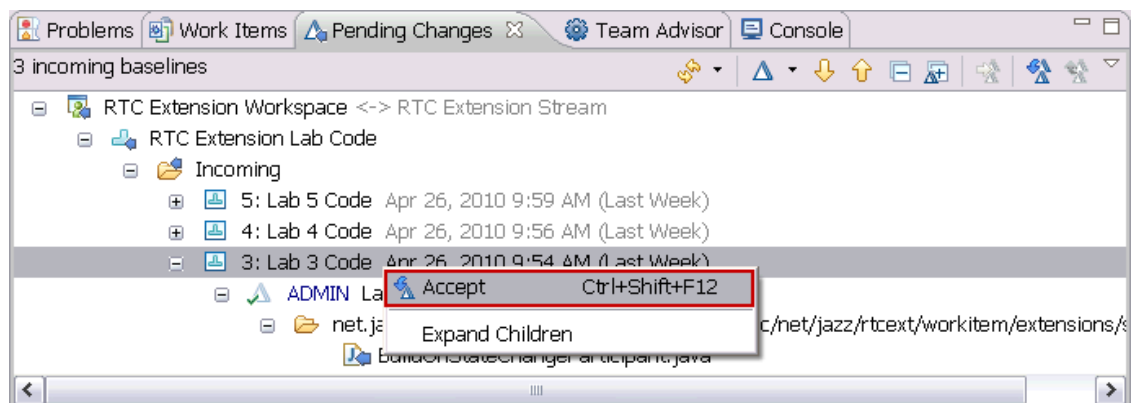
- ___b. Right click the **Lab 3 Code** change set under the **RTC Extension Workspace** node, and then click the **Expand Children** action. This will reveal all the changes made for lab 3. As you can see just the participant implementation class itself has changed.



- ___c. Double click the changed class to open a comparison editor. You may want to double click the tab of the opened editor to maximize it.



- ___d. Browse the changes and you will notice these key changes. The additional behavior will be discussed in detail after the code is loaded.
 - ___i. The collector parameter to the run method is now passed through to the build method where it will be used.
 - ___ii. The build method now checks for several error conditions in this order.
 - (1) Can the build definition be found?
 - (2) Was the build request created successfully?
 - ___iii. In all cases, even success, information is added to the collector.
- ___e. Close the comparison editor and then in the **Pending Changes** view, right click the **Lab 3 Code** baseline under the **RTC Extension Workspace** node, and then click the **Accept** action. This will accept and load the lab 3 delta on top of what you already have loaded from lab 2.



- ___3. Understand the error handling code.
 - ___a. Back in the **Package Explorer** view, expand the **src/net.jazz.rtcext.workitem.extensions.service** source package and then double click the **BuildOnStateChangeParticipant.java** file.
 - ___b. First, make sure the breakpoint at the start of the run method is still present and active. If it is not, add the breakpoint again by double clicking in the left margin next to the first line. Note that the load of the updated code may have moved the breakpoint into a comment. If that is the case, remove the breakpoint and create a new one at the start of the run method.
 - ___c. Scroll down to the build method. Note as before that the information collector is now passed to the build method.

- ___d. The first change to the body of the method is to check that the build definition was actually found.

```
/*
 * If the build definition was found, try to run the build.
 */
if (buildDef != null) {
```

If the test fails the information collector is updated in the corresponding else block as follows.

- The NLS.bind method inserts the build id into the message at the {0} insertion point. The single quotes are double so that the resulting substitution looks like 'buildId'.
- The collector.createInfo method is a simple factory method.
- The severity of ERROR is then set.
- Finally, the report info item is added to the collector. Note that this is not done automatically by the createInfo factory method.

```
/*
 * The build definition was not found, report this back as an error.
 * An error report will stop the work item save from succeeding and
 * will show up in the team advisor.
 */
String description = NLS
    .bind("The build request for build definition '{0}' could not be found.",
        buildId);
IReportInfo info = collector.createInfo("Unable to request build",
    description);
info.setSeverity(IProcessReport.ERROR);
collector.addInfo(info);
```

The second change is to check that the build request was successfully submitted.

```
/*
 * If the build request has been submitted, report success back. It
 * will show up in the team advisor if success reports are not being
 * filtered out and the show details tree is expanded.
 */
if ((response != null) && (response.getFirstClientItem() != null)) {
```


If the test fails the information collector is updated in the corresponding else block in the same manner as above. If the test passes, the information collector is also update to indicate success as follows.

- The NLS.bind method inserts the build id into the message at the {0} insertion point. The single quotes are double so that the resulting substitution looks like 'buildId'.
- The collector.createInfo method is a simple factory method.
- There is no need to set a severity since OK is the default.
- Finally, the report info item is added to the collector. Note that this is not done automatically by the createInfo factory method.

```
String description = NLS
    .bind("A new build request for build definition '{0}' was submitted.",
        buildId);
IReportInfo info = collector.createInfo("Build request successful", description);
collector.addInfo(info);
```

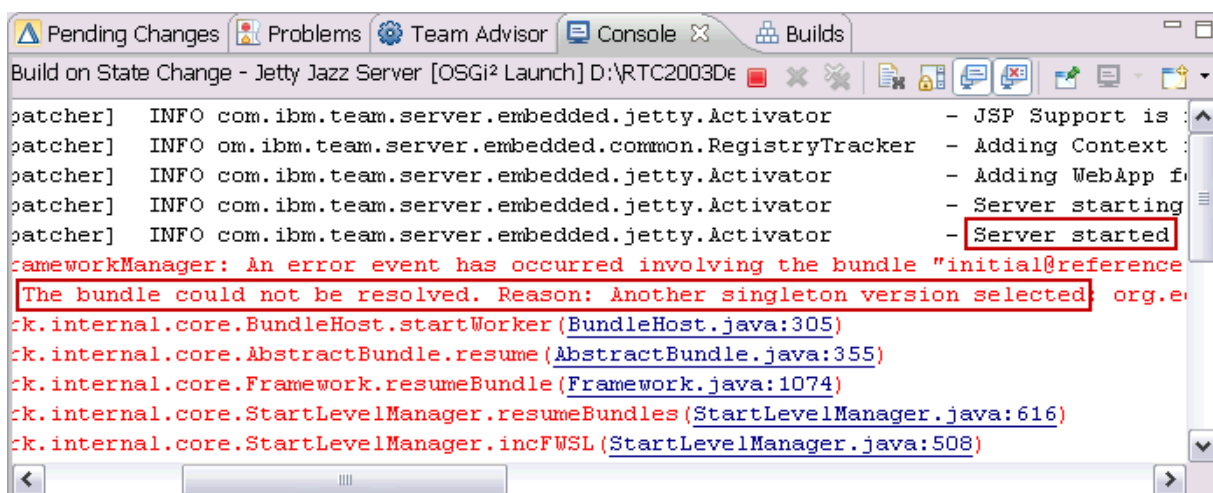
3.2 Launch the Server for Debug Using Jetty

___1. Use the existing launch configuration from lab 2.

- ___a. From the Debug toolbar dropdown () in the toolbar, select **Build on State Change - Jetty Jazz Server**.

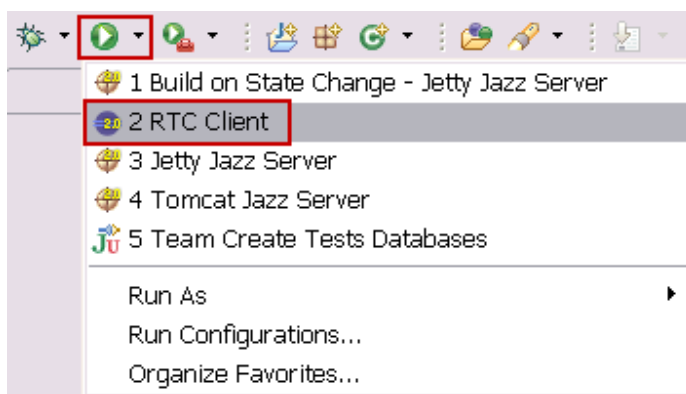
- __b. The **Console** view will take focus and log messages will start appearing there. Once you see a message that ends in **Server started**, the server is up and ready to be used.

Note: You may see an exception or two concerning the presence of two versions of an Eclipse plug-in being present. The messages contain the text “**The bundle could not be resolved. Reason: Another singleton version selected**” just before the exception’s stack trace. You can ignore these exceptions.





3.3 Launch an RTC Client and Connect to the Server

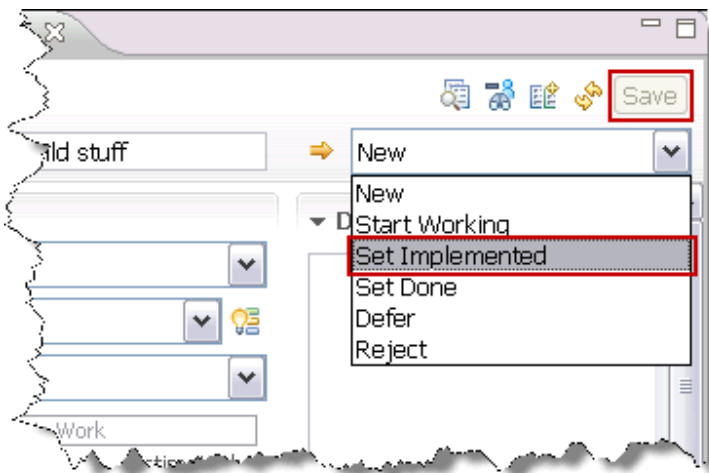
- __1. Launch the RTC Client.
- __a. From the dropdown menu of the **Run** toolbar icon, select **RTC Client**. Note that you are just running the client and not debugging. The same launch configuration can be used for both. You will debug a client in a future lab.






- __b. The RTC Eclipse client will start up and will connect automatically to the Jetty server you just launched via the repository connection you created in lab 2. The project area will still be connected and is configured for the participant since you did that in lab 2.

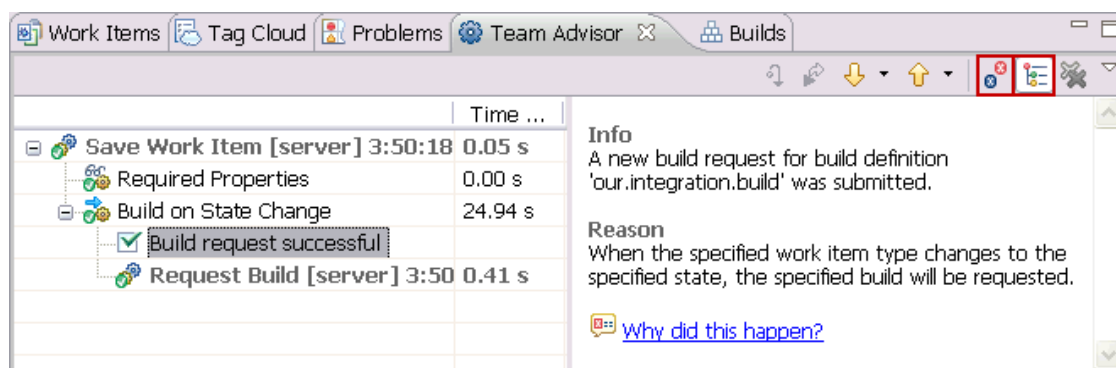
3.4 Trigger the Participant

- __1. Find the Story work item used in lab 2 and move it out of the Implemented state (via the Reopen action) or create a new story.
 - __a. Either of these will cause the breakpoint you set earlier to trigger. The RTC Eclipse client in which you were studying the code will now surface (if asked about switching to the debug perspective, click **Yes**). If it does not surface, you probably minimized it earlier. In this case, it will be flashing in the Windows taskbar. Click it in the taskbar to surface the debugger.
 - __b. If you wish, step through the run method using the **Step Over** button or F6. The check for the target state will fail and the run method will exit without requesting a build. In any case, be sure to click the resume button ().
 - __c. Switch back to the RTC Eclipse client where you created the work item. Your work item will be successfully saved. If it shows a failure due to timeout, close the editor without saving, recreate the Story (or reedit the existing Story) and when the breakpoint hits, just use the resume button ().
- __2. Move the Story to the Implemented state.
 - __a. At the upper right portion of the work item editor, select **Set Implemented** or **Complete Development** (depends on which workflow state the story is currently in) and then click **Save**.



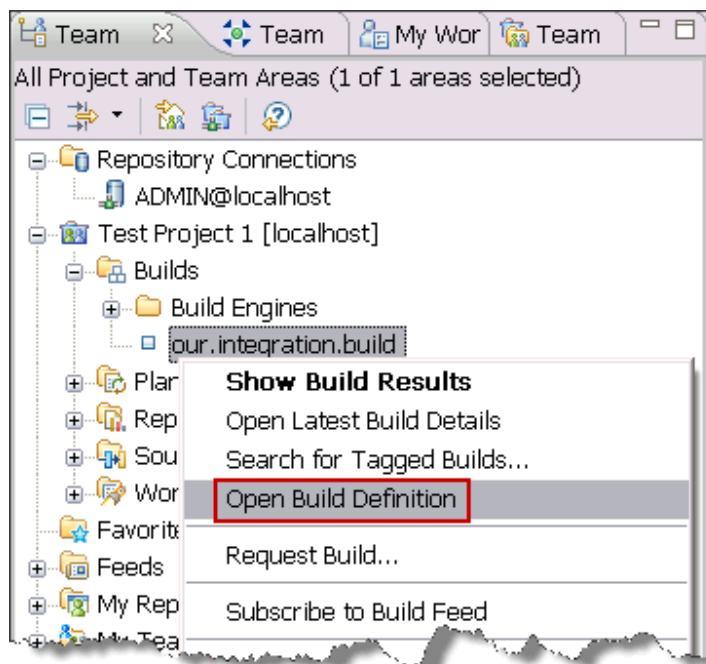
- __b. Once again the breakpoint is hit and your debugger surfaces (or you need to click it in the Windows taskbar). Step through the code again. When you get to the call to the build method, use the **Step Into** button (). You can then step through the check and status code that have been added around the same four core lines of code that request the build. Remember to click the resume button () when done stepping.

- ___c. Switch back to the RTC Eclipse client where you created the work item. Your work item will be successfully saved. If it shows a failure due to timeout, try saving again and when the breakpoint hits, just use the resume button ().
- ___d. If you go to the **Team Advisor** view and check to make sure the **Show Failures Only** filter is off and **Show Detail Tree** is on (see highlight below), you can browse the results of this successful operation. Also, if you refresh the **Builds** view, you will now see a new pending build request.

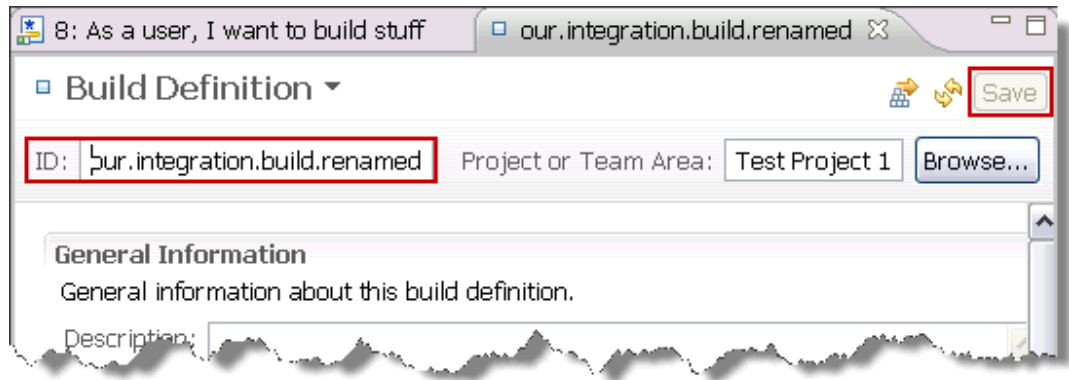


3.5 Rename Build Definition and Try Again

- ___1. Rename the build definition.
 - ___a. In the **Team Artifacts** view, right click the **our.integration.build** build definition and then click **Open Build Definition**.

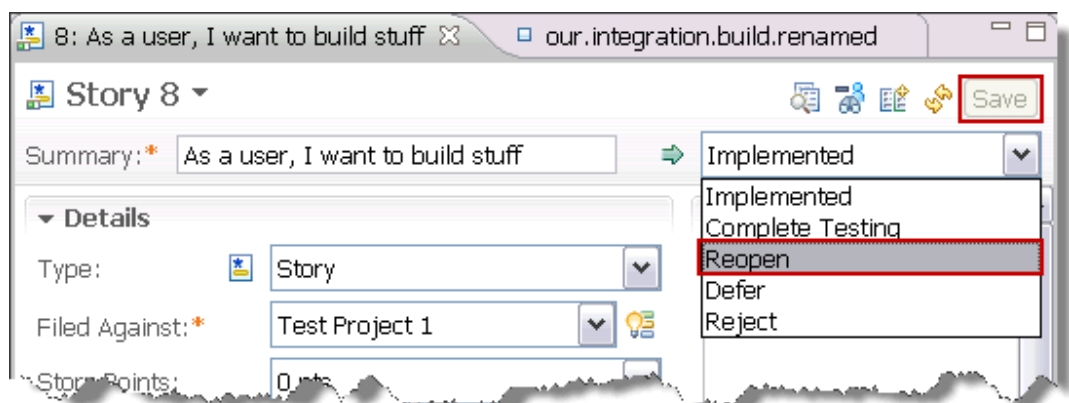


- ___b. In the build definition editor change the **ID** to `our.integration.build.renamed` and then click **Save**. Do not close the editor as you will want to rename it back soon.

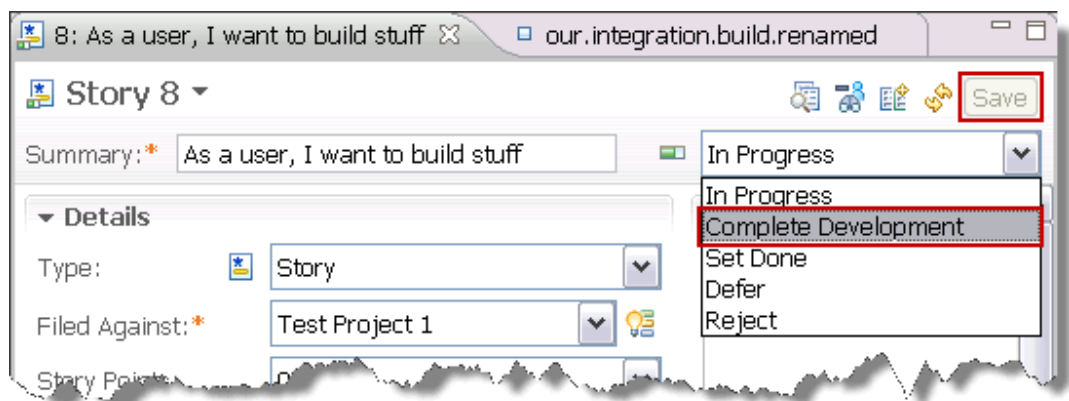


- ___2. Move the story to the Implemented state again.

- ___a. Switch back to the work item editor and select **Reopen** from the state dropdown and then click Save. When the debugger surfaces, just click the resume button (). You are not to the interesting bit yet.



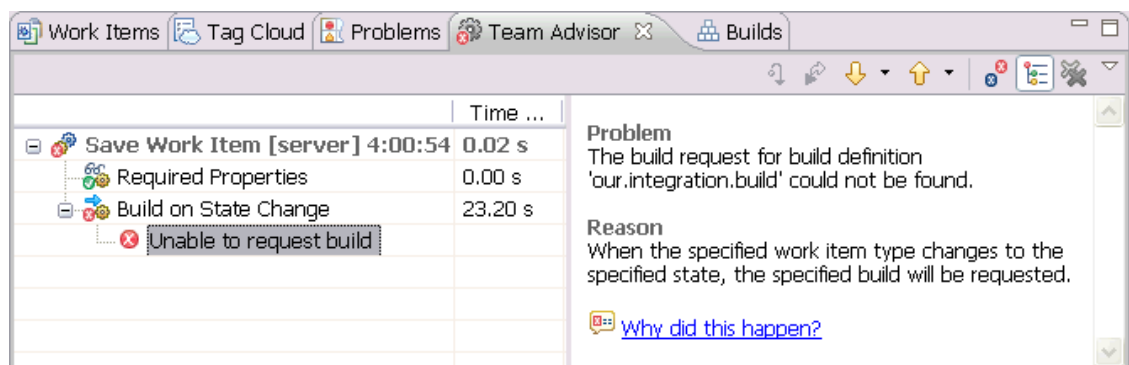
- ___b. Again in the work item editor, select **Complete Development** from the same dropdown and click **Save** again.



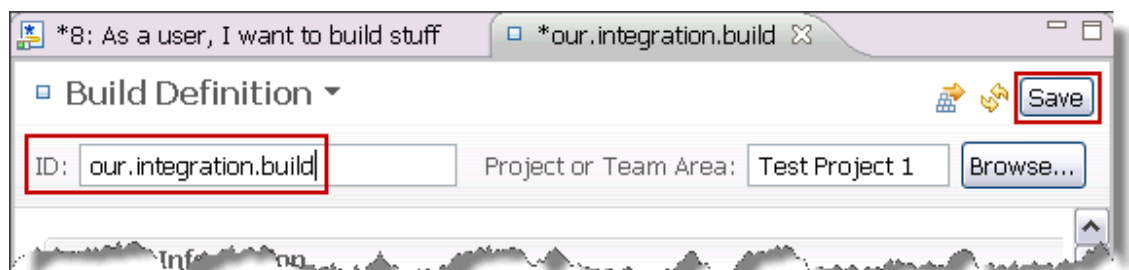
- ___c. This time, when the debugger surfaces, use the step over button to get to the build method call and then use the step into button. Step through the build method and note the major difference this time. The call to get the build definition returns null, but this time a null pointer exception is not thrown as in lab 2. This time, your new code carefully records and returns the error. Click the debugger's resume button. Then switch back to your work item editor and note the red at the top, "Unable to request build". Already you have a bit better information as to what went wrong. Click the red error text to go to the **Team Advisor** view.



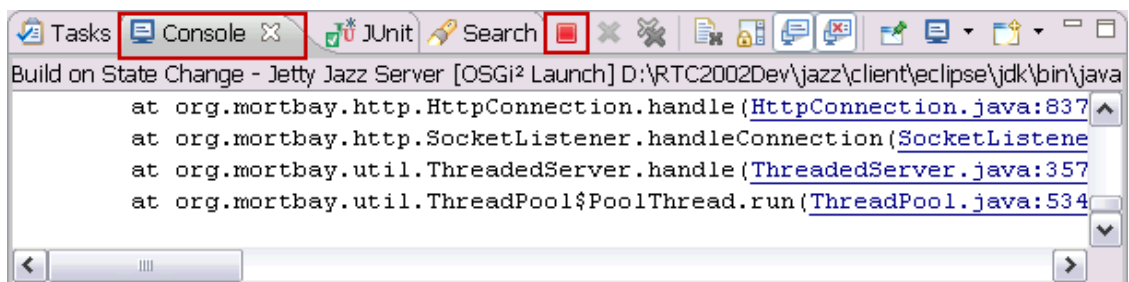
- ___d. The **Team Advisor** view appears with more information on the error. The left side of the view shows the structure of the error condition. Click the nodes on the left to see what information is available. It is clear that you now have much better information as to what went wrong.



- ___e. Switch back to the build definition editor and change the **ID** back to `our.integration.build` and click **Save**.



- ___f. Switch back to the work item editor and click **Save**. When the debugger surfaces, you can step into the build method again or just hit resume. Once you do resume, the work item save should complete okay. Return to the work item editor to confirm this. If you go to the **Team Advisor** view and the **Show Failures Only** filter is off, you can browse the results of this successful operation. Also, if you refresh the **Builds** view, you will see another new pending build request.
- ___3. Close down the launched client and server.
 - ___a. Close the RTC Eclipse client where you were working with the Story and build definition.
 - ___b. Back in the original RTC Eclipse client, go to the **Console** view and click the **Terminate** icon.



You have completed lab 3. Your initial server side operation participant fails in a much friendlier manner. In future labs, you will make the work item type, state and build id configurable.

Lab 4 Parameterization



Lab Scenario

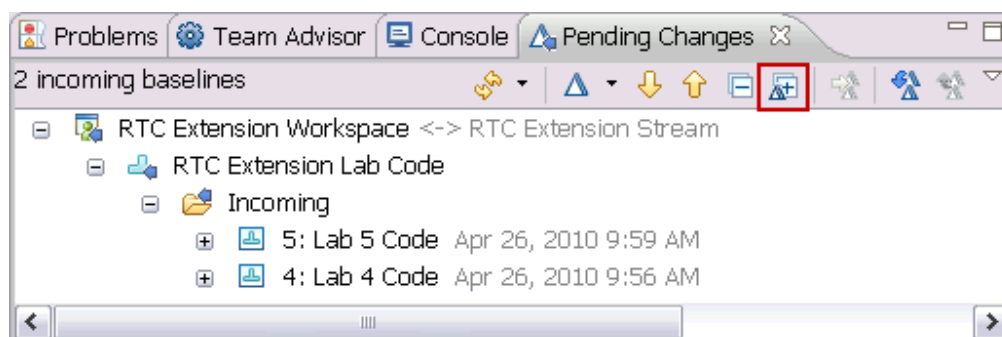
The error and success message are sweet! However, your scrum masters clients have more ideas. Now they want to be able to configure the work item type and state to trigger a build. They also want to be able to specify which build to run. You think they could of mentioned that the first time!

