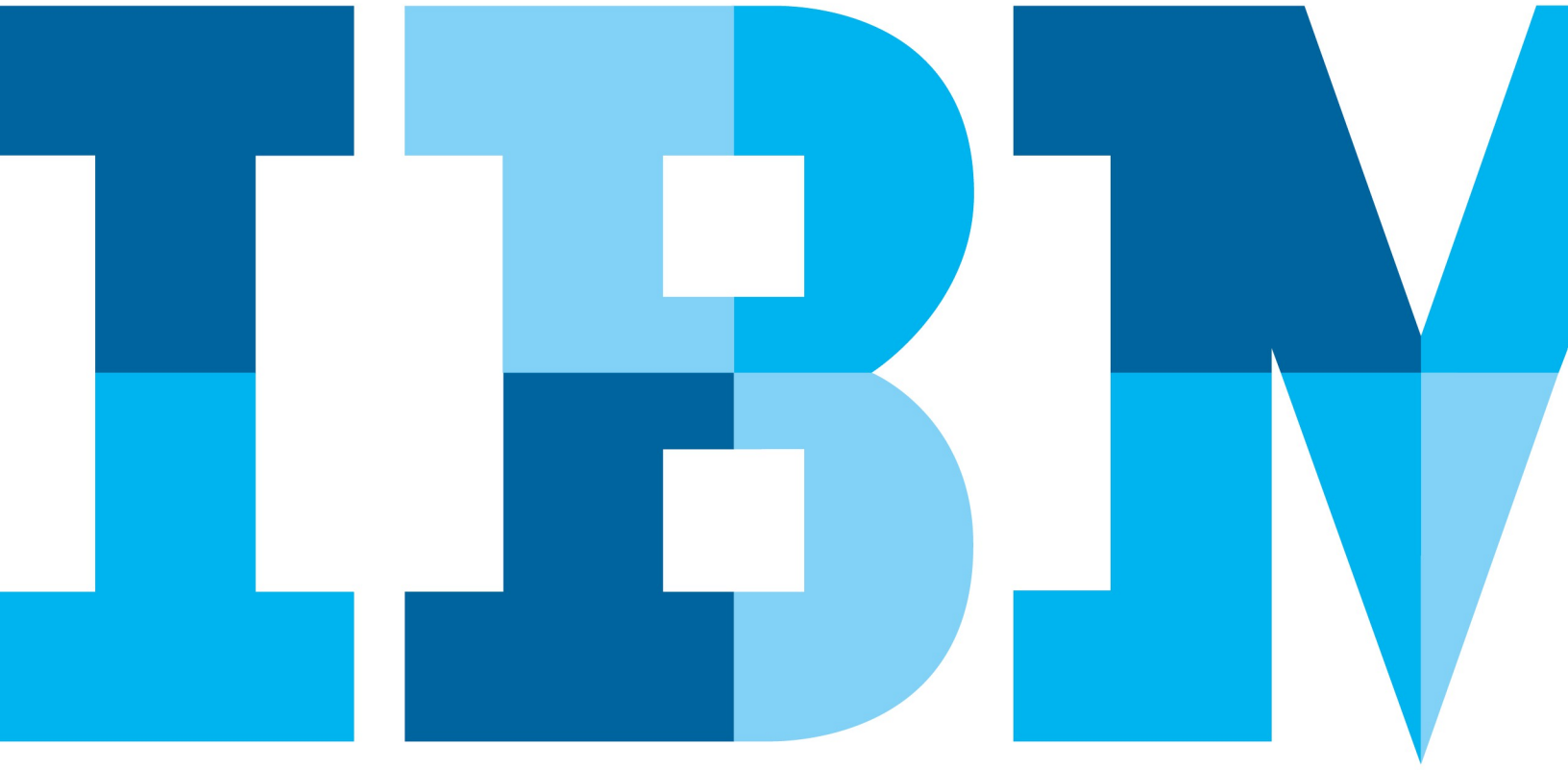
IBM Software
Rational

IBM Rational Team Concert 4.x Extensibility

Lab Exercises



IBM

Contents

Overview

You are a member of the team managing the deployment and administration of your company's software development tools infrastructure, including Rational Team Concert (RTC). Among other things, one of your assignments is to create extensions to the tools as required by the software development teams.

Now you have been assigned to create a new work item save operation participant (or follow-up action). If the participant is configured for a project and a Story is changed to the Implemented state, one of the project's builds will be run. If the build can not be started, the work item save is stopped.