If your RTC server is not running, start it now

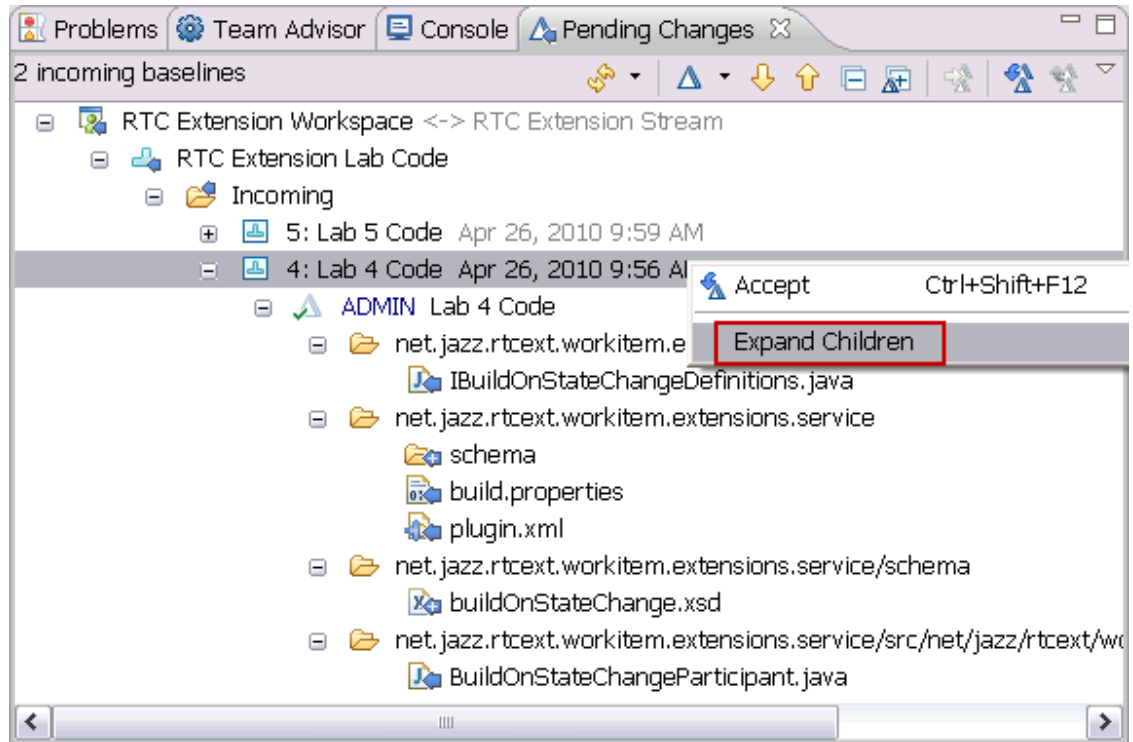
(C:\RTC2002Dev\jazz\server\server.startup.bat).

4.1 Understanding Parameterization

- __1. If your RTC development environment is not open, navigate to C:\RTC2002Dev\jazz\client\eclipse in the Windows explorer and double click **eclipse.exe**. If prompted to select an Eclipse workspace, select the same one you used in lab two. If the **Plug-in Development** perspective is not open, open it now by selecting **Window > Open Perspective > Other... > Plug-in Development** from the menu bar.
- __2. Browse and load the Lab 4 code.
 - __a. In the **Pending Changes** view, click the **Expand to Change sets** icon. This will show 2 incoming baselines as shown here.

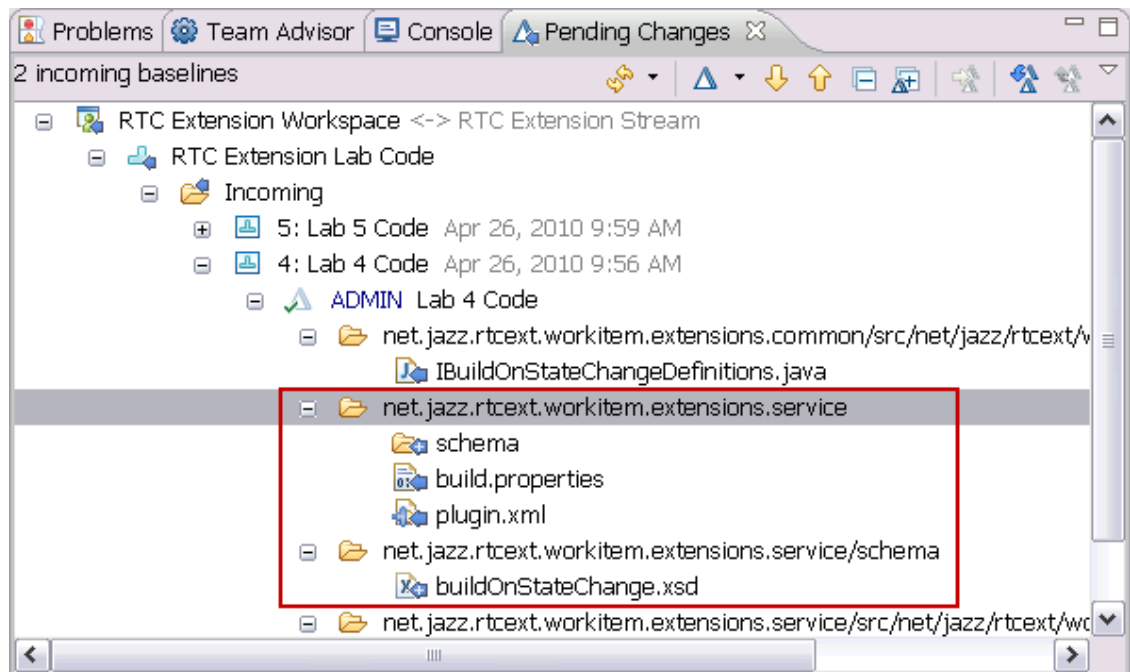


- ___b. Right click the **Lab 4 Code** change set under the **RTC Extension Workspace** node, and then click the **Expand Children** action. This will reveal all the changes made for lab 4. As you can see there are quite a few more changes in this lab.

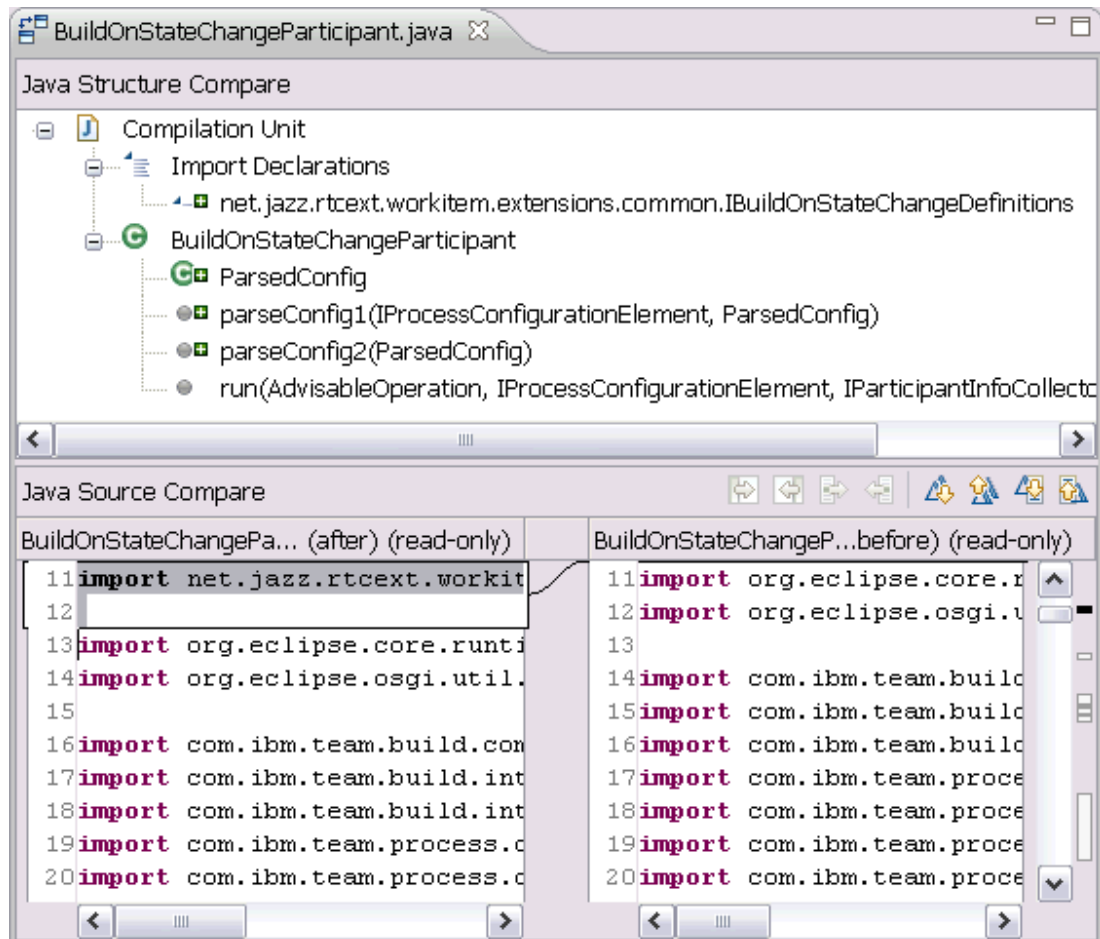


- ___c. Double click the first changed file, **IBuildOnStateChangeDefinitions.java** to open a comparison editor. You may want to double click the tab of the opened editor to maximize it. A set of constants have been added to this file. Most of them define elements of the XML schema that will be used to configure your follow up action. You will look a little closer at this soon. Close the comparison editor.

- ___d. The next four changes all have to do with adding the schema definition. The first adds the schema folder to the service plug-in. The second adds that folder and its contents to the plug-in's build properties. The third, adds the schema to the participant's extension point definition from lab 2. The fourth change adds the schema file itself. You will look at the schema file in some detail later in this lab.

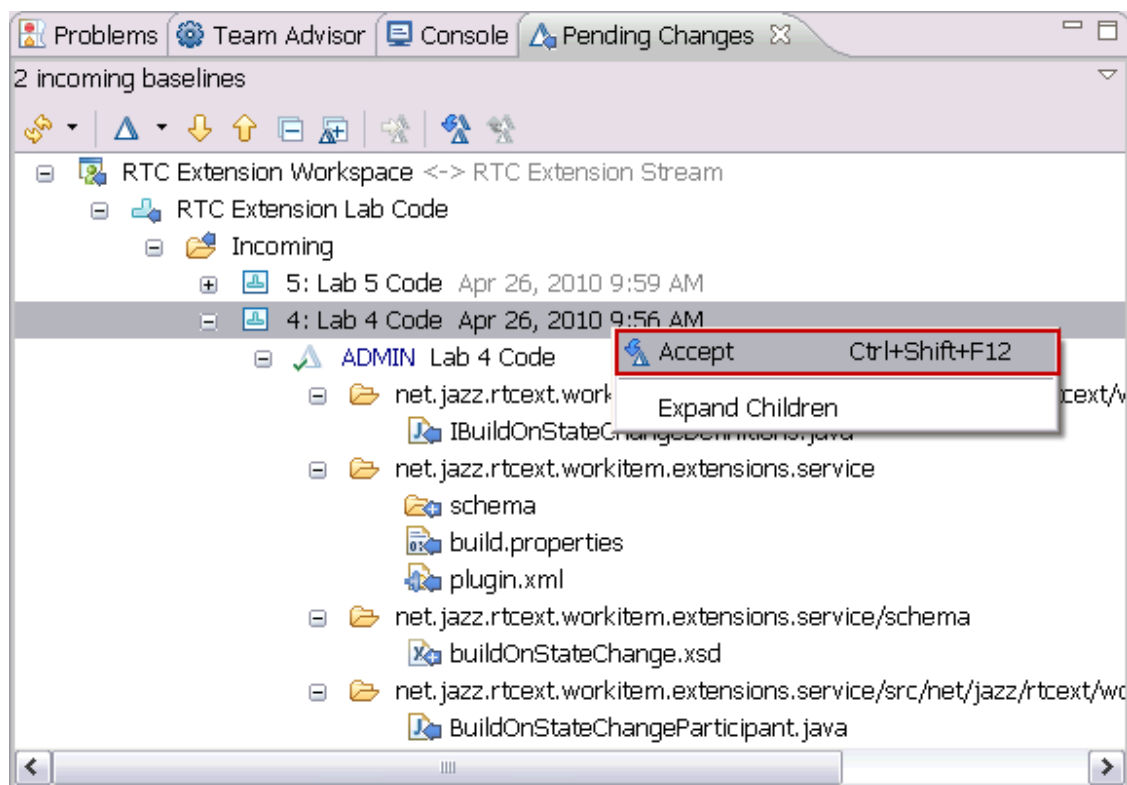


- ___e. The final change is once again to the participant implementation itself. Double click the **BuildOnStateChangeParticipant.java** file to open a comparison editor. You may want to double click the editor's tab to maximize it.



- ___f. Browse the changes and you will notice these key changes. The additional behavior will be discussed in detail after the code is loaded.
- ___i. A new nested class has been added, `ParsedConfig`. It is a simple structure used to pass configuration results between the parsing stages. Remember, no instance state variables in an operation participant!
 - ___ii. Two new parse methods have been added. They perform a two stage parse on the configuration. Note that doing a two stage parse in this case is not really needed since the second stage has no real performance implications, but the pattern will be explained later when you look at the code in detail.
 - ___iii. The `run` method no longer uses hard coded ids.

- ___g. Close the comparison editor and then in the **Pending Changes** view, right click the **Lab 4 Code** baseline under the **RTC Extension Workspace** node, and then click the **Accept** action. This will accept and load the lab 4 delta on top of what you already have loaded from lab 3.



___3. Understanding the schema.

- ___a. Back in the **Package Explorer** view, expand the **src/net.jazz.rtext.workitem.extensions.common** source package and then double click the **IBuildOnStateChangeDefinitions.java** file.
- ___i. The critical additions to this file are the comments that describe the syntax for the participant's configuration XML and the constant definitions that go with them. Snippets of XML that follow this syntax will be added to the process configuration of a project or team area using the follow up action.
- ___ii. The first comment and set of constants defines what how the triggering work item type and state are configured.

```
// <trigger>
// <changed-workitem-type id="workitem.type.id"/>
// <trigger-state id="trigger.state.id"/>
// </trigger>
public static final String ID = "id";
public static final String TAG_TRIGGER = "trigger";
public static final String TAG_CHANGED_WORKITEM_TYPE = "changed-workitem-type";
public static final String TAG_TRIGGER_STATE_ID = "trigger-state";
```

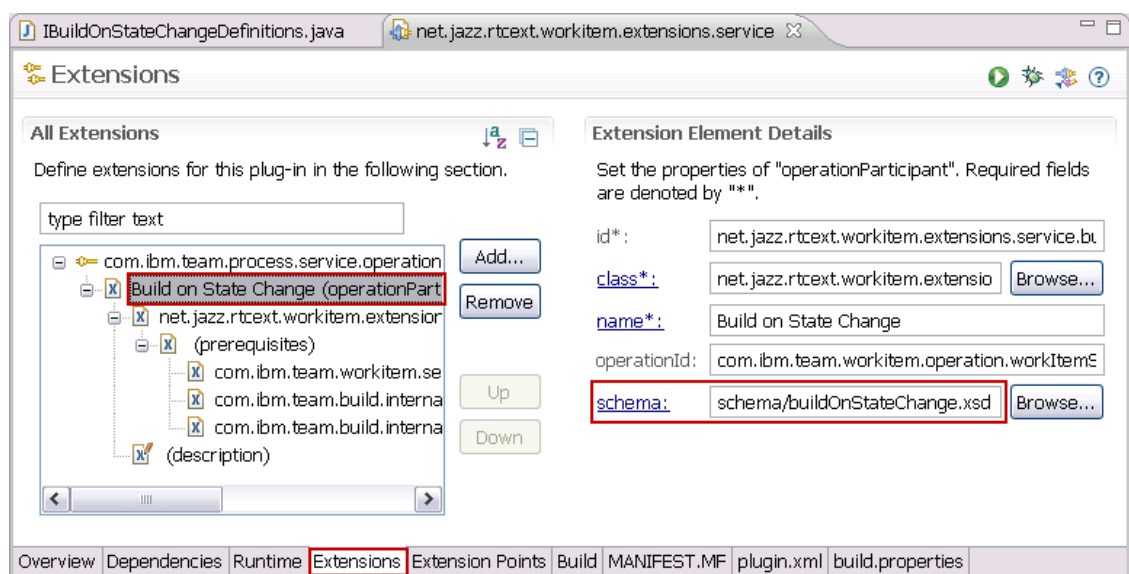
- ___iii. The second comment and set of constants defines what how the target build is configured.

```
// <build>
// <build-definition id="build.definition.id"/>
// </build>
public static final String TAG_BUILD = "build";
public static final String TAG_BUILD_DEFINITION = "build-definition";
```

- ___iv. You may want to keep this file open to reference the syntax comments as you examine the other files.

- ___b. Back in the **Package Explorer** view, expand the first level of the **net.jazz.rtcext.workitem.extensions.service** plug-in project and then double click the **plugin.xml** file.

- ___i. Click on the **Extensions** tab and expand the nodes under the participant on the left. Note the schema field on the right. Adding this reference to the schema file is the only change to the plugin.xml file for lab 4. You can close the plugin.xml file editor.



- ___c. Back in the **Package Explorer** view, expand the first level of the **schema** folder inside the **net.jazz.rtcext.workitem.extensions.service** plug-in project and then double click the **buildOnStateChange.xsd** file. What editor opens depends on which Eclipse plug-ins you have installed. If you are just using RTC, you will get a text editor. If you have Rational Application Developer (RAD) or the Eclipse Web Tools Platform (WTP) installed along with RTC, you will get a much richer XML schema editor. In either editor, you will see the definition of one element and three types.

- __i. The element definition and first type definition define how these schema elements fit into the overall process definition schema. The first documentation element explains how the element at the top of this section and the base attribute of this type establish where this schema extends the base process definition schema. Note that the process schema is imported and given the XML namespace prefix “process” in earlier elements. Also, as the documentation points out, the required and fixed valued id attribute establishes linkage to your participant. Finally, note that the two nested elements are both required and can occur only once. These are the “trigger” and “build” elements. The details of the structure of these elements are defined in the following type definitions.

```
<xsd:element name="followup-action" substitutionGroup="process:followup-action"
             type="buildOnStateChangeType" />

<xsd:complexType name="buildOnStateChangeType">
  <xsd:annotation>
    <xsd:documentation>
      This type defines the build on state change type. It is a
      subtype of the abstract process:followupActionType. This
      restriction, along with the substitutionGroup specification
      above, makes it possible to add configuration of the participant
      to a project or team area's process configuration. Note the
      forward references to the trigger and build types defined below.
      Take particular note of the id attribute. It is required and has
      a fixed value that points to our operation participant extension.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:restriction base="process:followupActionType">
      <xsd:all>
        <xsd:element name="trigger" type="triggerType"
                     minOccurs="1" maxOccurs="1" />
        <xsd:element name="build" type="buildType" minOccurs="1"
                     maxOccurs="1" />
      </xsd:all>
      <xsd:attribute name="id" type="xsd:string" use="required"
                     fixed="net.jazz.rtcext.workitem.extensions.service.buildOnStateChange" />
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

- __ii. The second type definition defines the trigger type. It may be helpful to refer to the simple syntax diagram in the IBuildOnStateChangeDefinitions.java file as you look at this type definition. There are also two nested elements defined for this type that are also required and can only occur once. They will contain the work item type and state ids (“changed-workitem-type” and “trigger-state”).

```
<xsd:complexType name="triggerType">
  <xsd:annotation>
    <xsd:documentation>
      This type defines the work item type to be monitored
      and the work item state that should trigger the
      operation participant.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:all>
    <xsd:element name="changed-workitem-type" minOccurs="1"
      maxOccurs="1">
      <xsd:complexType>
        <xsd:attribute name="id" type="xsd:string"
          use="required"/>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="trigger-state" minOccurs="1" maxOccurs="1">
      <xsd:complexType>
        <xsd:attribute name="id" type="xsd:string"
          use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:all>
</xsd:complexType>
```

- __iii. The third type definition defines the target build type. It may be helpful to refer to the simple syntax diagram in the IBuildOnStateChangeDefinitions.java file as you look at this type definition. There is one nested element defined for this type that is also required and can only occur once. It will contain the build definition id (“build-definition”).

```
<xsd:complexType name="buildType">
  <xsd:annotation>
    <xsd:documentation>
      This type defines the build to run. At this point, it just
      includes the build definition id. In the future, it could
      include more information, for example, a list of properties
      to pass to the build.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:all>
    <xsd:element name="build-definition" minOccurs="1" maxOccurs="1">
      <xsd:complexType>
        <xsd:attribute name="id" type="xsd:string"
          use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:all>
</xsd:complexType>
```

- ___4. Understanding the build on state change participant code changes.
- ___a. Back in the **Package Explorer** view, expand the **src/net.jazz.rtcontext.workitem.extensions.service** source package and then double click the **BuildOnStateChangeParticipant.java** file.
 - ___b. First, make sure the breakpoint at the start of the run method is still present and active. If it is not, add the breakpoint again by double clicking in the left margin next to the first line.
 - ___c. The changes in this class are all about using the configured ids as opposed to the hard coded ids. There is a two stage parse used. As noted in the large comment block in the run method that starts with “Perform the first stage of configuration parsing”, a single stage would be fine in this case since none of the parsing has important performance considerations. However, the pattern can be useful in some common scenarios and needs to be illustrated.
 - ___d. Just below that comment in the run method, note how the work item type id is now used from the configuration. You will look at the two new parsing methods soon.

```
ParsedConfig parsedConfig = new ParsedConfig();
parseConfig1(participantConfig, parsedConfig);
String newType = newState.getWorkItemType();

/*
 * If the work item is not of the proper type, do not build. If
 * the work item type id is null, the test will return false and
 * a build will not be attempted.
 */
if (newType.equals(parsedConfig.fWorkItemId)) {
```

- ___e. Just after that, note how the work item state id is also used from the configuration.

```
/*
 * Finally, if the new state is the target state, build.
 * Again, a null id is handled in the same manner.
 */
if (newState.getState2().getStringIdentifier().equals(
    parsedConfig.fWorkItemStateId)) {
```

- ___f. Finally, in the run method, note that only if it is known that a build is needed then the second stage of the parse is performed and the build is requested using the build definition id from the configuration and that a null id means no build to run. Also note that the build method has not changed at all.

```

/*
 * Now it is time for the second stage of the
 * configuration parse. Only build if the build
 * definition id is not null.
 */
parseConfig2(parsedConfig);
if (parsedConfig.fBuildDefinitionId != null)
    build(parsedConfig.fBuildDefinitionId, collector);

```

- ___g. The other major change to this class, of course, is the addition of the two parsing methods and the structure used with them to pass the intermediate (after parse 1 but before parse 2) and final parsing results around the participant. The structure is very simple as shown here. The first three fields are filled in by parse 1. Parse 2 uses the cached third field to fill in the final field. Recall that there are two stages since you are pretending that retrieving and/or calculating the build definition id is expensive and it should only be done if required. This is not really true, but illustrates a useful pattern.

```

/**
 * This class is used retrieve results from the participant
 * configuration parsing methods.
 */
private class ParsedConfig {
    public String fWorkItemTypeId = null;
    public String fWorkItemStateId = null;
    public IProcessConfigurationElement fBuildConfigElement = null;
    public String fBuildDefinitionId = null;
}

```

- ___h. The first parse method looks more complicated than it is. The first thing to know is that the participantConfig parameter passed in via the run method is as described as follows in the run method comment. The required single occurrence “trigger” and “build” elements are children of this element.

```

* @param participantConfig
*     the configuration element which configures this participant;
*     this corresponds to the XML element which declares this
*     participant in the process specification/customization.
*     <p>
*     This participant obtains the trigger work item type and state
*     from this parameter. The build definition id is also found
*     here.

```


- ___i. The code in the first parse method loops through the children of the parent configuration element and looks for the “trigger” and “build” elements. When it finds the “trigger” element it parses deeper to get the work item type and state ids. When it finds the “build” element, it simply caches the element in the proper field of the parseConfig parameter for use by the second parse method. As can be seen here, the deeper parse of the “trigger” element follows the same loop and examine pattern on the children of the “trigger” element.

```

if (element.getName().equals(
    IBuildOnStateChangeDefinitions.TAG_TRIGGER)) {
    /*
     * Found a trigger definition. Cycle through its child elements
     * to find the work item and state ids.
     */
    IProcessConfigurationElement[] children = element.getChildren();
    for (int i = 0; i < children.length; i++) {
        IProcessConfigurationElement child = children[i];
        String elementName = child.getName();
        if (elementName
            .equals(IBuildOnStateChangeDefinitions.TAG_CHANGED_WORKITEM_TYPE)) {
            parsedConfig.fWorkItemTypeId = child
                .getAttribute(IBuildOnStateChangeDefinitions.ID);
        } else if (elementName
            .equals(IBuildOnStateChangeDefinitions.TAG_TRIGGER_STATE_ID)) {
            parsedConfig.fWorkItemStateId = child
                .getAttribute(IBuildOnStateChangeDefinitions.ID);
        }
    }
} else if (element.getName().equals(IBuildOnStateChangeDefinitions.TAG_BUILD)) {
    /*
     * Found the build definition. For now, just set aside the
     * element. It will only be parsed if we need to run a build.
     */
    parsedConfig.fBuildConfigElement = element;
}


```

- __j. The second parse method is uses a similar pattern but is a bit simpler since it has less to parse and the “build” element has already been cached Note the check for null at the start of the method to make sure the “build” element really was found by the first parse method.

```
/**
 * Second stage of the configuration parsing that handles the build
 * definition.
 *
 * @param parsedConfig
 *      the build definition element is now parsed and the build
 *      definition id is updated. Note that the id is not validated by
 *      this method and may still be null.
 */
private void parseConfig2(ParsedConfig parsedConfig) {
    if (parsedConfig.fBuildConfigElement != null) {
        IProcessConfigurationElement[] children =
            parsedConfig.fBuildConfigElement.getChildren();
        for (int i = 0; i < children.length; i++) {
            IProcessConfigurationElement child = children[i];
            String elementName = child.getName();
            if (elementName
                .equals(IBuildOnStateChangeDefinitions.TAG_BUILD_DEFINITION)) {
                parsedConfig.fBuildDefinitionId = child
                    .getAttribute(IBuildOnStateChangeDefinitions.ID);
            }
        }
    }
}
```

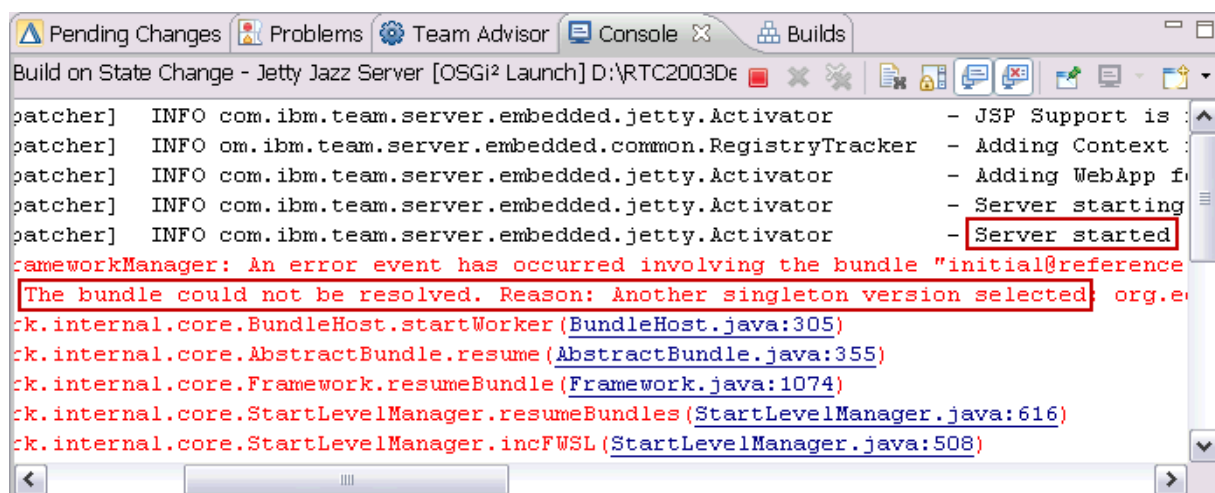
- __k. You can now close all your open editors and proceed to the next section to configure and again step through the configured follow-up action.

4.2 Launch the Server for Debug Using Jetty

- __1. Use the existing launch configuration from lab 2.
- __a. From the Debug toolbar dropdown () in the toolbar, select **Build on State Change - Jetty Jazz Server**.

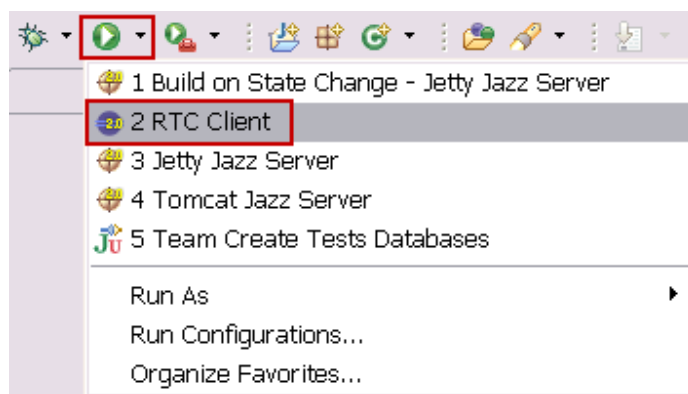
- __b. The **Console** view will take focus and log messages will start appearing there. Once you see a message that ends in **Server started**, the server is up and ready to be used.

Note: You may see an exception or two concerning the presence of two versions of an Eclipse plug-in being present. The messages contain the text “**The bundle could not be resolved. Reason: Another singleton version selected**” just before the exception’s stack trace. You can ignore these exceptions.



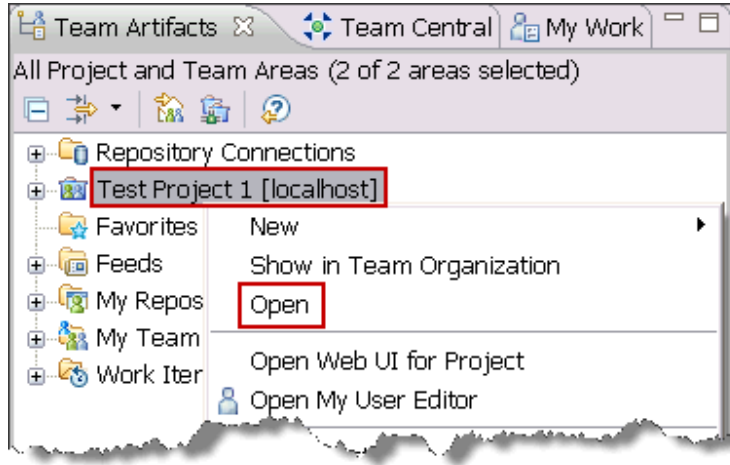
4.3 Launch an RTC Client and Configure the Participant

- __1. Launch the RTC Client.
- __a. From the dropdown menu of the **Run** toolbar icon, select **RTC Client**. Note that you are just running the client and not debugging. The same launch configuration can be used for both. You will debug a client in a future lab.

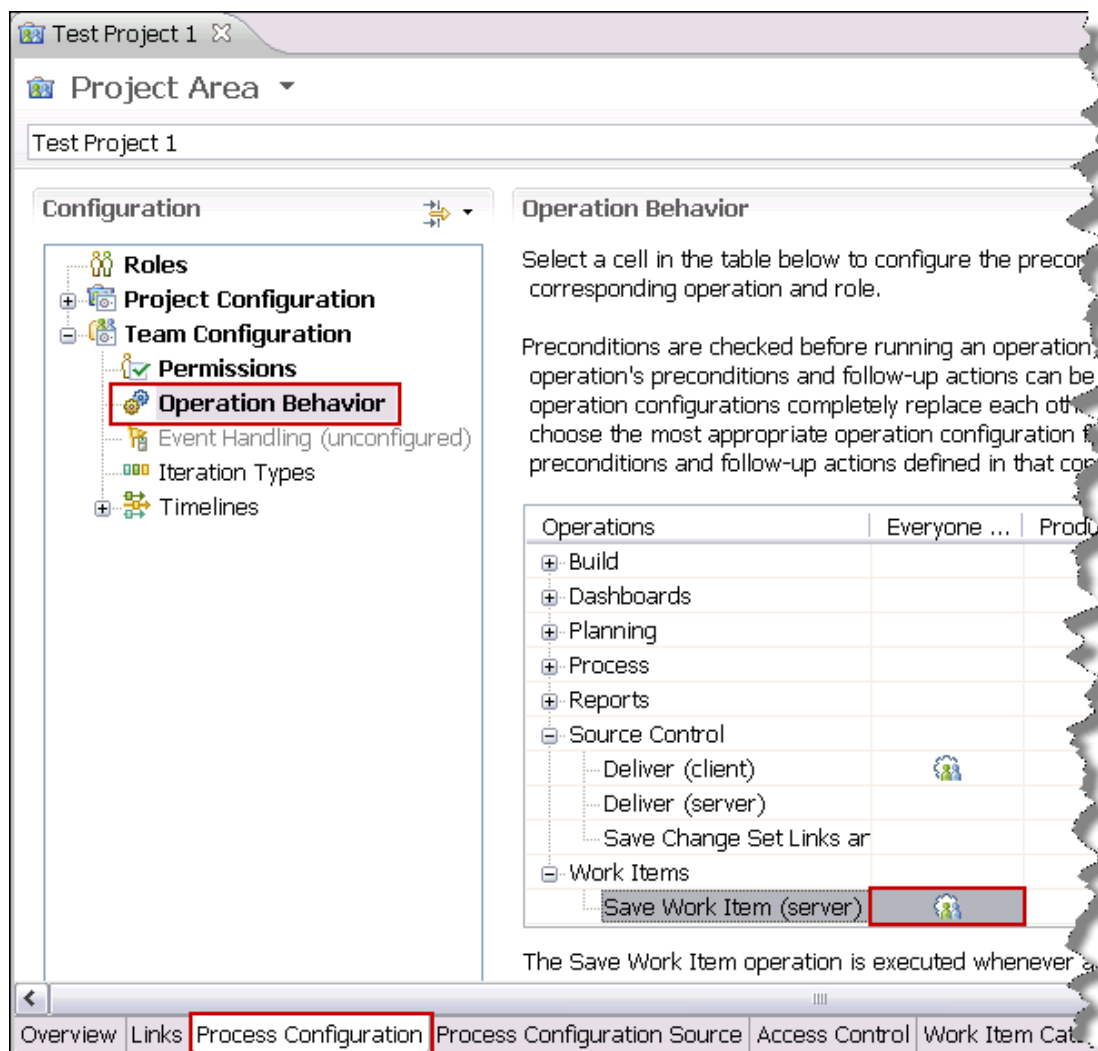


- __b. The RTC Eclipse client will start up and will connect automatically to the Jetty server you just launched via the repository connection you created in lab 2. The project area will still be connected; however, you do have some more work to do this time. The participant is still added as a follow-up action on work item save, but it has not been configured with the required work item type, state and build definition ids. You need to fix this.

- ___2. There are two steps required to fix the build on state change participant that is currently configured for your test project. In this first step, you will make sure the XML generated from adding the participant is associated with the schema you just added.
- ___a. In the **Team Artifacts** view, right click the **Test Project 1** project area and then click the **Open** action in the menu.

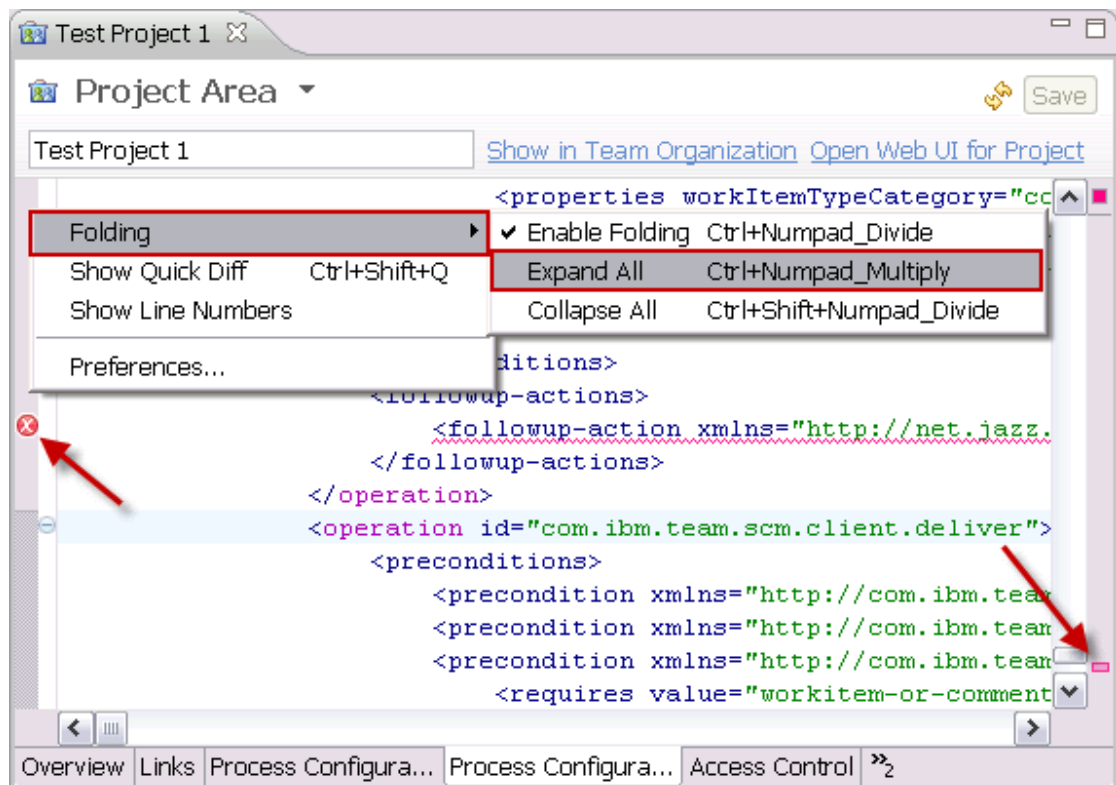


- ___b. In the project editor that opens, switch to the **Process Configuration** tab and then on the left, expand the **Team Configuration** tree then select **Operation Behavior**. Then, on the right, scroll down to the **Work Items > Save Work Item (server)** operation and select the **Everyone (default)** column next to it as shown here.



- ___c. Scroll down to find the **Follow-up actions** section on the right, remove the **Build on State Change** participant that is already in the list and then add it back in again. This may seem unusual, but there is a good reason for it. If you looked at the XML for the participant before and after doing this, you will notice one key difference, that is, the addition of an xmlns attribute that references the schema. The XML validator for the process configuration uses this information to produce the proper error messages for incorrect or incomplete (your case here) process configuration elements.
- ___d. Press **Save** in the upper right corner of the editor to save this change.

- ___3. In this second fix up step, you will actually configure the required work item type, state and build definition ids.
- ___a. Switch to the **Process Configuration Source** tab. Right click in the left margin and from the menu, select **Folding > Expand All**. You will then see in the right margin and small red rectangle indicating an error. Left click the small red rectangle and the editor will scroll to the line with an error. The error will be further indicated by a red circle with an X in the left margin and a red squiggly underline.



- ___b. Hover your mouse over the red circle with the X in the left margin and you will see the following message describing the error. Because you have created a schema and linked it to your participant extension point, the process editor is aware that the configuration of the follow-up action is not complete.

cvc-complex-type.2.4.b: The content of element 'followup-action' is not complete. One of '{ "http://net.jazz.rtext.workitem.extensions.service/server/buildOnStateChange": trigger, "http://net.jazz.rtext.workitem.extensions.service/server/buildOnStateChange": build }' is expected.

- ___c. Since you do not yet have an editor for your XML aspect (next lab), you will need to edit the XML by hand. Here is what the `followup-action` element and its children should end up looking. You do not need to type all of this or rely on your typing skills to get the syntax just right. You can use Ctrl+Space to use context sensitive code assist. Do note the values of the ids. They are the same as the ones that use to be hard coded in the participant.

```
<followup-action
  xmlns="http://net.jazz.rtcext.workitem.extensions.service/server/buildOnStateChange"
  description="When the specified work item type changes to the specified state, the
    specified build will be requested."
  id="net.jazz.rtcext.workitem.extensions.service.buildOnStateChange"
  name="Build on State Change">
  <trigger>
    <changed-workitem-type id="com.ibm.team.apr.workItemType.story"/>
    <trigger-state id="com.ibm.team.apr.story.tested"/>
  </trigger>
  <build>
    <build-definition id="our.integration.build"/>
  </build>
</followup-action>
```

First, change the existing `followup-action` element to have an explicit end tag. That is, change the `/>` at the end of the existing tag to just `>` and then add a `</followup-action>` end tag on a new line after the existing tag. Also leave a blank line between the two. It will now look like this.

```
<followup-action
  xmlns="http://net.jazz.rtcext.workitem.extensions.service/server/buildOnStateChange"
  description="When the specified work item type changes to the specified state, the
    specified build will be requested."
  id="net.jazz.rtcext.workitem.extensions.service.buildOnStateChange"
  name="Build on State Change">
</followup-action>
```

On the blank line, after indenting a tab if you wish, hit Ctrl+Space and you will see a list of valid elements to place at this point. Choose “trigger” from the list. It will now look like this.

```
<followup-action
  xmlns="http://net.jazz.rtcext.workitem.extensions.service/server/buildOnStateChange"
  description="When the specified work item type changes to the specified state, the
    specified build will be requested."
  id="net.jazz.rtcext.workitem.extensions.service.buildOnStateChange"
  name="Build on State Change">
  <trigger></trigger>
</followup-action>
```

Add another blank line after the line you just added, use Ctrl+Space again and this time select “build” from the list. It will now look like this.

```
<followup-action
  xmlns="http://net.jazz.rtcext.workitem.extensions.service/server/buildOnStateChange"
  description="When the specified work item type changes to the specified state, the
    specified build will be requested."
  id="net.jazz.rtcext.workitem.extensions.service.buildOnStateChange"
  name="Build on State Change">
  <trigger></trigger>
  <build></build>
</followup-action>
```

Place your cursor between the “trigger” start and end tags and use Ctrl+Space again (you may first want to hit enter a couple times first to add a blank line between them and perhaps add some tabs to make it look better). Select “changed-workitem-type” from the list. You will need to add the id value of com.ibm.team.apt.workItemType.story. It will now look like this.





```
<followup-action
  xmlns="http://net.jazz.rtcext.workitem.extensions.service/server/buildOnStateChange"
  description="When the specified work item type changes to the specified state, the
    specified build will be requested."
  id="net.jazz.rtcext.workitem.extensions.service.buildOnStateChange"
  name="Build on State Change">
  <trigger>
    <changed-workitem-type id="com.ibm.team.apt.workItemType.story" />
  </trigger>
  <build></build>
</followup-action>
```

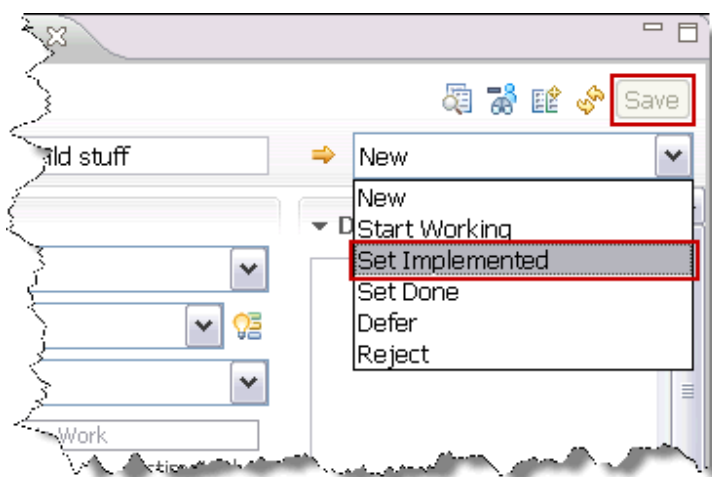
Continue in the same manner to add the “trigger-state” element inside the “trigger” and the “build-definition” element inside the “build” until it looks like the finished product noted previously.




- ___d. Click **Save** at the top right of the project area editor. Your follow-up action is now properly configured. **Leave the editor open at this point.** You will soon come back here and make a small change.

4.4 Trigger the Participant

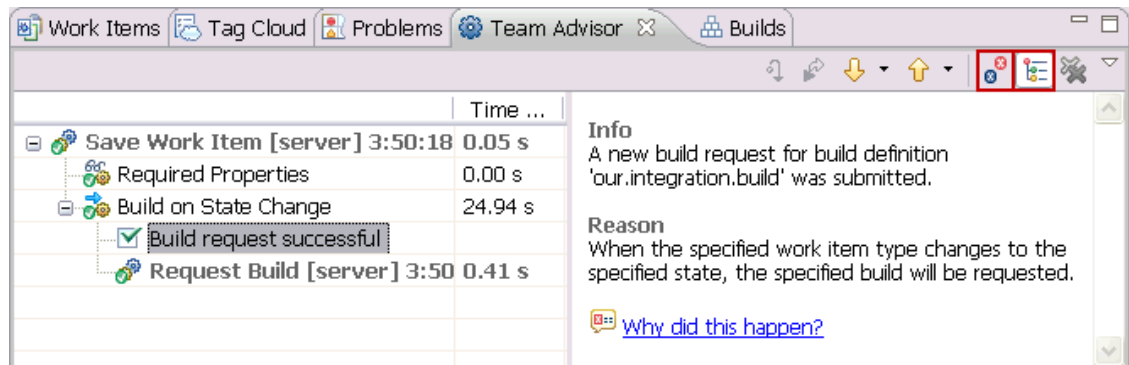
- ___1. Find the Story work item used in lab 2 and 3 and then move it out of the Implemented state (via the Reopen action) or create a new story.
 - ___a. Either of these will cause the breakpoint you set earlier to trigger. The RTC Eclipse client in which you were studying the code will now surface (if asked about switching to the debug perspective, click **Yes**). If it does not surface, you probably minimized it earlier. In this case, it will be flashing in the Windows taskbar. Click it in the taskbar to surface the debugger.

- ___b. Step through the run method using the **Step Over** button () or F6. When you get to the `configParse1` method call, click the **Step Into** button () or F5 in order to step through the first stage of the parse. Eventually, the check for the target state will fail and the run method will exit without requesting a build. In any case, be sure to click the resume button ().
 - ___c. Switch back to the RTC Eclipse client where you created the work item. Your work item will be successfully saved. If it shows a failure due to timeout, close the editor without saving, recreate the Story (or reedit the existing Story) and when the breakpoint hits, just use the resume button ().
- ___2. Move the Story to the Implemented state.
- ___a. At the upper right portion of the work item editor, select **Set Implemented** or **Complete Development** (depends on which workflow state the story is currently in) and then click **Save**.



- ___b. Once again the breakpoint is hit and your debugger surfaces (or you need to click it in the Windows taskbar). Step through the code again. If you wish, you can step into the `parseConfig1` method but it will do exactly the same thing it did last time. As you step through the run method, the state check will pass this time and a build will be run. When you get to the call to the `parseConfig2` method, use the **Step Into** button (). You can then step through this method for the first time. When you get to the call to the build method, you can step in or not. It has not changed in this lab. Remember to click the resume button () when done stepping.
- ___c. Switch back to the RTC Eclipse client where you created the work item. Your work item will be successfully saved. If it shows a failure due to timeout, try saving again and when the breakpoint hits, just use the resume button ().

- ___d. If you go to the **Team Advisor** view and check to make sure the **Show Failures Only** filter is off and **Show Detail Tree** is on (see highlight below), you can browse the results of this successful operation. Also, if you refresh the **Builds** view, you will now see a new pending build request.



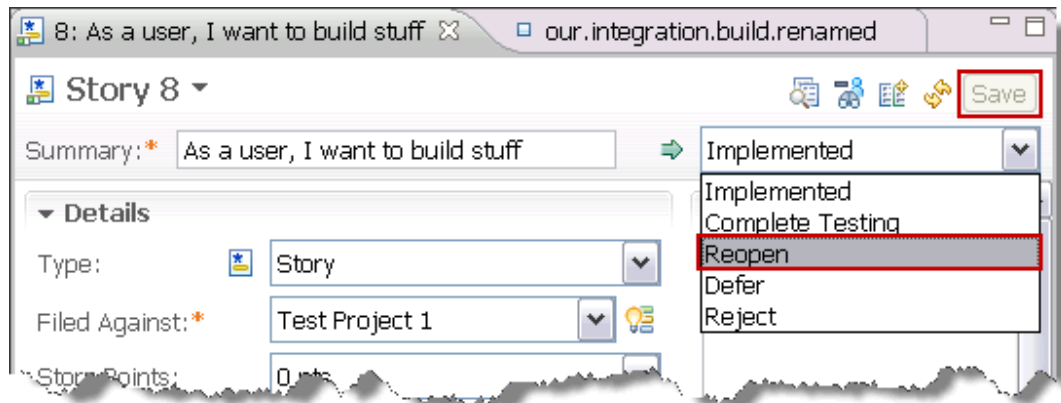
4.5 Change the Build Id in the Configuration and Try Again

- ___1. Return to the Test Project 1 project area editor and change the build id.
- ___a. The editor should still be open to the XML you edited earlier. Find the build-definition element and change the id attribute to `our.integration.build.bogus` and then click **Save** at the upper right of the project area editor. The configuration will now look like this.

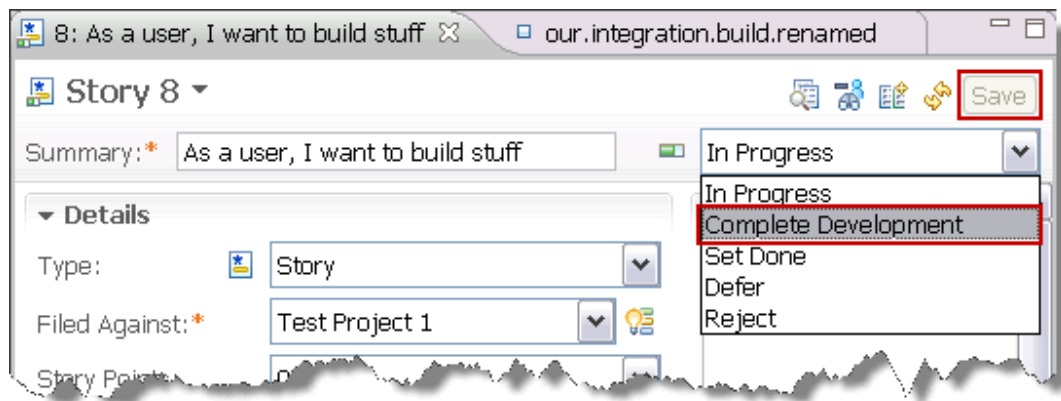
```
<followup-action
  xmlns="http://net.jazz.rtcext.workitem.extensions.service/server/buildOnStateChange"
  description="When the specified work item type changes to the specified state, the
    specified build will be requested."
  id="net.jazz.rtcext.workitem.extensions.service.buildOnStateChange"
  name="Build on State Change">
  <trigger>
    <changed-workitem-type id="com.ibm.team.apr.workItemType.story"/>
    <trigger-state id="com.ibm.team.apr.story.tested"/>
  </trigger>
  <build>
    <build-definition id="our.integration.build.bogus"/>
  </build>
</followup-action>
```

__2. Move the story to the Implemented state again.

- __a. Switch back to the work item editor and select **Reopen** from the state dropdown and then click **Save**. When the debugger surfaces, just click the resume button (). You are not to the interesting bit yet.



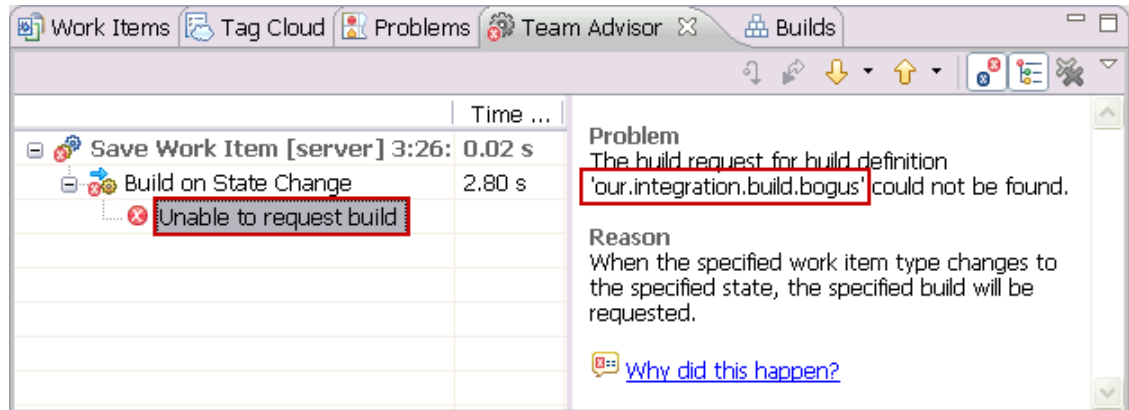
- __b. Again in the work item editor, select **Complete Development** from the same dropdown and click **Save** again.



- __c. This time, when the debugger surfaces, you can step into the configParse2 method to confirm that the new build definition id is returned or you can simply hit resume and trust that the build definition will not be found as expected. Once you do click the debugger's resume button, switch back to your work item editor and note the error.



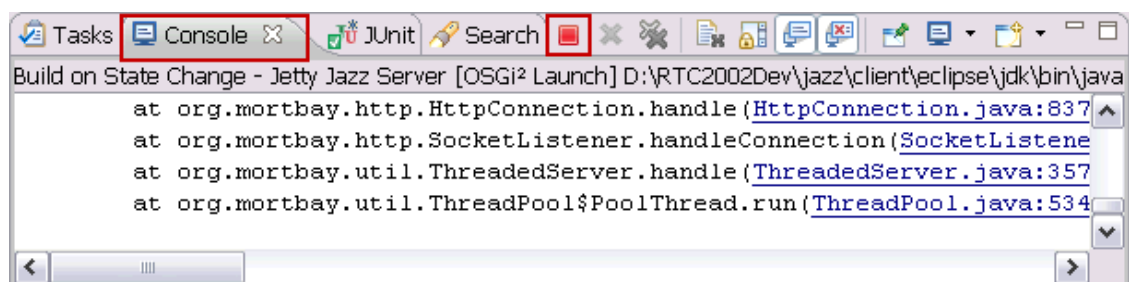
- ___d. The **Team Advisor** view has more information on the error. The left side of the view shows the structure of the error condition. Click the nodes on the left to see what information is available. You can see here that the changed build definition id was used.



- ___e. Switch back to the project area editor and change the **ID** back to `our.integration.build` and click **Save**.
- ___f. Switch back to the work item editor and click **Save**. When the debugger surfaces, you can step into the build method again or just hit resume. Once you do resume, the work item save should complete okay. Return to the work item editor to confirm this. If you go to the **Team Advisor** view and the **Show Failures Only** filter is off, you can browse the results of this successful operation. Also, if you refresh the **Builds** view, you will see another new pending build request.

- ___3. Close down the launched client and server.

- ___a. Close the RTC Eclipse client where you were working with the Story and project area.
- ___b. Back in the original RTC Eclipse client, go to the **Console** view and click the **Terminate** icon.





You have completed lab 4. You can now configure your follow-up action to react to any work item type and state. You can also configure it to run any build. Cool! If you want, you can add multiple instances of the follow-up action to a project or team area and configure each one differently to handle multiple needs.

Lab 5 Adding an Aspect Editor



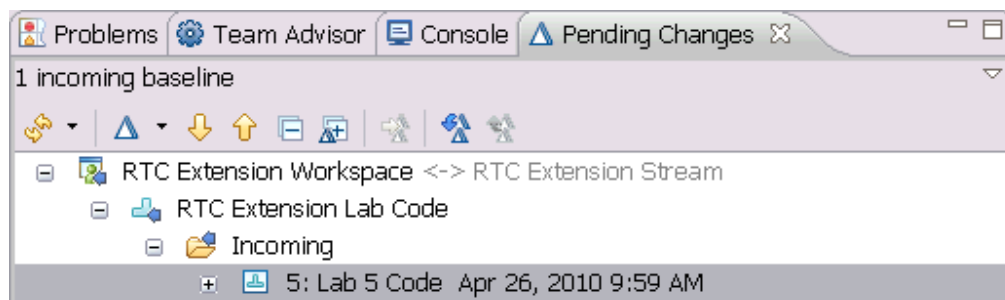
Lab Scenario

No more hard coded ids! Your scrum masters must be thrilled now! Well, not quite. They do not like messing with the process configuration XML. You explain that it is some really simple XML and that assistance is available via Ctrl+Space, but to no avail. Time to brush up on your UI design skills. You will create a simple editor for the participant's aspect editor, an editor responsible for the participant's XML aspect (the small XML bit defined by the participant's schema that extends the process schema).

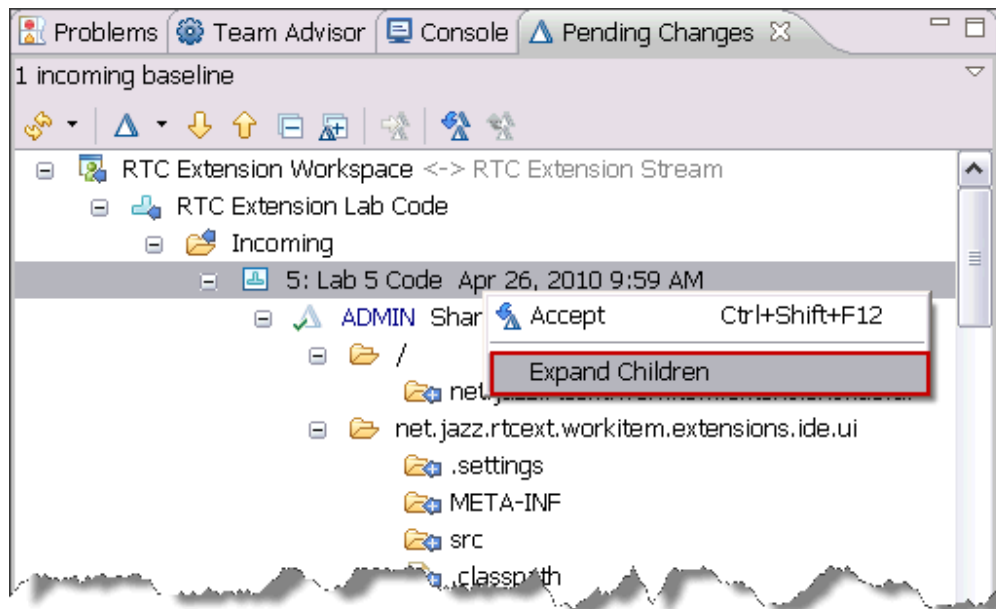
If your RTC server is not running, start it now
(C:\RTC2002Dev\jazz\server\server.startup.bat).

5.1 Understanding the Aspect Editor

- __1. If your RTC development environment is not open, navigate to C:\RTC2002Dev\jazz\client\eclipse in the Windows explorer and double click **eclipse.exe**. If prompted to select an Eclipse workspace, select the same one you used in lab two. If the **Plug-in Development** perspective is not open, open it now by selecting **Window > Open Perspective > Other... > Plug-in Development** from the menu bar.
- __2. Browse and load the Lab 5 code.
 - __a. In the **Pending Changes** view, click the **Expand to Change sets** icon. This will show 1 incoming baseline as shown here.

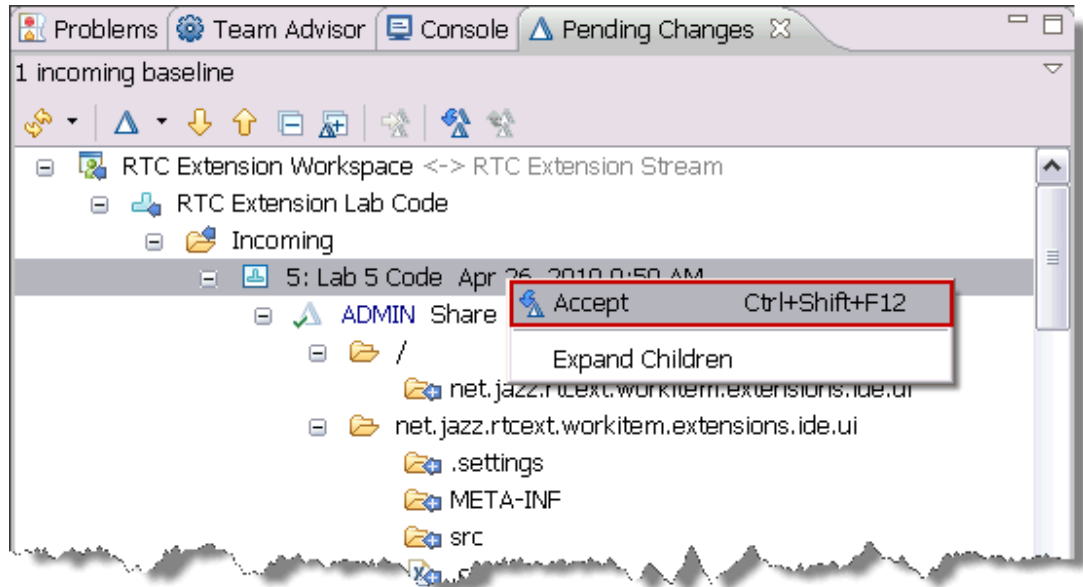


- __b. Right click the **Lab 5 Code** change set under the **RTC Extension Workspace** node, and then click the **Expand Children** action. This will reveal all the changes made for lab 5. As you can see the full change is the addition of a new plug-in project. The first entry shows a folder addition to the root, that folder contains all the other additions in the following changes.



- __c. Browse the changes and you will notice these key changes. The additional behavior will be discussed in detail after the code is loaded.
- __i. A new nested class has been added, `ParsedConfig`. It is a simple structure used to pass configuration results between the parsing stages. Remember, no instance state variables in an operation participant!
 - __ii. Two new parse methods have been added. They perform a two stage parse on the configuration. Note that doing a two stage parse in this case is not really needed since the second stage has no real performance implications, but the pattern will be explained later when you look at the code in detail.
 - __iii. The run method no longer uses hard coded ids.

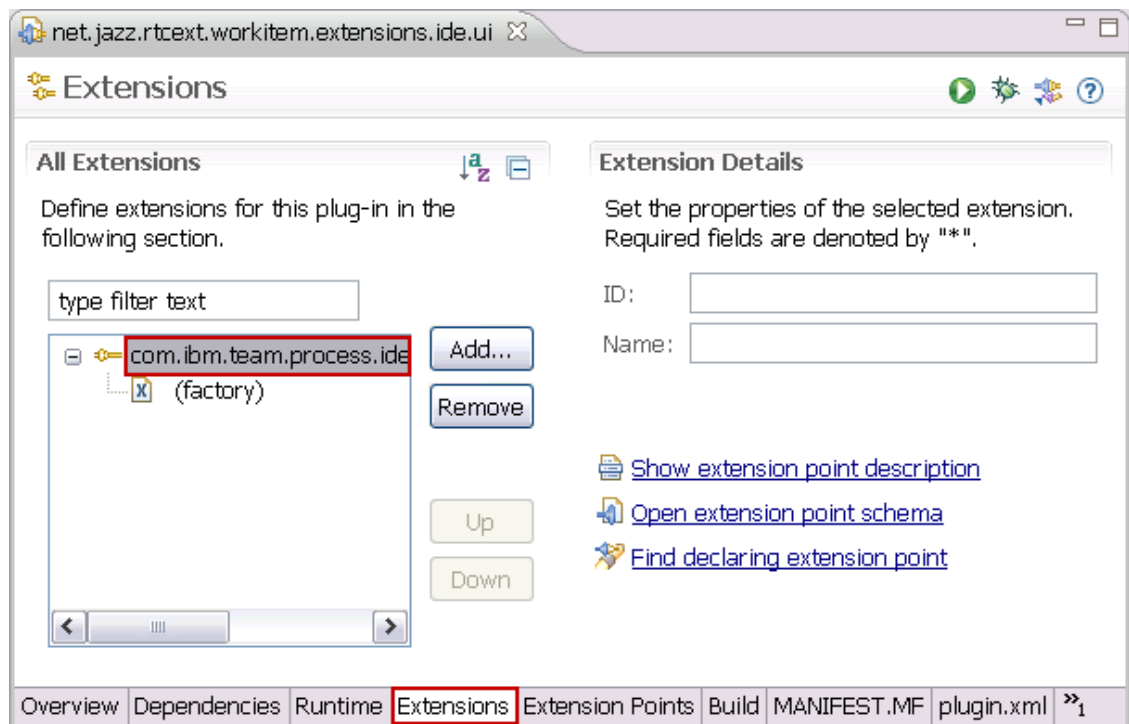
- ___d. In the **Pending Changes** view, right click the **Lab 5 Code** baseline under the **RTC Extension Workspace** node, and then click the **Accept** action. This will accept and load the new lab 5 plug-in project.



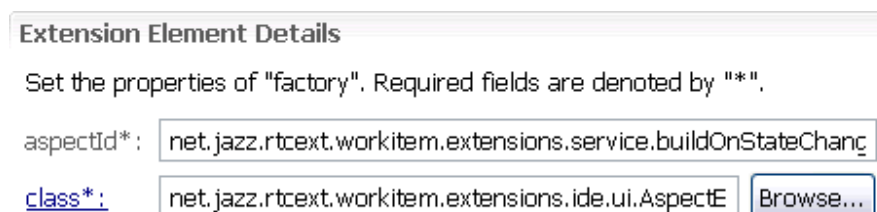
___3. Understanding the aspect editor plug-in.

- ___a. In the **Package Explorer** view, expand the tree for the new user interface project (net.jazz.rtext.workitem.extensions.ide.ui) and double click the **plugin.xml** file. The editor that opens presents information from not only the plugin.xml file but also the build.properties and META-INF/MANIFEST.MF files. As before, the content reflects standard Eclipse plug-in practices. Note on the **Overview** page that there is one significant difference, the addition of an activator class. More on that later when you take a look at that class. Also note on the **Dependencies** page that this plug-in depends on, among other things, the common plug-in but not the service plug-in. The common plug-in, as the name implies, is deployed on both the client and server. The service, just on the server and the aspect editor, just on the client.

- ___b. Once again the most interesting part is on the **Extensions** tab. On the left side, you see an instance of the **com.ibm.team.process.ide.ui.processAspectEditorFactories** extension point. All client side aspect editor factories are defined using this extension point. An aspect editor factory is a class that knows how to construct an aspect editor for one or more process XML aspects. Note that the tree is a structural editor for the xml that comprises the definition. The raw xml can be seen on the **plugin.xml** tab of the editor. The text in parenthesis on each line is the name of the xml element for that line.



- ___c. Select the **(factory)** node in the tree on the left and the right side of the editor will look like the following. The **aspectId** is set to the same value as the participant's id in order to create a link from adding the participant to the process and knowing that this factory needs to be invoked to get the aspect editor. The **class** is set to the factory class. More on that class later when you take a look at it.



__4. Understanding the aspect editor code.

- __a. Back in the **Package Explorer** view, expand the **src/net.jazz.rtcext.workitem.extensions.ide.ui** source package and then double click the **WorkitemExtensionsPlugin.java** file. This is the plug-in's activator class mentioned earlier. This is a very simple class as explained by the class comment.

```
/**
 * Eclipse bundles can optionally contain an activation singleton that is
 * invoked when the bundle is first loaded, usually lazily as in this case. This
 * activator does not do anything interesting on start or stop. However, it is
 * also common practice to have the activation class provide some basic common
 * services that are needed by other classes in your bundle. In the case here,
 * we have common error logging methods for use by the classes in the bundle.
 */
```

- __b. Back in the **Package Explorer** view, double click the **AspectEditorFactory.java** file. This is the aspect editor factory class mentioned earlier.

- __i. This is a very simple class as explained by the class comment. Note that it implements the **IProcessAspectEditorFactory** interface as required by the process editor framework.

```
/**
 * This factory class is configured in the aspect editor extension point, not
 * the aspect editor class itself. One factory may be configured to construct
 * several aspect editors. The process framework passes in the id of the aspect
 * so that the factory knows which to create.
 */
public class AspectEditorFactory implements IProcessAspectEditorFactory {
```

- __ii. It then implements the one method in the interface in a rather straight forward manner. An instance of the `BuildOnStateChangeAspectEditor` class is returned. You will look at that class real soon.

```
/**
 * This is the factory method called by the process framework to get the
 * aspect editor.
 *
 * @param processAspectId
 *         the aspect id as configured in the extension point. One
 *         factory may be configured to construct several different
 *         aspect editors.
 * @return the aspect editor
 */
public ProcessAspectEditor createProcessAspectEditor(String processAspectId) {
    /*
     * If the aspect id is recognized, return the proper aspect editor.
     */
    if (processAspectId.equals(IBMBuildOnStateChangeDefinitions.EXTENSION_ID)) {
        return new BuildOnStateChangeAspectEditor();
    }

    /*
     * It should never happen that an unrecognized id is passed to this
     * method, however, it is common practice to handle that case by
     * throwing an illegal argument exception.
     */
    throw new IllegalArgumentException(NLS.bind("Unknown aspect id: {0}",
        processAspectId));
}
```

- __c. Back in the **Package Explorer** view, double click the **BuildOnStateChangeModel.java** file. The class provides a simple get and set interface for the ids. The class encapsulates reading and writing the XML aspect. There are a few special things about this class as you will see next.

- __i. The get methods are straight forward; however, the set methods are a bit atypical. For example, the set method for the work item type id. Note that the id is normalized (trimmed and never null) and that true is returned if the value actually changed.

```
/**
 * Set access method for the work item type id. The id is normalized and
 * true is returned if a changes is actually made.
 *
 * @param workItemId
 *         the work item type id to set
 * @return true if the value changed, false if it did not
 */
public boolean setWorkItemId(String workItemId) {
    boolean changed = false;
    String normalizedId = normalize(workItemId);
    if (!fWorkItemId.equals(normalizedId)) {
        fWorkItemId = normalizedId;
        changed = true;
    }
    return changed;
}
```

- __ii. The readFrom method should look familiar. It is basically the same as the parse methods that were added to the participant implementation in the last lab. A root object, in this case an IMemento, is passed in and the descendent nodes are searched for the values that are then set into this model. Notice that this method uses the exact same constants from the common plug-in as the participant for the element and attribute names. Note that the root memento comes from the process framework via your aspect editor and that the framework handles the physical reading and parsing of the XML.
- __iii. The saveTo method is the readFrom method's opposite. All the elements and attributes are always written (they are all required and they all can only appear once). The ids are never null; however, they may be empty strings. This leads to a rather straight forward implementation where descendents of the passed memento are added in a fixed manner. Note that the root memento comes from the process framework via your aspect editor and that the framework handles the physical writing of the XML.
- __d. Back in the **Package Explorer** view, double click the **BuildOnStateChangeAspectEditor.java** file. The class provides the actual aspect editor. It is instantiated by the factory and uses the other classes to get its work done. This class is easily the most complicated class in this workshop. You will probably need to debug through parts of it a few times to fully understand it. Here is an overview of each method and type.
 - __i. The class extends the OperationDetailsAspectEditor abstract class.

```

/**
 * The configuration information for an operation participant is stored in the
 * project or team area's process configuration XML. The process framework
 * manages the overall document. For extensions from other components, like this
 * one, the process framework delegates editing of the relevant XML, an aspect,
 * to an aspect editor. The process framework is able to learn from our schema
 * exactly which aspect of the XML to delegate to this editor.
 *
 * This class is an aspect editor for the details of the build on state change
 * follow-up action for work item save. The user can select ids from comboboxes.
 */
public class BuildOnStateChangeAspectEditor extends
    OperationDetailsAspectEditor {

```

- __ii. There are four inherited abstract methods that must be implemented.
 - (1) restoreState(IMemento memento) which passes through to the readFrom(IMemento memento) method on the model class you just studied. Note that this method is always called before createControl.
 - (2) saveState(IMemento memento) which passes through to the saveTo(IMemento memento) method on the model class you just studied.
 - (3) dispose() which does nothing.

- (4) `createControl(final Composite parent, FormToolkit toolkit)` which as the name implies is suppose to create the user interface controls for the aspect editor. The parent composite created by the process editor framework is passed in along with a form toolkit.

```
/**
 * Called by the process editor framework when the user decides to edit the
 * settings for the build on state change operation participant.
 *
 * @param parent
 *     the composite provided by the framework to hold the controls.
 *     This method must set the appropriate layout on this composite.
 * @param toolkit
 *     a control factory provided by the framework. The process
 *     framework specializes the Eclipse UI form toolkit so that the
 *     underlying controls behave properly in the process
 *     configuration editor. All controls are either created directly
 *     from the toolkit or passed to the toolkit's adapt method right
 *     after creation. This makes the aspect editor creator's job
 *     much easier with regard to the proper process configuration
 *     editor look and feel. Note that the control decorations are
 *     not adapted to the toolkit.
 */
public void createControl(final Composite parent, FormToolkit toolkit) {
```


As shown in the implementation of the `createControl` method, there are three basic steps: create the controls, establish the layout data and initialize the user interface values. The implementation of the `createControl` method looks rather straight forward; however, the methods that are called from here are rather complex. Let's look at them and all the other methods and nested types grouped by purpose.

- ___iii. The first group is used to create the user interface controls. They include `createTriggerControls` and `createBuildControls`. These two methods do exactly what their names imply. In addition, they add listeners to the comboboxes to detect changes in selection of the ids.
- ___iv. The second group is for initialization of the user interface. They include `initUI` and `initStates`.
 - (1) The `initUI` method is only called once for any aspect editor instance from the end of `createControls`. It sets the list of values for each combobox and then uses the model to select the proper element of each combobox.
 - (2) The `initStates` method is broken out from the `initUI` method (`initUI` does call it) because it is also needed from the selection listener on the work item type combobox. When the work item type changes, the list of valid states can also change. This method sets the list of values for the work item state combobox and uses the model to select the proper element.

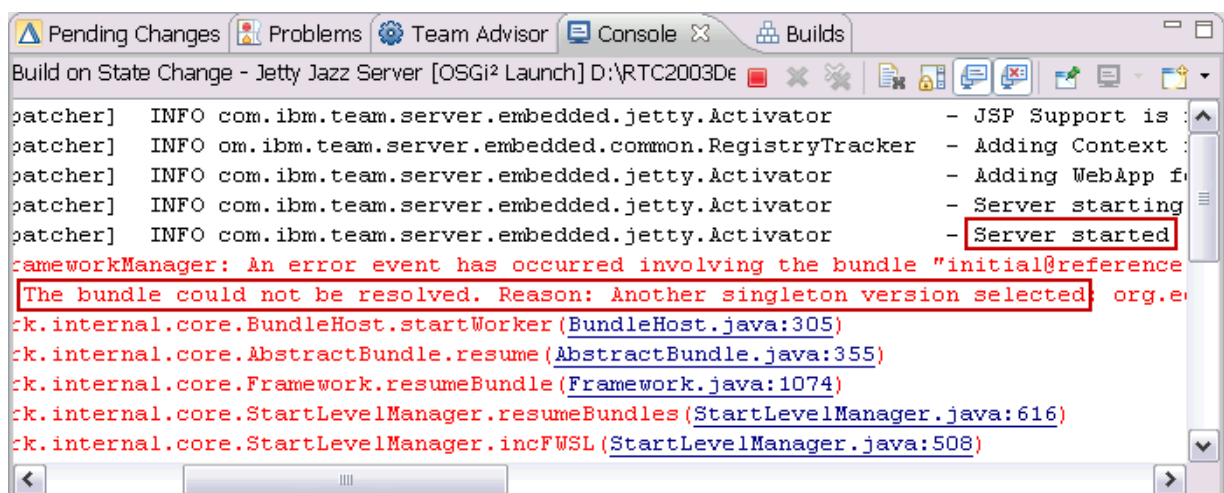
- ___v. The third group is used from the combobox selection changed listeners (and a couple other locations) to validate user selections.
 - (1) The validateSelections method is called whenever a new value is set or selected in the user interface to make sure the user is properly informed as to the validity of the selections.
 - (2) The setValidationMessage method is use by validateSelections to actually manipulate the UI elements that are used to inform the user of validation issues.
- ___vi. The forth group includes the getModel, restoreState and saveState implementations. The getModel method is a straight forward lazy evaluation method for the model instance. The other two pass through to the model as described earlier.
- ___vii. The fifth group includes the getWorkItemCommon and getWorkflowManager methods. These two methods obtain and cache the service objects used to obtain the list of work item types and work item states configured for the project area in which the aspect is being configured. These services are used more than once and are therefore cached. The service used to get the build definitions is only use once per aspect editor instance so it is not cached.
- ___viii. The sixth group includes getWorkItemTypes, getStatesForTypeCategory and their related nested types: WorkItemType and WorkItemState.
 - (1) The nested types are rather simple. Each instance contains the item's id, name and display name. An array of each of these is set as the values for the comboboxes. The comboboxes access the display name via the toString method on each of these nested types. Note that each instance of WorkItemType contains its array of valid WorkItemState instances (the code is actually optimized such that types from the same type group reference the same array of states).
 - (2) The getStatesForTypeCategory method returns an array of WorkItemState instances that are valid for the passed workflow id.
 - (3) The getWorkItemTypes method returns an array of WorkItemType instances that are valid for the project area. It only calculates the list once per aspect editor instance. It also contains the optimization around lists of states for work item types in the same type group. It only calls getStatesForTypeCategory once for each type category.
- ___ix. The seventh and final group includes the getBuildDefinitions method and the BuildDef nested type.
 - (1) The nested type is quite simple. It just contains the id. The toString method is overridden to return the id for display in the combobox.

- (2) The `getBuildDefinitions` method returns an array of `BuildDef` instances that are valid for the project area. It only calculates the list once per aspect editor instance.
- ___e. Next there is the issue of setting breakpoints for your upcoming debug session(s). Recommended locations include the beginning of the `createControl` method and the beginning of the `selectionChanged` method of each selection listener attached to a combobox (there are 3 of them). Also, the `restoreState` and `saveState` methods. Stepping (with a lot of step into) from those points will hit virtually all the code in these classes. You can also clear the breakpoints in the server side participant if you wish. That code has not changed at all for this lab.
- ___f. You can now close all your open editors and proceed to the next section to try out your new aspect editor.

5.2 Launch the Server for Debug Using Jetty

- ___1. Use the existing launch configuration from lab 2.
 - ___a. From the Debug toolbar dropdown () in the toolbar, select **Build on State Change - Jetty Jazz Server**.
 - ___b. The **Console** view will take focus and log messages will start appearing there. Once you see a message that ends in **Server started**, the server is up and ready to be used.

Note: You may see an exception or two concerning the presence of two versions of an Eclipse plug-in being present. The messages contain the text **"The bundle could not be resolved. Reason: Another singleton version selected"** just before the exception's stack trace. You can ignore these exceptions.



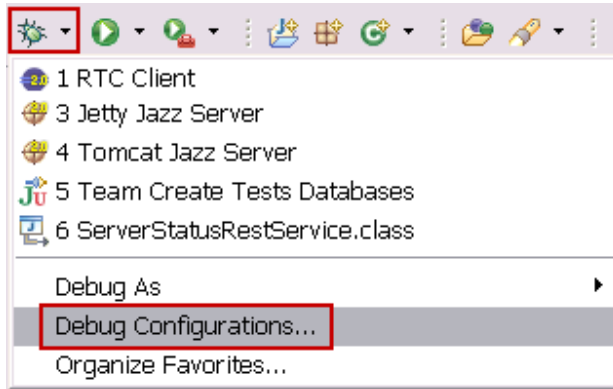
```

patcher] INFO com.ibm.team.server.embedded.jetty.Activator - JSP Support is
patcher] INFO om.ibm.team.server.embedded.common.RegistryTracker - Adding Context
patcher] INFO com.ibm.team.server.embedded.jetty.Activator - Adding WebApp f
patcher] INFO com.ibm.team.server.embedded.jetty.Activator - Server starting
patcher] INFO com.ibm.team.server.embedded.jetty.Activator - Server started
An error event has occurred involving the bundle "initial@reference
The bundle could not be resolved. Reason: Another singleton version selected; org.e
ck.internal.core.BundleHost.startWorker(BundleHost.java:305)
ck.internal.core.AbstractBundle.resume(AbstractBundle.java:355)
ck.internal.core.Framework.resumeBundle(Framework.java:1074)
ck.internal.core.StartLevelManager.resumeBundles(StartLevelManager.java:616)
ck.internal.core.StartLevelManager.incFWSL(StartLevelManager.java:508)
  
```

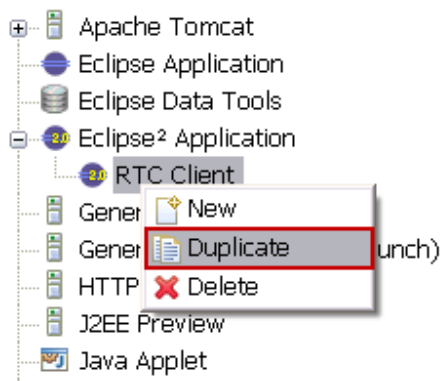
5.3 Launch an RTC Client and Configure the Participant

___1. Create a new launch configuration for the RTC Client plus your aspect editor.

___a. From the **Debug** toolbar dropdown, select **Debug Configurations...**

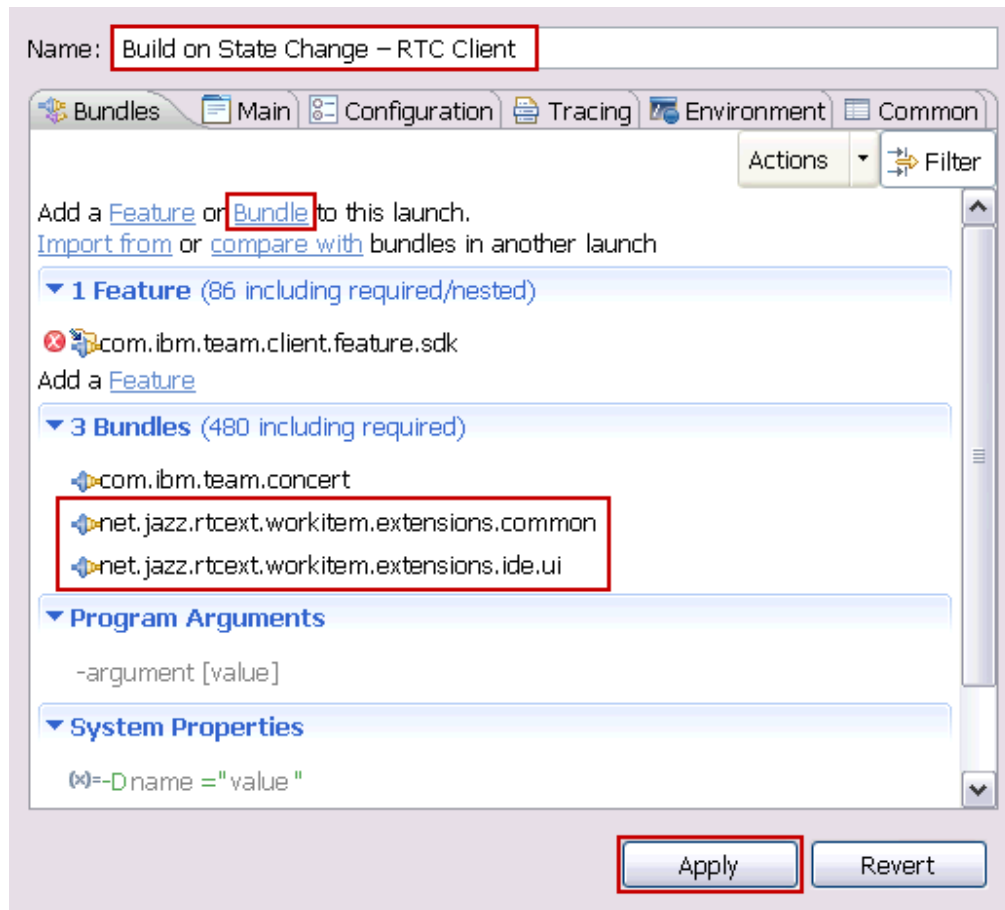


___b. In the **Debug Configurations** dialog, expand the **Eclipse² Application** tree and right click the **RTC Client** configuration and then from the popup menu, select **Duplicate**. Note that you are not changing the existing launch but creating a copy of it. You should keep the original launch around unchanged to use as a known working base from which to create other launch configurations.



___c. Change the **Name** of the new configuration to **Build on State Change - RTC Client**.

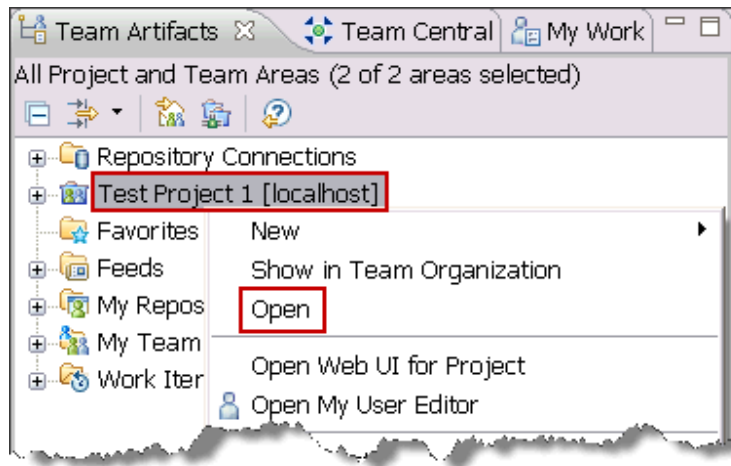
- ___d. Add the common and ui bundles to the configuration. Click on the **Bundle** link and in the **Add Bundle** dialog, type `rttext` in the filter field, select the common plug-in and then click **OK**. Repeat, but select the ui plug-in this time. Your launch configuration should look like this.



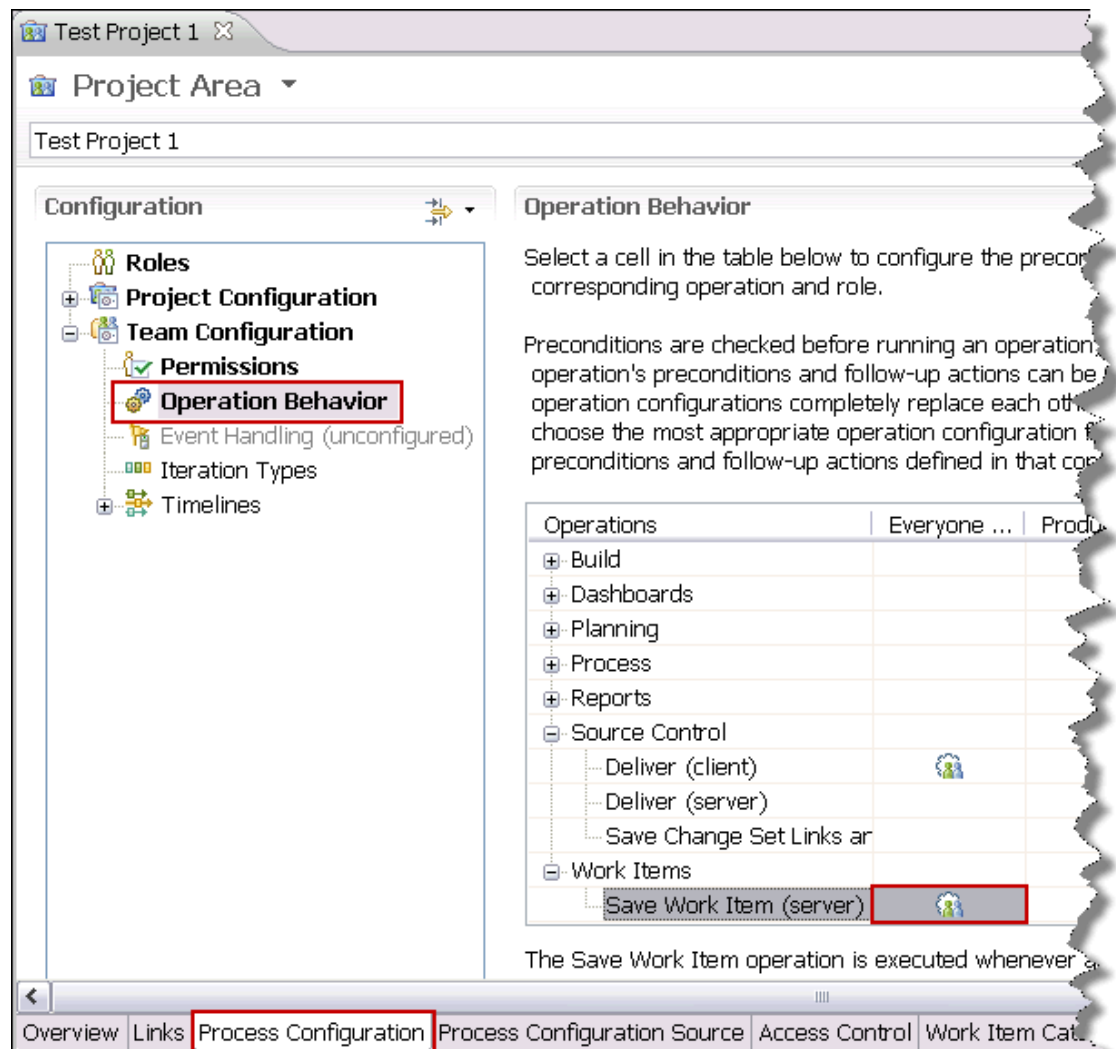
- ___e. Click **Apply** to save your changes but do not close the dialog.
- ___2. Launch the RTC client.
- ___a. Click **Debug** at the bottom of the **Debug Configurations** dialog.
- ___b. The client will launch with the aspect editor included. It will connect automatically to the Jetty server you just launched via the repository connection you created in lab 2 (you may need to enter the password `ADMIN` again). The project area will still be connected and the participant is fully configured from lab 4.
- ___c. The next time you want to debug this server configuration, you will be able to click a shortcut to it on the dropdown of the Debug toolbar icon. You will not need to open the **Debug Configurations** dialog.

___3. Try out the new aspect editor.

- ___a. In the **Team Artifacts** view, right click the **Test Project 1** project area and then click the **Open** action in the menu.



- ___b. In the project editor that opens, switch to the **Process Configuration** tab and then on the left, expand the **Team Configuration** tree then select **Operation Behavior**. Then, on the right, scroll down to the **Work Items > Save Work Item (server)** operation and select the **Everyone (default)** column next to it as shown here.



- ___c. Scroll down to find the **Follow-up actions** section on the right and select the **Build on State Change** entry.
- ___i. If you set it, your breakpoint in the `restoreState` method will trigger. Step into and through the two methods called from here.
 - ___ii. Hit the debugger's resume button and your breakpoint in `createControl` will trigger. Step into and through the methods called from here.
 - ___iii. After you hit resume from `createControl` or one of its called methods, the breakpoints in the selection changes listeners will start to trigger because of the initial setting of the combobox selected element during initialization.
 - ___iv. Once you have hit resume after all the selection change listener breakpoints (each may trigger twice), switch back to the launched RTC Eclipse client and see the aspect editor in action.
- ___d. The selected values in the comboboxes should look familiar. In fact, even better since the actual work item type and state names and not just the ids are shown. Note that the id is all that is put into the process XML.

Name: ☐ Fail if not installed

Description:

When the specified work item type changes to the specified state, the specified build will be requested.

Work Item Trigger

Type Id: * ▼

State Id: * ▼



Build Definition

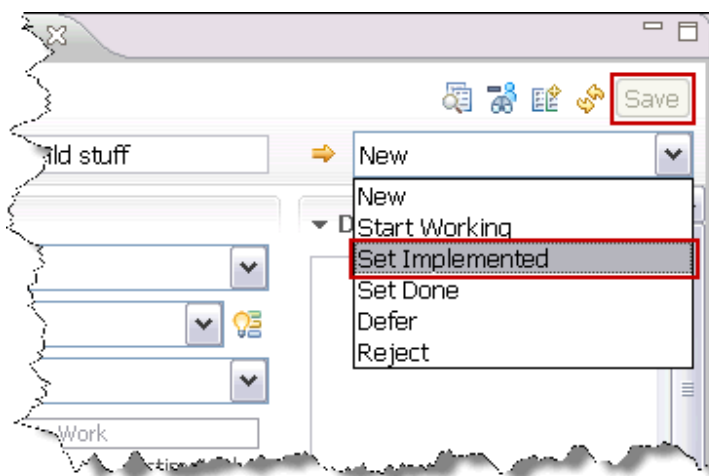
Id: * ▼


- ___e. Select a different work item type and see how the list of states changes in the state id combobox. If the type you chose has a state with the same name, Implemented, the state setting will be recognized as valid even if the id is different. The state id in the model will be updated if required. However, if you choose a work item type that does not have an Implemented state, the state will be flagged as an error. Hover over the little red error icon to see the error message. You may need to try a few times to find a case where the state is still valid after changing the type (hint: Defect and Task both have an "In Progress" state). Also note how the project area editor is marked dirty after your first change and the Save button is enabled. Also note how annoying having all those breakpoints set can be. ☹ You may want to disable some of them.

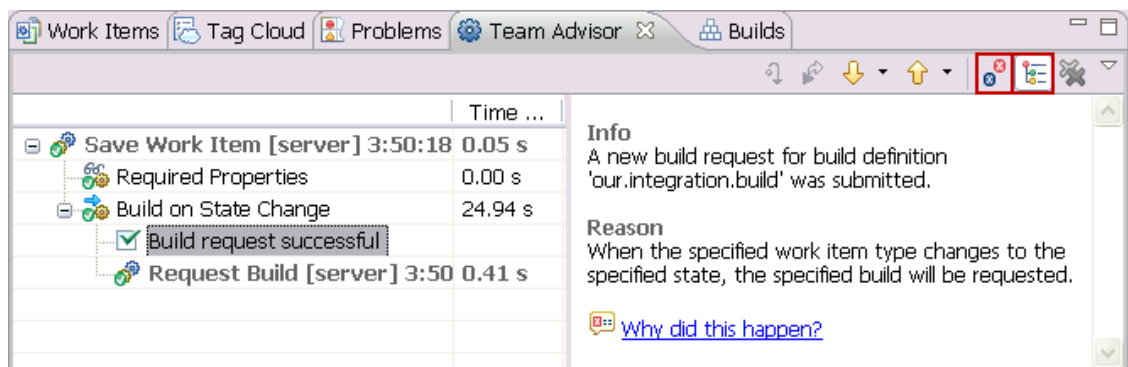
- ___f. When done, click Save at the top right of the project editor and your breakpoint in the saveState method will trigger. Step into and through the called methods if you wish and then return to the launched RTC Eclipse client.
- ___g. **Leave the editor open at this point.** You will soon come back here and make a change.

5.4 Trigger the Participant

- ___1. Depending on how you left the follow-up action configured, you may need to alter these instructions to match your work item type and state.
- ___2. Find the Story work item used in lab 2 and 3 and then move it out of the Implemented state (via the Reopen action) or create a new story.
 - ___a. Either of these will cause the breakpoint you set earlier to trigger (unless you cleared it). The RTC Eclipse client in which you were studying the code will now surface (if asked about switching to the debug perspective, click **Yes**). If it does not surface, you probably minimized it earlier. In this case, it will be flashing in the Windows taskbar. Click it in the taskbar to surface the debugger.
 - ___b. Simply resume execution since this code has not changed ().
 - ___c. Switch back to the RTC Eclipse client where you created the work item. Your work item will be successfully saved. If it shows a failure due to timeout, close the editor without saving, recreate the Story (or reedit the existing Story) and when the breakpoint hits, just use the resume button ().
- ___3. Move the Story to the Implemented state (or your different type to the trigger state).
 - ___a. At the upper right portion of the work item editor, select **Set Implemented** or **Complete Development** (depends on which workflow state the story is currently in) and then click **Save**.



- ___b. Once again the breakpoint is hit (unless you cleared it) and your debugger surfaces. Go ahead and resume again.
- ___c. Switch back to the RTC Eclipse client where you created the work item. Your work item will be successfully saved. If it shows a failure due to timeout, try saving again and when the breakpoint hits, just use the resume button ().
- ___d. If you go to the **Team Advisor** view and check to make sure the **Show Failures Only** filter is off and **Show Detail Tree** is on (see highlight below), you can browse the results of this successful operation. Also, if you refresh the **Builds** view, you will now see a new pending build request.



5.5 Add another Instance of the Follow-up Action and Try Again

- __1. Return to the Test Project 1 project area editor and add another instance.
 - __a. The editor should still be open to where you were before. Next to the Follow-up action list, click Add... and in the Add Follow-up Actions Dialog, select Build on State Change from the list and click OK. Only the restoreState and createControl breakpoints will trigger this time. The process configuration editor will now look like this. Note the errors. None of these can be empty.

The Save Work Item operation is executed whenever a work item is saved in the repository.

☒ Preconditions and follow-up actions are configured for this operation

☐ Final (ignore customization of this operation in child team areas)

Preconditions (6 available):

Name: ☐ Fail if not installed

Description:

When the specified work item type changes to the specified state, the specified build will be requested.

Follow-up actions (1 available):

Work Item Trigger

Type Id:

State Id:

Build Definition

Id:

- __b. Select a work item and state that are different from the ones configured for the first instance. Select the one and only build definition. If you wish, you can create a new build definition. If you do create a new build definition, you will not see it until a new instance of the aspect editor is created. A new instance is created each time you select a participant in the Follow-up actions list.
- __2. Now create a new work item of the type you selected and move to the selected state. Once you do, a build will be submitted. It will still work for the original settings too.
 - __3. Close down the launched client and server.
 - __a. Close the RTC Eclipse client where you were working with the work items and project area.

- ___b. Back in the original RTC Eclipse client, go to the **Console** view and click the **Terminate** icon.



You have completed lab 5. You can now configure your follow-up actions using a nice aspect editor. What could the scrum masters possibly ask for next?

Lab 6 Deploying the Server Side



Lab Scenario

Now the code is really complete. Only the deployment to the production environment is left to do. This lab will concentrate on the server side deployment.

Client side deployment of the common and ui plug-ins is rather simple and well documented elsewhere. Since only people that will modify the process configuration need the client side plug-ins, they could simply place them in there dropins folder (C:\RTC2002Dev\jazz\client\eclipse\dropins in this lab setup). Alternatively, a client side feature and update site could be created as described at

http://wiki.eclipse.org/FAQ_How_do_I_create_an_update_site_%28site.xml%29%3F.

Actually, the server side deploy contains all those same steps plus a couple more. So, you can also use this lab as a guide for a client side update site too. Up to the server side specific steps and except for which plug-ins to include:

- common and ui on the client
- common and service on the server

If your RTC server is not running, start it now

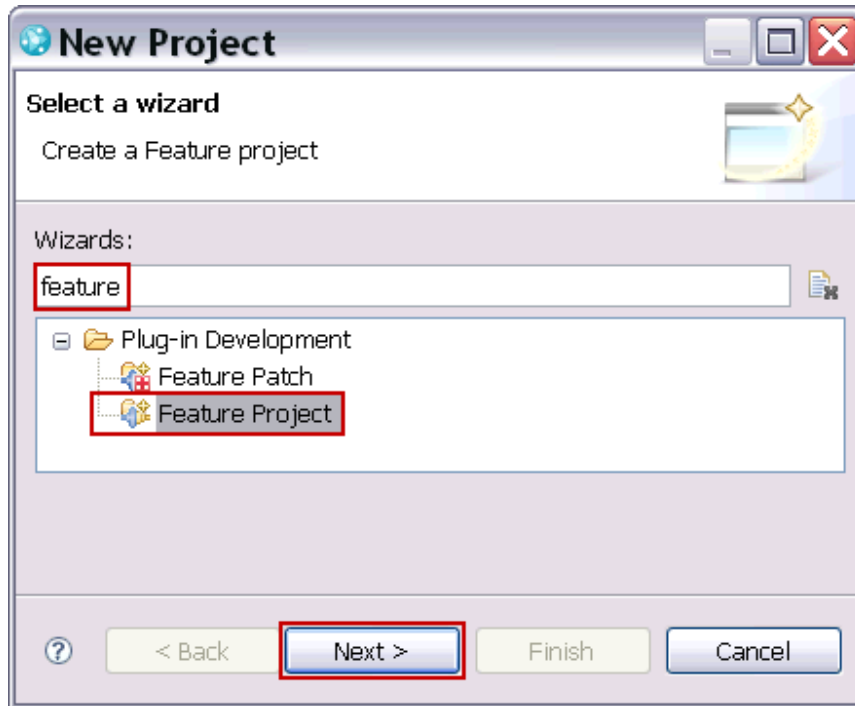
(C:\RTC2002Dev\jazz\server\server.startup.bat).

6.1 Creating a Server Side Feature

1. If your RTC development environment is not open, navigate to C:\RTC2002Dev\jazz\client\eclipse in the Windows explorer and double click **eclipse.exe**. If prompted to select an Eclipse workspace, select the same one you used in lab two. If the **Plug-in Development** perspective is not open, open it now by selecting **Window > Open Perspective > Other... > Plug-in Development** from the menu bar.

___2. Create the server side feature.

- ___a. From the menu bar, select File > **New > Project...** then in the **New Project** wizard, type **feature** in the filter field, select **Feature Project** from the list and then click **Next**.



- __b. On the second page of the wizard type `net.jazz.rtcext.workitem.extensions.server.feature` into the **Project name** field. As you type, the **Feature ID** is set to a reasonable value but the **Feature Name** should be reset to like: `Work Item Extensions Server Feature`. You can set the **Feature Provider** to yourself or your company, if you wish. It is not required. Click **Next**.

New Feature

Feature Properties
Define properties that will be placed in the feature.xml file

Project name: `net.jazz.rtcext.workitem.extensions.server.feature`

☒ Use default location
Location: `D:\RTC2003Dev\DevWS\net.jazz.rtcext.workitem` Browse...

Feature properties

Feature ID: `net.jazz.rtcext.workitem.extensions.server.feature`

Feature Name: `Work Item Extensions Server Feature`

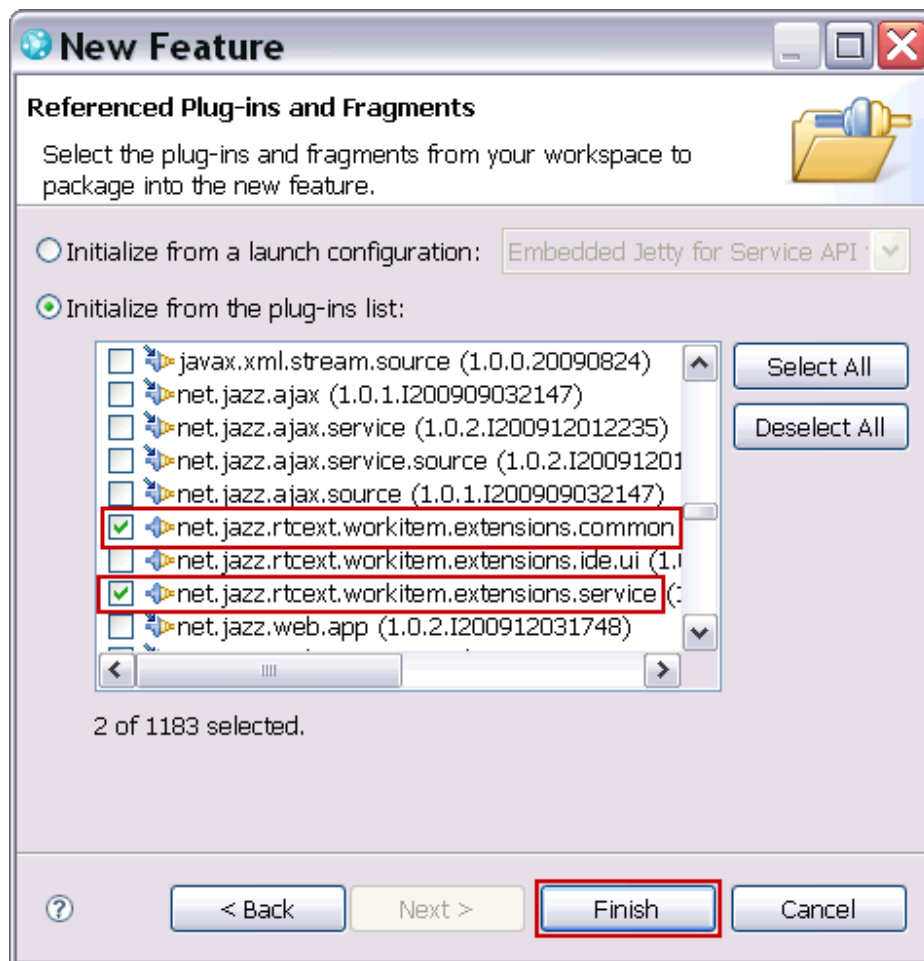
Feature Version: `1.0.0`

Feature Provider:

Install Handler Library:

? < Back **Next >** Finish Cancel

- ___c. On the final page of the wizard select the common and service plug-ins and then click **Finish**.



- ___d. Your new feature project appears in the **Package Explorer** view and an editor opens on the feature.xml file. On the Overview tab, change the Version to 1.0.0.qualifier. This is the same standard Eclipse practice you used for the plug-ins.

General Information

This section describes general information about this feature.

ID:

Version:

Name:

Provider:

Branding Plug-in:

Update Site URL:

Update Site Name:

- ___e. Still in the editor, switch to the **Information** tab, select the **Feature Description** sub-tab and enter a **Text** description as shown here. If you wish you can look at other information that can be added, such as a copyright and license information.

*net.jazz.rtext.workitem.extensions.server.feature

Information

Enter description, license and copyright information. Optionally, provide links to update sites for installing additional features.

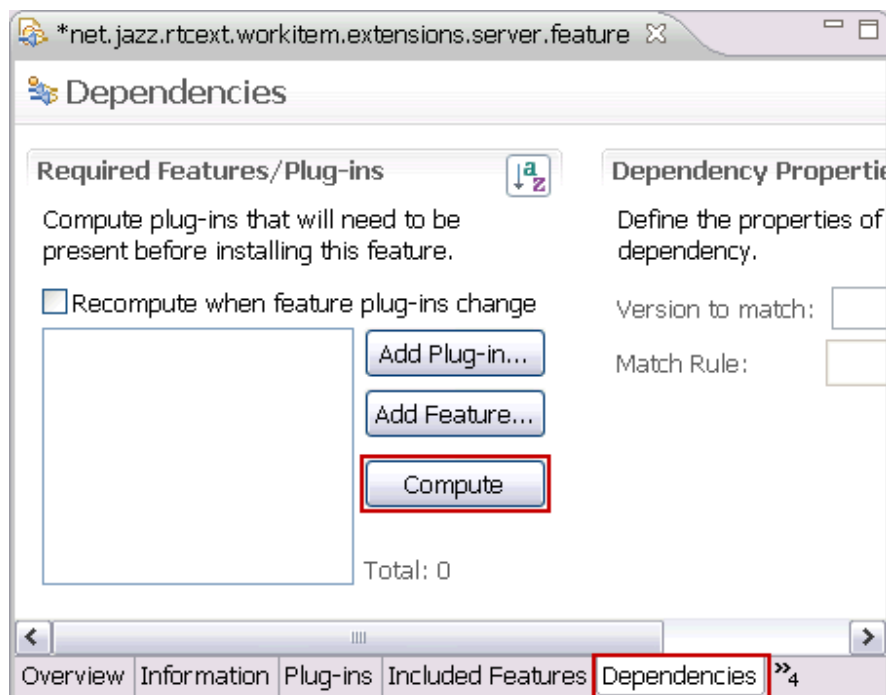
Feature Description Copyright Notice License Agreement Sites to Visit

Optional URL:

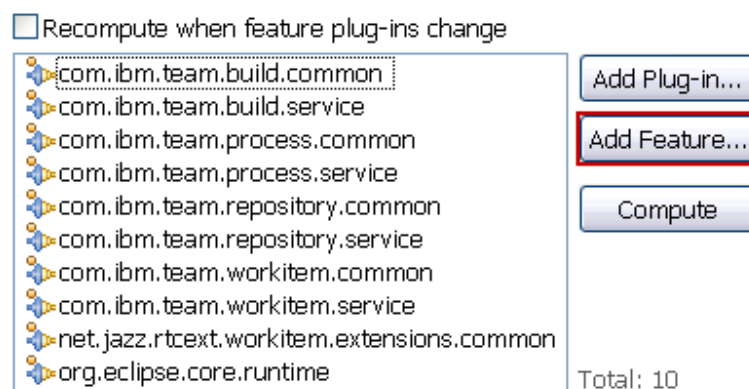
Text:

Overview **Information** Plug-ins Included Features Dependencies Installation Build »

- __f. Switch to the **Dependencies** tab and click **Compute**.

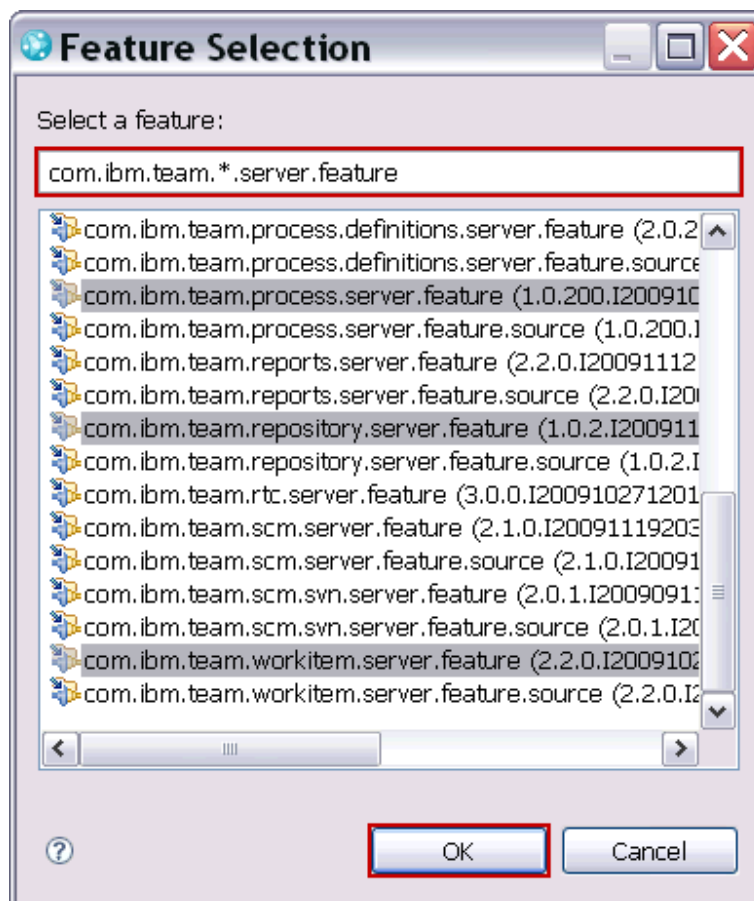


- __g. The dependencies list is computed as shown here. The dependencies are expressed in terms of plug-ins; however, for Jazz server side provisioning, you need to use features. Using the compute button was helpful because having the list of plug-ins makes it straight forward to figure out the list of features you really want. You will need four server side features in the dependency list: one each for repository, process, workitem and build. The server side features on which you will generally depend (the ones that provide services that you will use from these and other plug-ins) follow a consistent naming pattern: **com.ibm.team.component.server.feature**. Click **Add Feature...**

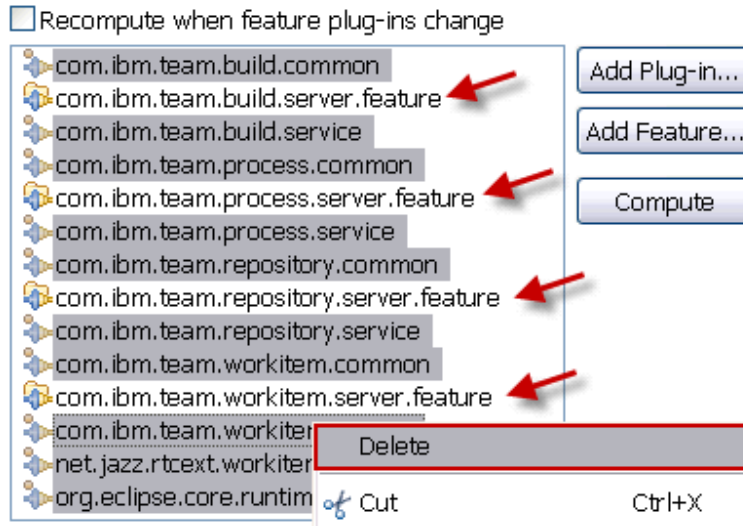


- ___h. In the **Feature Selection** dialog type `com.ibm.team.*.server.feature` into the filter field, select these four features and then click **OK**.

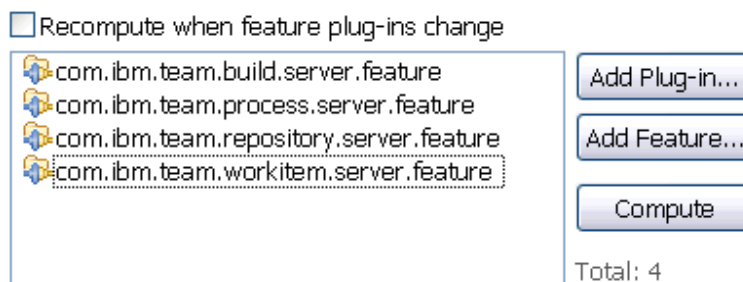
`com.ibm.team.build.server.feature`
`com.ibm.team.process.server.feature`
`com.ibm.team.repository.server.feature`
`com.ibm.team.workitem.server.feature`



- ___i. The dependency list will now contain the four features (red arrows) in addition to plug-ins it had before (selected). Select all the plug-ins as shown here, right click one of them and then select **Delete** from the menu.

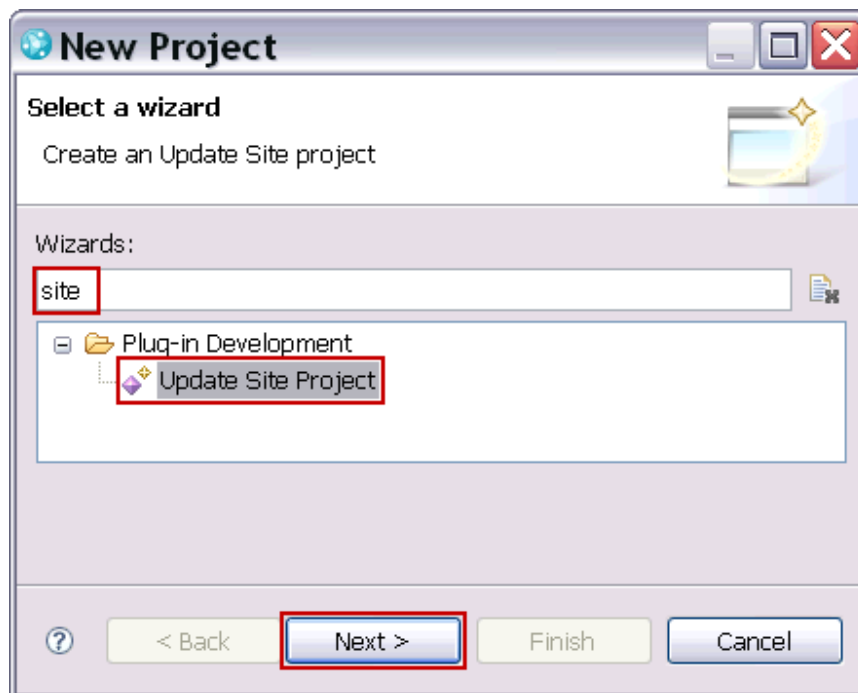


- ___j. The list will now look like this. Type **Ctrl+S** to save the feature.xml file. You can now close the editor.

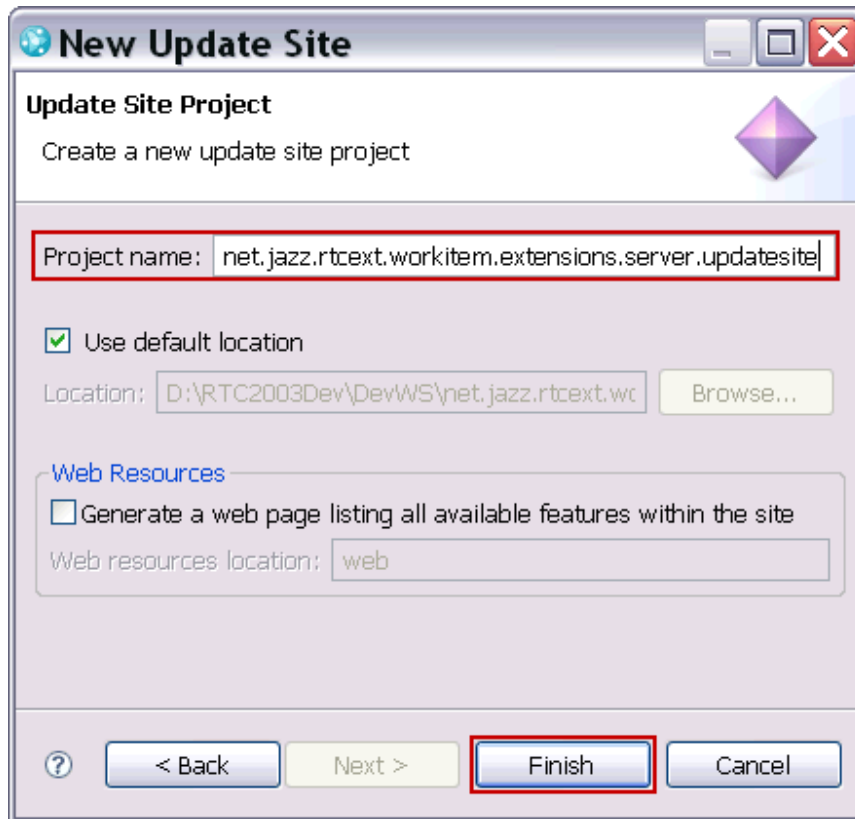


6.2 Create the Server Update Site

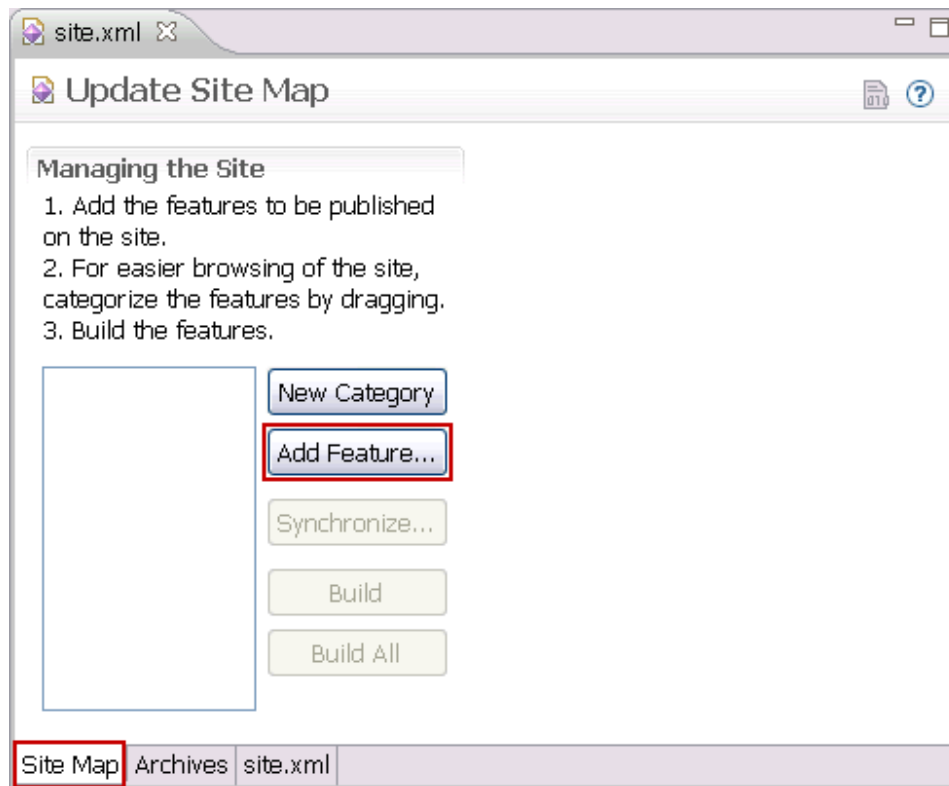
- ___1. Create the update site.
 - ___a. From the menu bar, select File > **New > Project...** then in the **New Project** wizard, type **site** in the filter field, select **Update Site Project** from the list and then click **Next**.



- __b. On the second page of the wizard type `net.jazz.rtcext.workitem.extensions.server.updatesite` into the **Project name** field. Click **Finish**.



- ___c. Your new update site project appears in the **Package Explorer** view and an editor opens on the site.xml file. In the editor, remain on **Site Map** tab and click **Add Feature**.

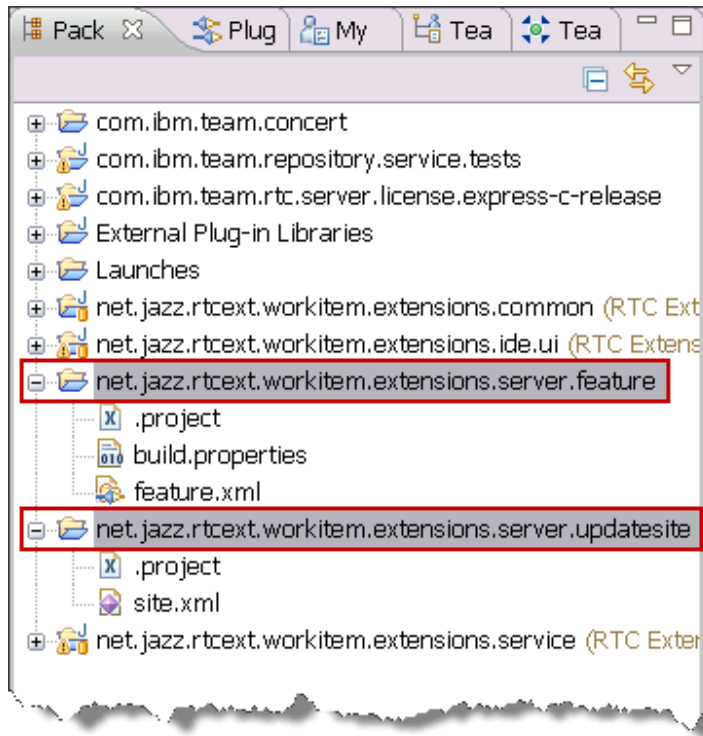


- ___d. In the **Feature Selection** dialog, type `*rttext` into the filter, select the feature you created in the last section and then click **OK**. Back on the site.xml editor type **Ctrl+S** to save the file.

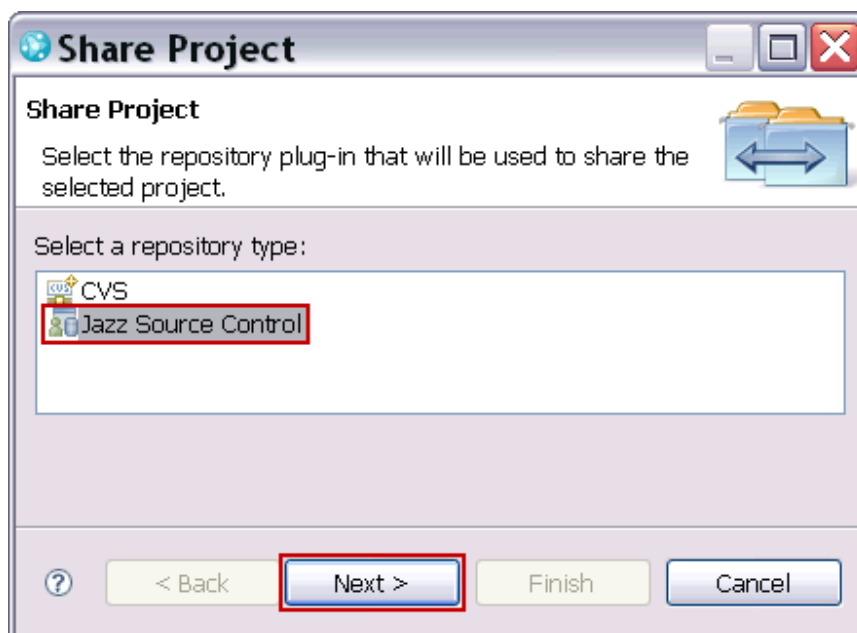


__2. Share the new projects to your repository workspace.

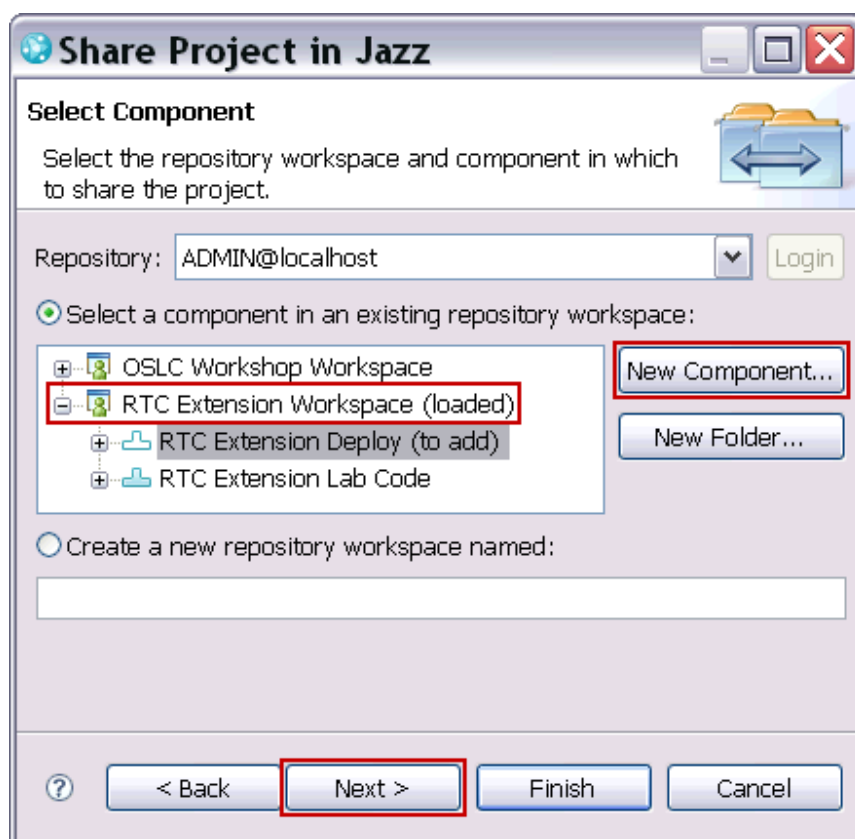
__a. In the **Package Explorer** view, select the feature and update site projects as shown here. Then, right click one of them and from the menu, select **Team > Share Project...**



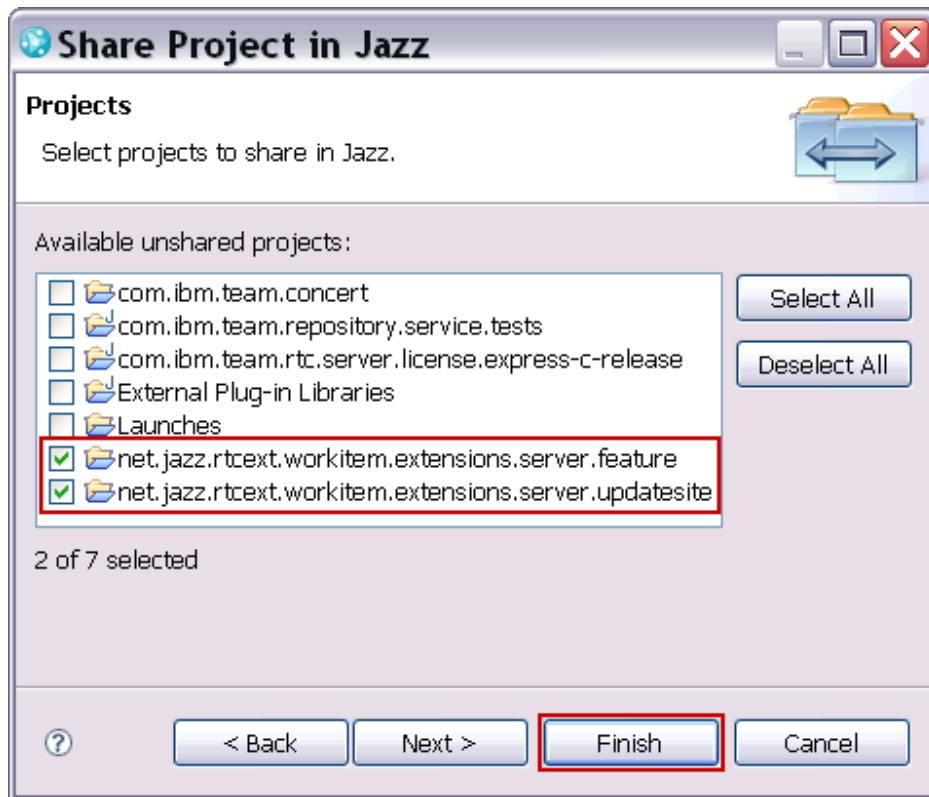
__b. In the **Share Project** wizard, select **Jazz Source Control** then click **Next**.



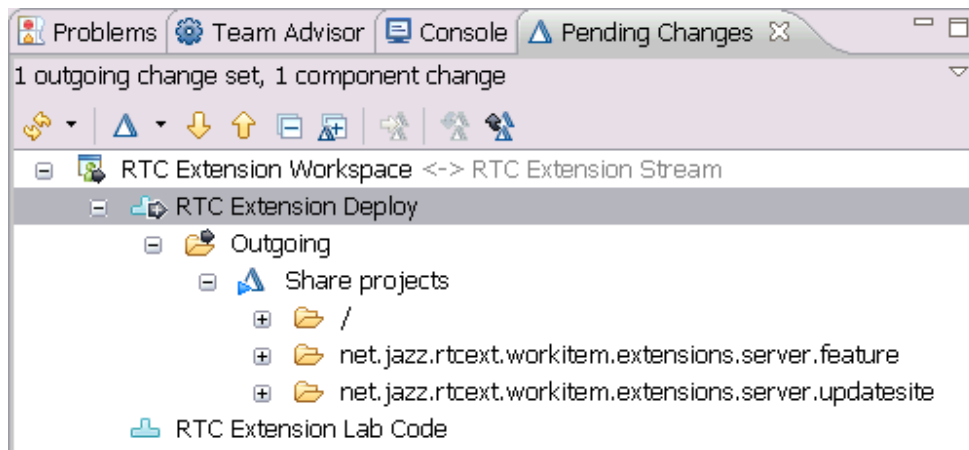
- ___c. On the second page of the wizard, select the **RTC Extension Workspace** (as highlighted with a red box) and click **New Component**. In the **New Component** dialog, enter `RTC Extension Deploy` as the component name and click **OK**. Finally, back to the wizard, make sure the new component is selected (as show with gray highlight) and then click **Next**.



- ___d. On the third page of the wizard, confirm that the feature and update site projects are selected and then click **Finish**.

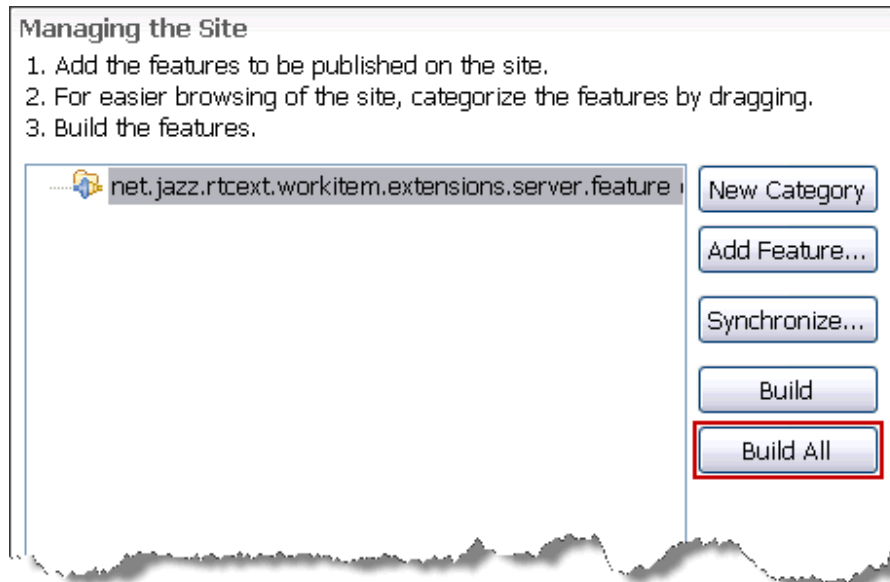


- ___e. The **Pending Changes** view will show your outgoing component addition with its newly shared projects. You will deliver them later.

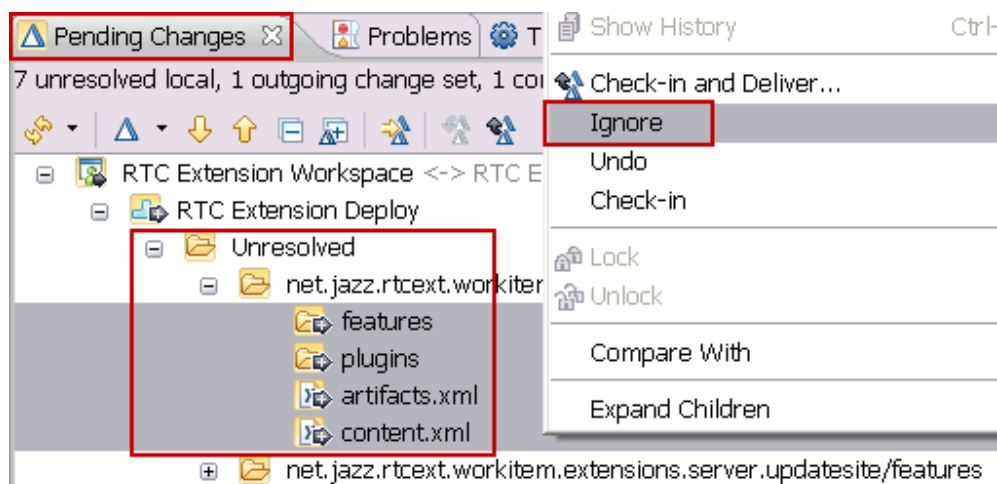


___3. Build the update site.

___a. Return to the site.xml editor and click **Build All**.

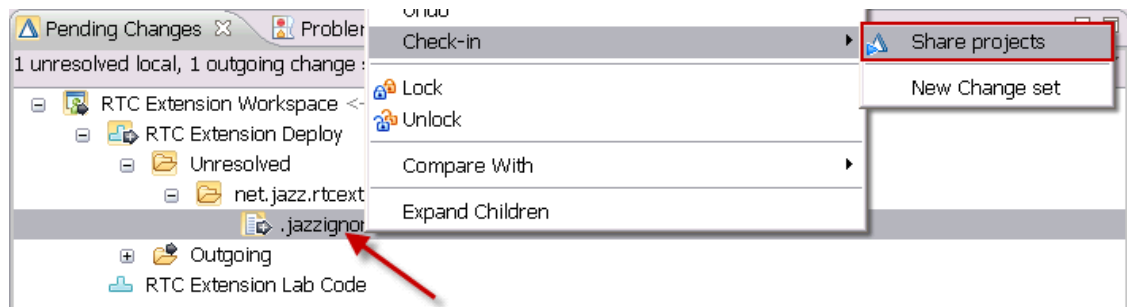


___b. The **Package Explorer** and **Pending Changes** views will show several new files in your update site project. In the Pending Changes view they will show up as **Unresolved**. Select all four entries in the root of the update site as shown here. Then, right click one of them and from the menu select **Ignore**. When prompted to confirm, click **Yes**. A dialog that explains how to un-ignore the resources later may appear. Click **OK** if it shows up. These files are created by the update site build and do not need to be stored under source control. This action along with the next sub-step will make sure you do not accidentally check them at another time.



Note that the artifacts.xml and content.xml files are used for the new P2 style update sites. The jazz server side provisioning does not use them at this time. However, if you create an update site for the client side plug-ins, you can create a P2 enabled update site.

- ___c. The **Pending Changes** view will now show a new **.jazzignore** file as **Unresolved**. Go ahead and check it in now by right clicking the file and then selecting **Check-in > Share projects** from the menu.



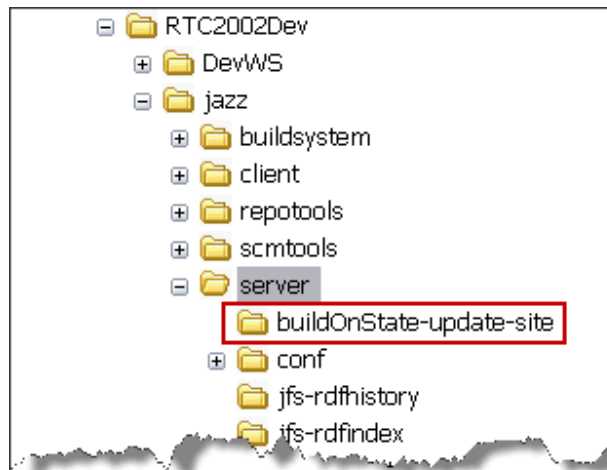
- ___d. If you now dig into the site.xml file and into the jars in the features and plugins folders inside the update site project, you will notice that all the update site build had converted all the “qualifier” segments of the version numbers to date and time stamps. This will make it easier to update your code in a test system during development. One final note. Generally, if you need to build the update site again, you will first want to delete the jars from the update site project’s features and plugins folders. The build will generate new jars with different date and time stamps and leave the old ones there too.

6.3 Deploy the Server Side Feature

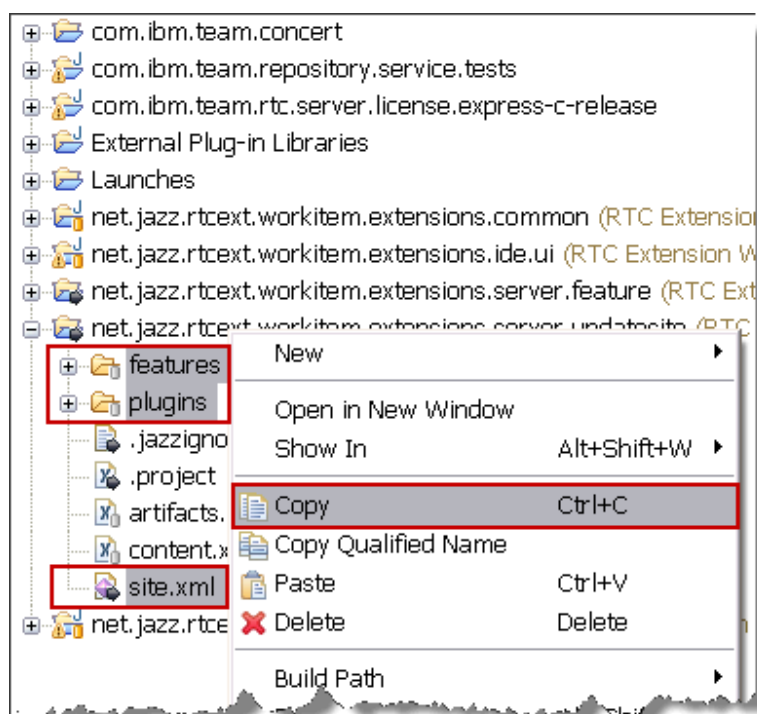
- ___1. Shutdown the RTC server.
- ___a. In the Windows Explorer, navigate to C:\RTC2002Dev\jazz\server and run the **server.shutdown.bat** file.
- ___b. Switch to the Tomcat window where the server is running. You may need to hit Enter after the message “INFO: Failed shutdown of Apache Portable Runtime” appears to completely shut the server down. Recall from lab 1 that this is because you enabled this server to have a debugger attached to it.

__2. Copy the update site into place.

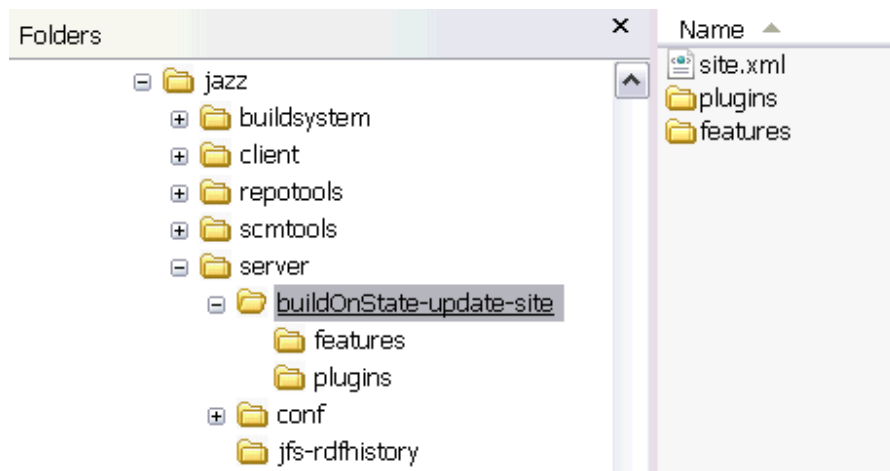
- __a. In the Windows Explorer, navigate to C:\RTC2002Dev\jazz\server and create a new folder to contain the extension. For this lab, call it buildOnState-update-site as shown here.



- __b. In the Package Explorer view, select the **site.xml** file and the **features** and **plugins** folders as shown here. Then right click one of them and select **Copy** from the menu.



- ___c. Back in the Windows Explorer, select the buildOnState-update-site folder and paste the extension update site into it. Here is the result.



- ___3. Create the provisioning ini file.

- ___a. In the Windows Explorer, navigate to C:\RTC2003Dev\jazz\server\conf\jazz\provision_profiles and create a new file in that folder named buildOnState.ini.

- ___b. Open the new ini file with Notepad and enter these two lines:

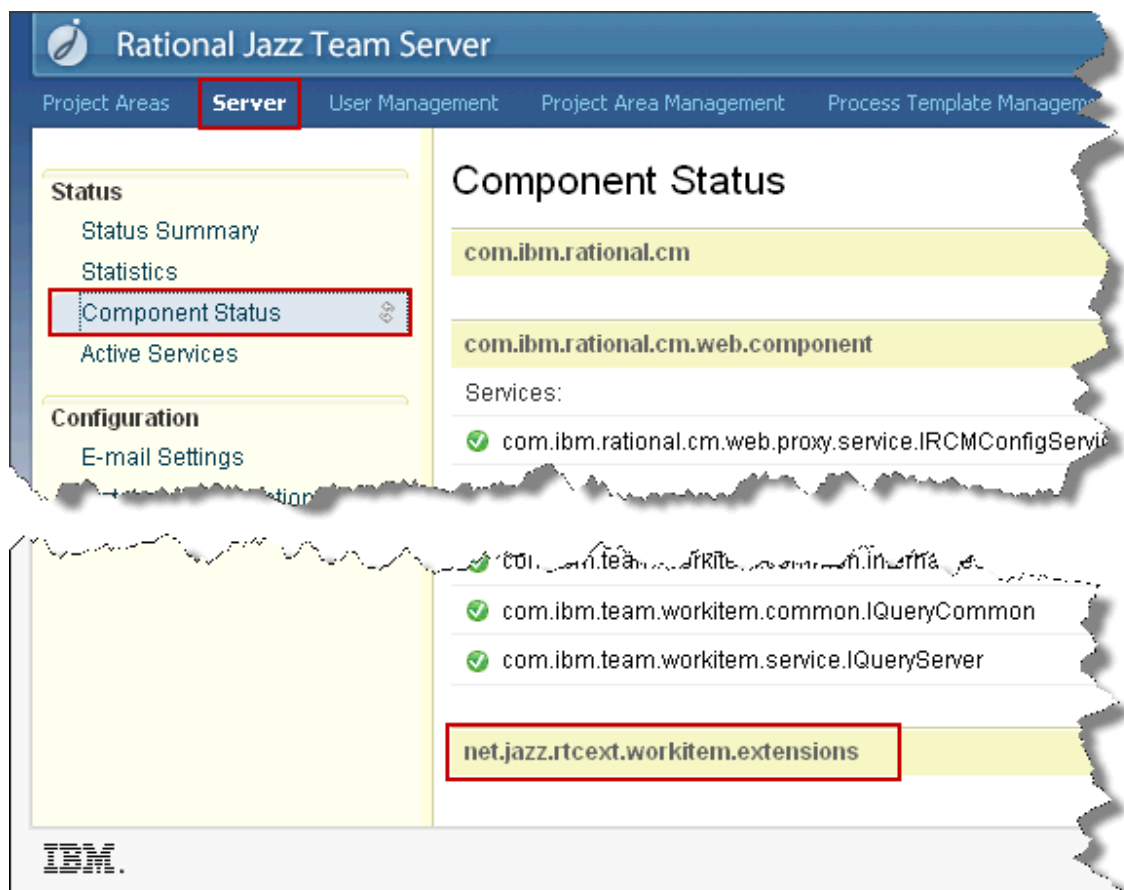
```
url=file:../buildOnState-update-site
featureid=net.jazz.rtctx.workitem.extensions.server.feature
```

- ___c. Save the file and close the editor. When you restart the RTC server, it will read this new provisioning ini file and find the path to the update site and the id of the new feature to load.

- ___4. Start the RTC server.

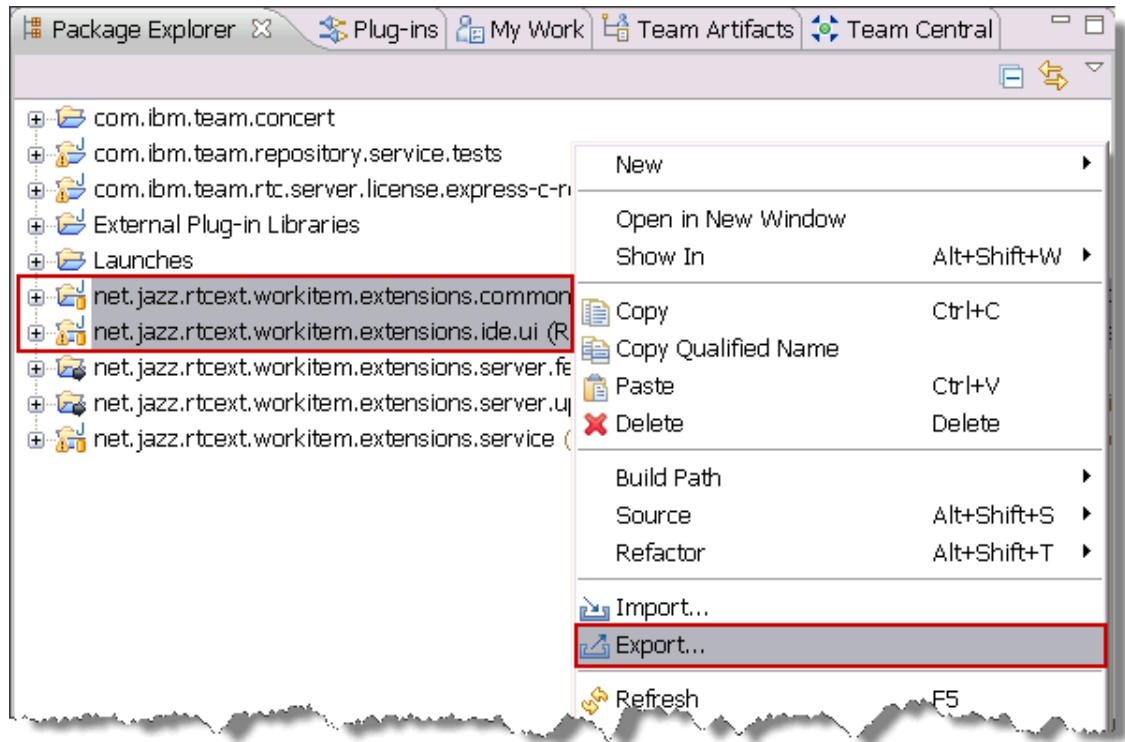
- ___a. In the Windows Explorer, navigate to C:\RTC2002Dev\jazz\server and run the **server.startup.bat** file.

- ___b. If you open your browser to this URL (<https://localhost:9443/jazz/admin>), login as ADMIN / ADMIN, make sure you are on the **Server** page, select the **Component Status** link on the left and then scroll to the bottom. You will see the **net.jazz.rtcext.workitem.extensions** component is running. It does not show any services since it just contains the operation participant.

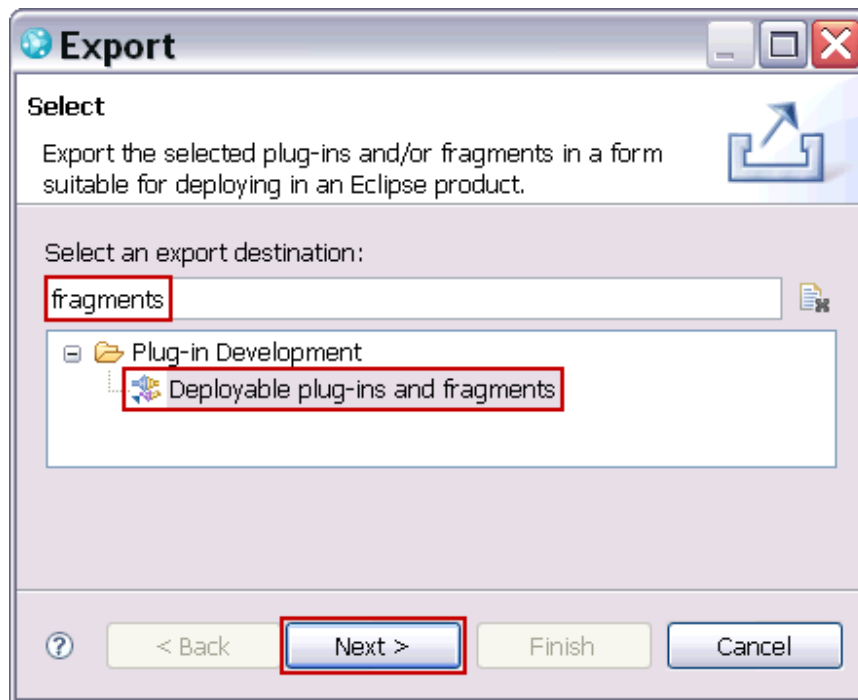


6.4 Deploy the Client Plug-ins

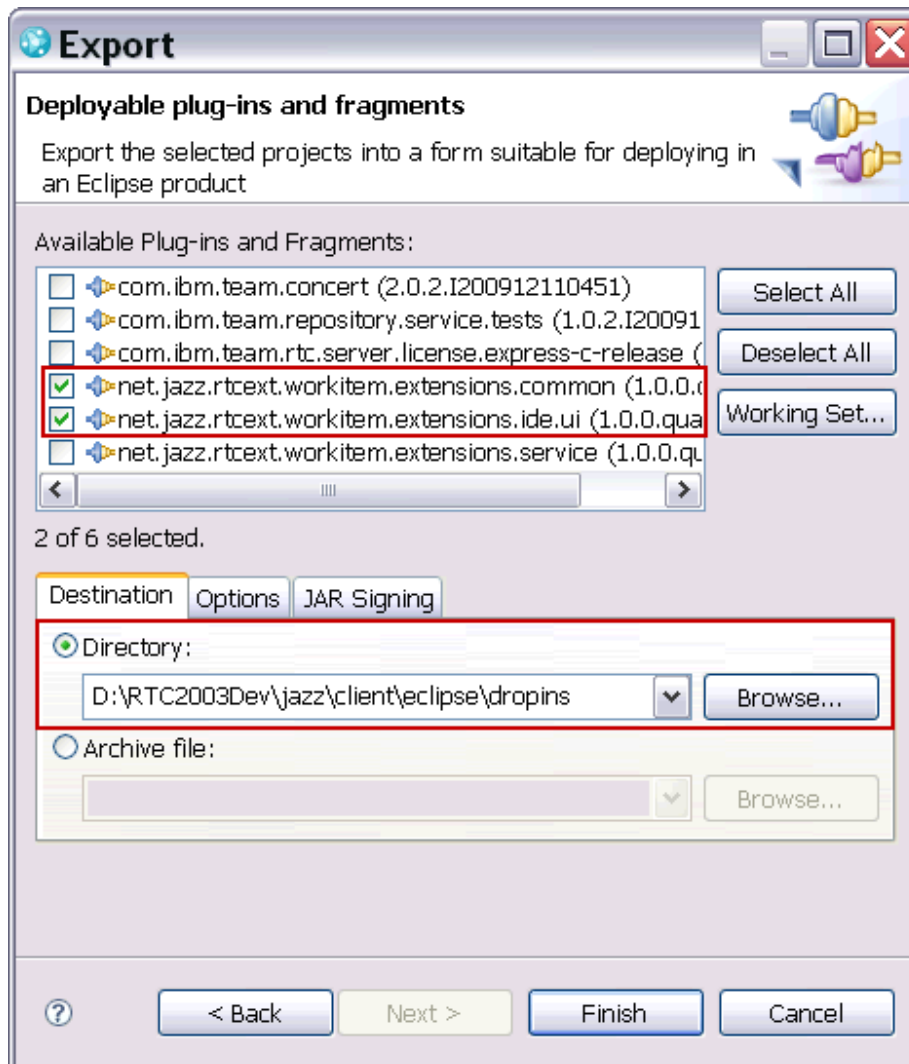
- ___1. Export deployable plug-ins to the drop-ins folder.
 - ___a. In the **Package Explorer** view, select the common and ui plug-ins as shown here and then right click one of them and select **Export...**



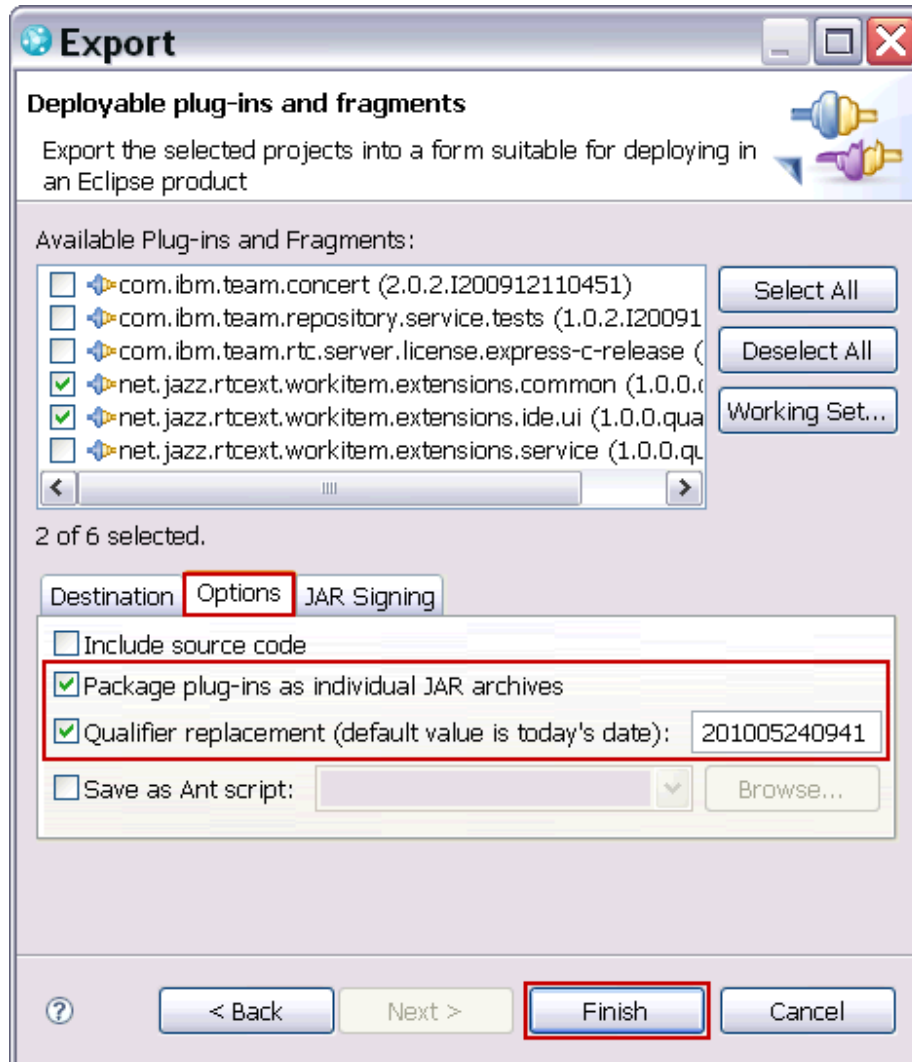
- __b. In the **Export** wizard, type fragments in the filter, select **Deployable plug-ins and fragments** from the list and then click **Next**.



- ___c. On the second page of the wizard, make sure the common and ui plug-ins are selected and specify the RTC Eclipse client's dropins folder as the destination as shown here. You may want to use the **Browse...** button. Do NOT hit Finish yet, but rather select the **Options** tab toward the bottom and proceed to the next step.



- ___d. On the **Options** tab, make sure the checkboxes are selected as shown here. Leave the default value for the qualifier alone (the wizard will fill in the appropriate value when you check the box). Now click **Finish**.

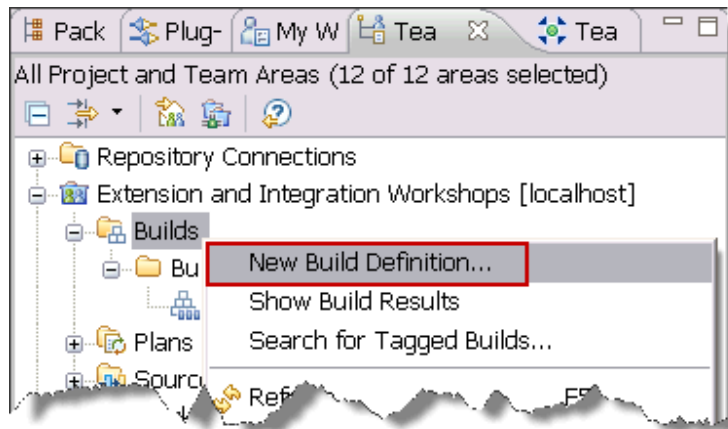


- ___2. Restart the RTC Eclipse client.
- ___a. Close your RTC Eclipse client.
- ___b. In Windows Explorer, navigate to `C:\RTC2002Dev\jazz\client\eclipse` and double click **eclipse.exe**.

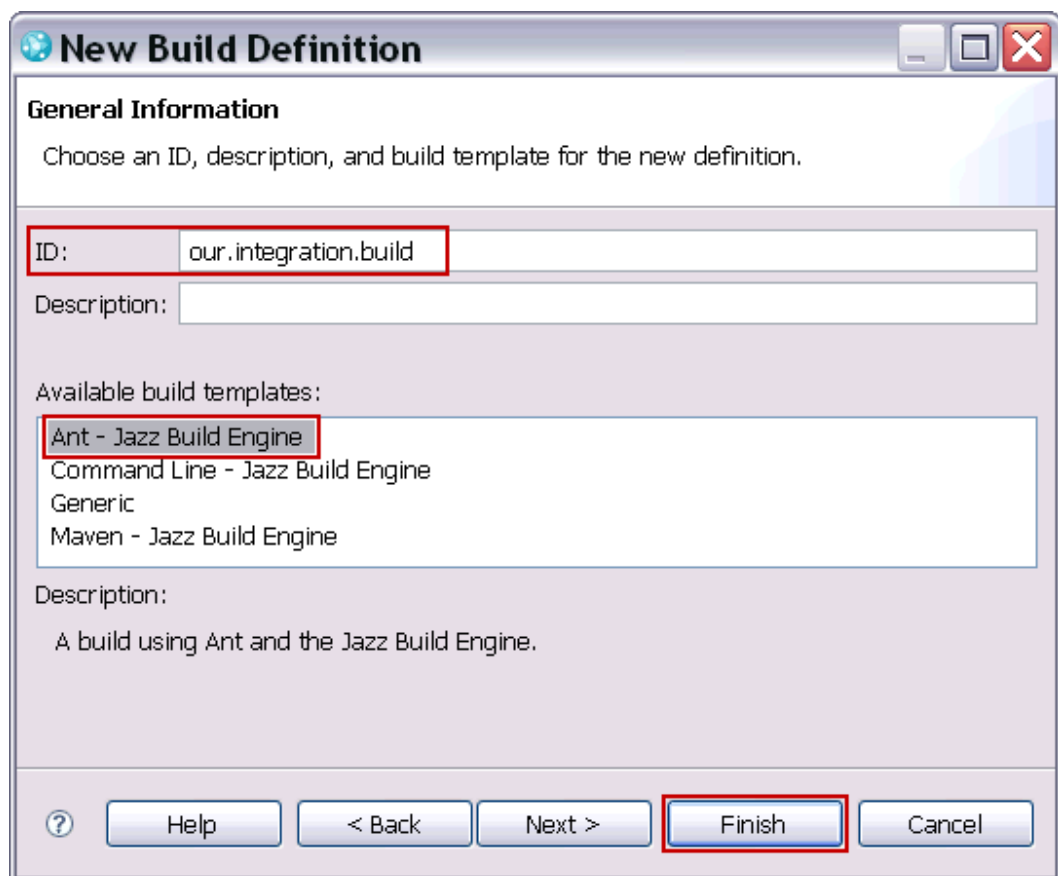
6.5 Test the Deployed Participant

- ___1. Create a dummy build definition. You just need a simple build definition to test the participant. The build does not need to run properly. The participant just needs to make requests for it.

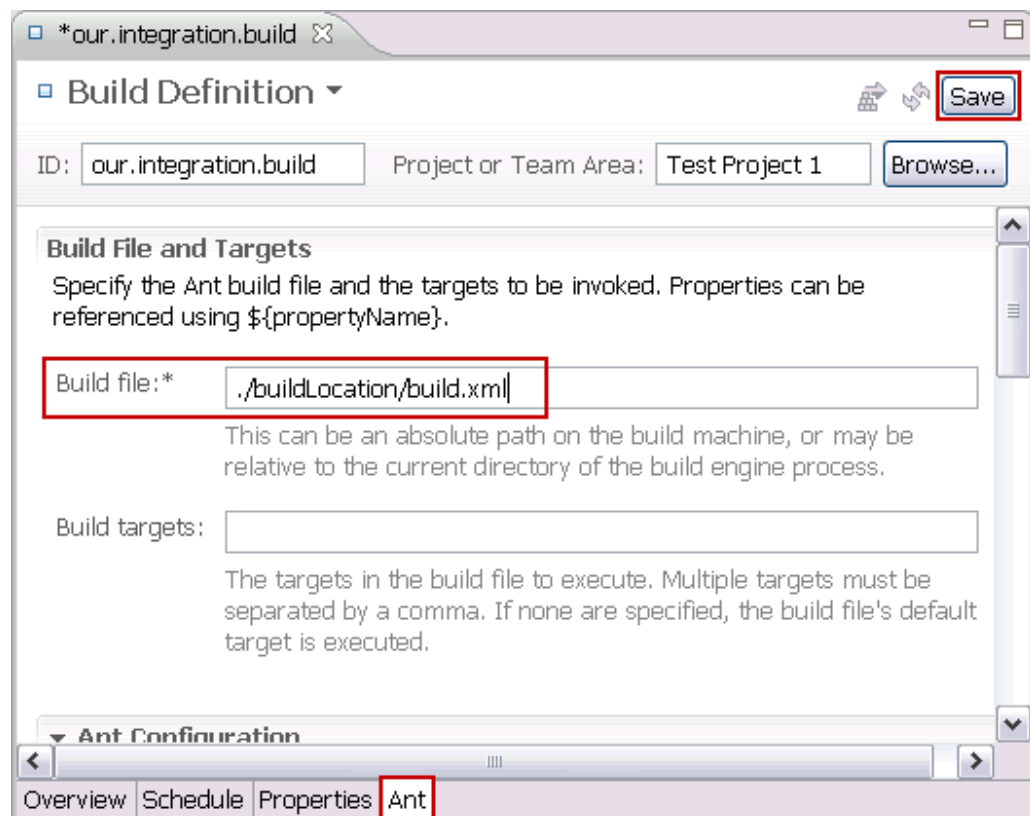
- ___a. In the **Team Artifacts** view, expand the **Extension and Integration Workshop** node, right click **Builds** and then click **New Build Definition...**



- ___b. In the **New Build Definition** wizard, make sure **Create a new build** is selected and then click **Next**. On the second page of the wizard, change the **ID** to `our.integration.build`, make sure **Ant - Jazz Build Engine** is selected and then click **Finish**.

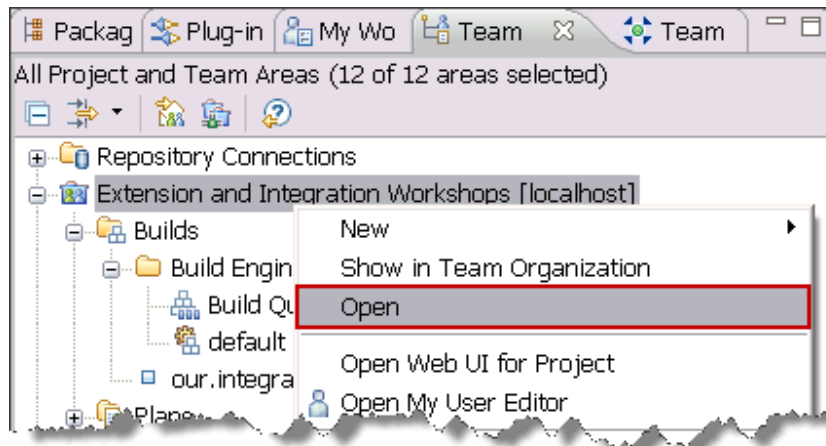


- ___c. In the build definition editor that opens, switch to the **Ant** tab, and enter a path for the **Build** file and then click **Save**. You may now close the editor. Note that the build file does not exist and any path will work for the current purpose. If you wish, you can use the path shown, `./buildLocation/build.xml`. Also note that a default build engine is created at this time and is associated with your new build definition. This actually is important. If there was no build engine for your build definition, the participant's request for a build would fail.

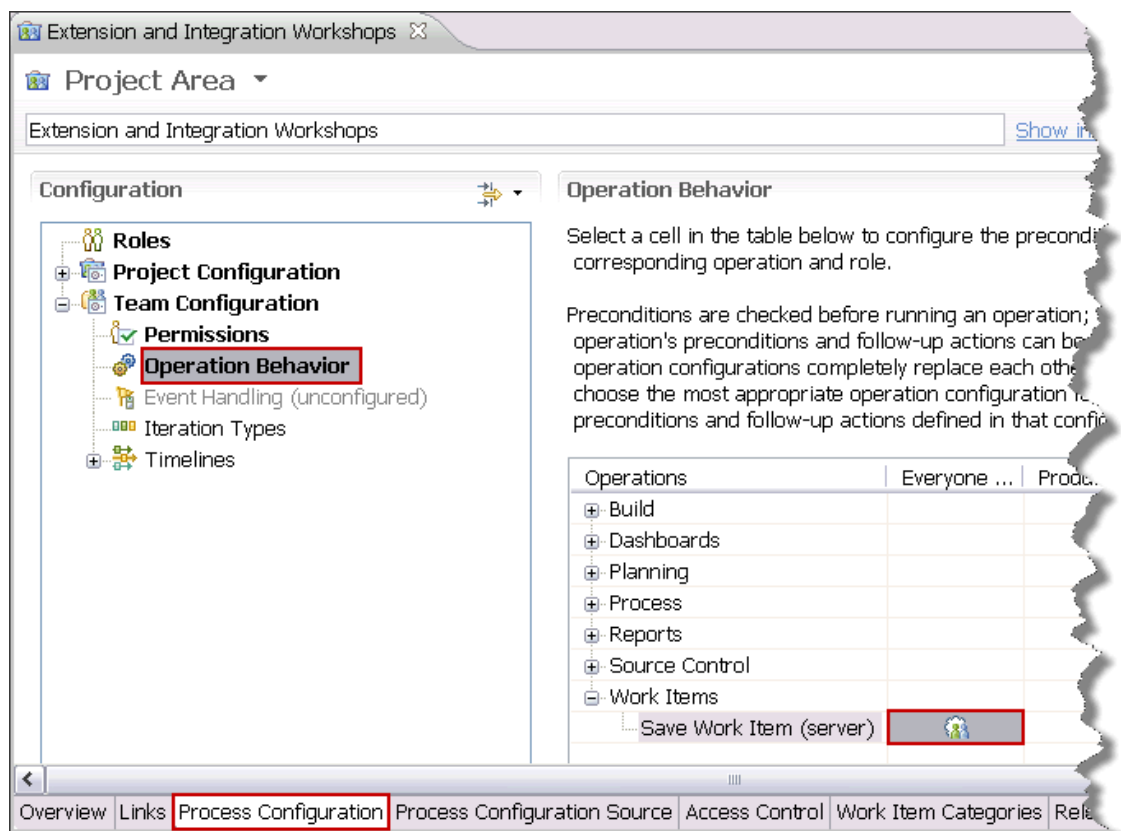


__2. Add the follow-up action to the project area.

- __a. In the **Team Artifacts** view, right click the **Extensions and Integration Workshops** project area and select **Open** from the menu.



- __b. In the project area editor, switch to the **Process Configuration** tab and then on the left, expand the **Team Configuration** tree then select **Operation Behavior**. Then, on the right, scroll down to the **Work Items > Save Work Item (server)** operation and select the **Everyone (default)** column next to it as shown here.



- ___c. Scroll down to find the **Follow-up actions** section on the right. Initially, the list will be empty. Click **Add...** then on the **Add Follow-up Actions** dialog, select **Build on State Change** (your new participant!) and click **OK**. Build on State Change will now be in the list and when it is selected, the window will look like the following image. The aspect editor is shown but needs to be filled out.

☒ Preconditions and follow-up actions are configured for this operation

☐ Final (ignore customization of this operation in child team areas)

Preconditions (6 available):

Name: Build on State Change ☐ Fail if not installed

Description:

When the specified work item type changes to the specified state, the specified build will be requested.

Work Item Trigger

Type Id:

State Id:

Build Definition

Id:

Follow-up actions (1 available):

Add... Build on State Change

Remove

Up

Down

- ___d. Fill in the **Work Item Trigger** as shown here. You may, of course, choose different values for the work item type and state, but then you will need to adjust the following steps accordingly.

Work Item Trigger

Type Id: * Story (com.ibm.team.appt.w)

State Id: * Implemented (com.ibm.tea)

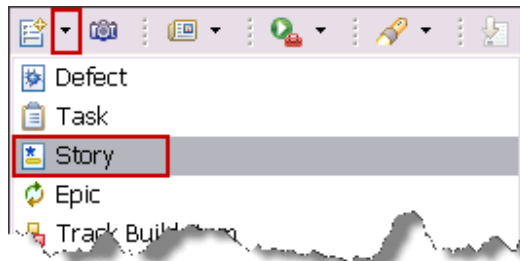
Build Definition

Id: * our.integration.build

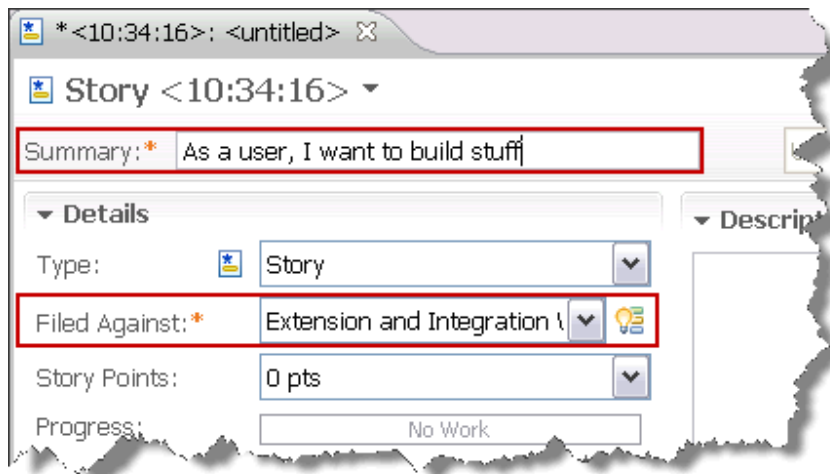
- ___e. Click Save at the top right of the editor. You may now close the project area editor and any other editors that may still be open.

___3. Create a Story and move it to the target state.

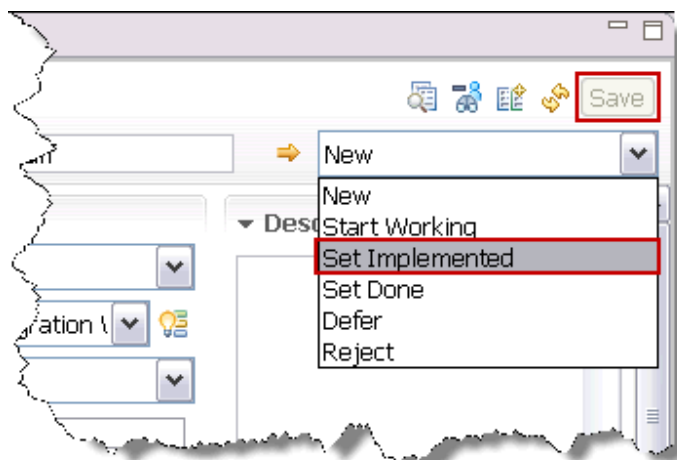
___a. Click the dropdown menu arrow next to the **New Work Item** toolbar icon and then click **Story**.



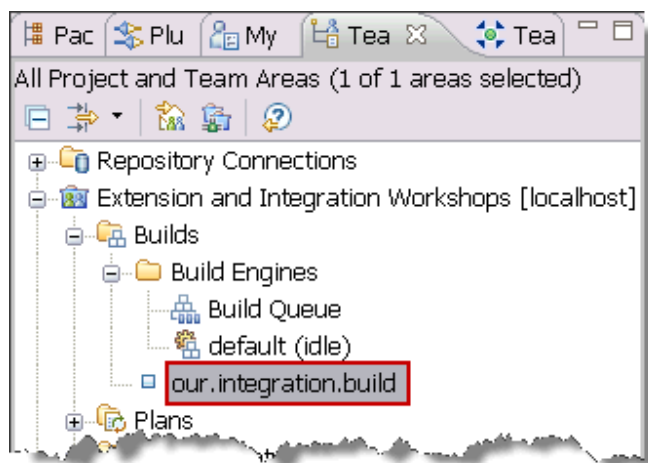
___b. In the new work item editor that opens, set the two required fields and shown here and then click **Save** in the upper right corner.



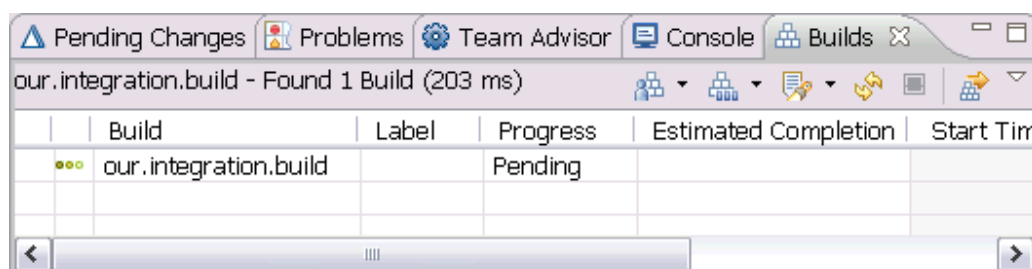
___c. At the upper right portion of the work item editor, select **Set Implemented** and then click **Save**.



- ___d. At this point, the participant has run twice (once on each save). The first one did not cause a build to be submitted, but the second did. In the **Team Artifacts** view, expand the **Builds** node as shown here and double click the **our.integration.build** build definition.



- ___e. The **Builds** view opens showing your submitted build.



6.6 Complete Development

- ___1. Deliver the new deploy component.
- ___a. If you wish, go to the **Pending Changes** view and now that you have tested the feature and update site, check-in any adjustments you have made since sharing and then deliver the added component and its content to the stream.
- ___2. The reset URL.
- ___a. Note that if you update a feature that has already been provisioned into a Jazz server and the server does not pickup the update but seems to still be running the prior version, there is a URL that can be used to force reprovisioning. For the server you have been using in this lab, the URL would be this (<https://localhost:9443/jazz/admin?internal#action=com.ibm.team.repository.admin.server.Reset>). A page will appear with a Request Server Reset button. Click that button and the next time the server is restarted, all the plug-ins will be reprovisioned.



You have completed lab 6 and the whole workshop. You have a complete work item save participant implementation and it is deployed into a real environment.

So what to do next? The next thing you would probably want to do is use this new found skill to solve a real issue at work. However, you may feel that you need more information. Perhaps you do not feel comfortable enough yet with the Eclipse plug-in model and are not sure you could create them from scratch yourself, or perhaps you want to extend RTC in a different way.

For the first issue, the place to start is with one of the many Eclipse plug-in development tutorials that can be found on the internet.

One such tutorial is at

<http://www.ibm.com/developerworks/library/os-eclipse-plugindev1/>.

Others can be found via an internet search on (without the quotes) "eclipse plugin development tutorial".



For the second issue, you now have an RTC extensions development environment that can support the various scenarios described in the RTC SDK at jazz.net

(<http://jazz.net/wiki/bin/view/Main/RtcSdk20>). Getting this set up properly is often the toughest part. So, look through the RTC SDK scenarios and you will probably find the starting point and an example for what you need to do. If not, use the Extending Team Concert forum (<http://jazz.net/forums/viewforum.php?f=2>) at jazz.net to ask questions about where to start for your specific problem. Be sure to be as specific as possible and do not assume that those that answer have also been through this workshop.

Appendix A. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have

been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental. All references to fictitious companies or individuals are used for illustration purposes only.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Appendix B. Trademarks and copyrights

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM

IBM Logo

Rational

Jazz

Adobe, Acrobat, Portable Document Format (PDF), and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Apache, Apache Tomcat and Tomcat are trademarks of The Apache Software Foundation.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both. See Java Guidelines

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product and service names may be trademarks or service marks of others.

NOTES

[illegible]

NOTES

[illegible]



© Copyright IBM Corporation 2010. All rights reserved.

The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. This information is based on current IBM product plans and strategy, which are subject to change by IBM without notice. Product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way.

IBM, the IBM logo, and other IBM products and services are trademarks of the International Business Machines Corporation, in the United States, other countries or both. Other company, product, or service names may be trademarks or service marks of others.