In this workshop, you will setup your development environment for creating RTC extensions and then implement this particular operation participant.

Introduction

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 4.x on Windows®. Please adjust accordingly for different operating systems (primarily the RTC Eclipse client download and the file paths) and RTC versions (downloads).

Along with this lab document(s), you should have received or downloaded the file WorkshopSetup-V4-YYYYMMMDD.zip. This file contains a small RTC Plain Java Client Libraries tool that will be used to create a project and populate it with data.

Icons

The following symbols appear in this document at places where additional guidance is available.

Icon	Purpose	Explanation
	Important!	This symbol calls attention to a particular step or command. For example, it might alert you to type a command carefully because it is case sensitive.
	Information	This symbol indicates information that might not be necessary to complete a step, but is helpful or good to know.
	Trouble-shooting	This symbol indicates that you can fix a specific problem by completing the associated troubleshooting information.

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 4.x on Windows®. Please adjust accordingly for different operating systems (primarily the RTC Eclipse client download and the file paths) and RTC versions (downloads).

The Workshop should run with 4.0 and higher versions of RTC. In case you have trouble, ask in the Jazz.net Forum for help.

1.1 Download and Unzip the Required Files from jazz.net

Important!
Please note, the install description for the RTC Server below is for the Web Installer. Please see the information below for other install options.

Other Install Options

You can use the other options to install RTC as well.

1. You can install from an IBM Installation manager repository similar to the web install below
2. You can use the RTC Plain Zip install file; use 7Zip and unzip to the folder **C:\RTC40Dev\installs\JazzTeamServer** and you are done

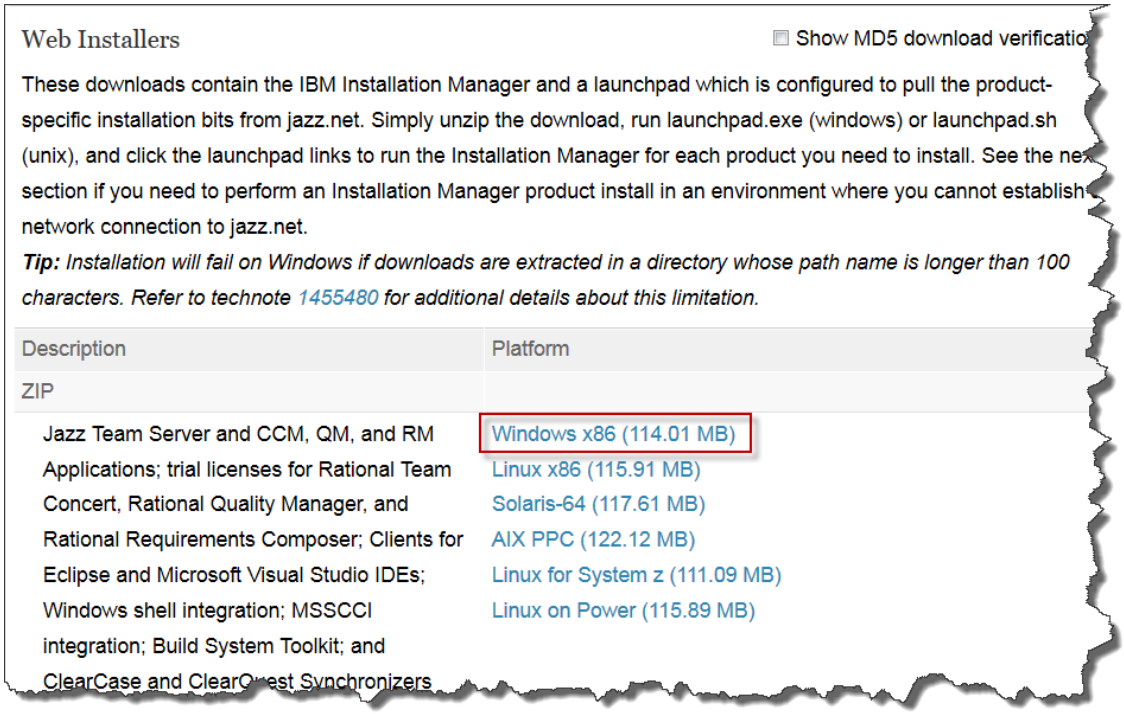
Separate RTC versions using different folder names

Use a different folder name for each RTC version, e.g. **RTC403Dev** for RTC 4.0.3 to be able to maintain different versions. Replace the folder name used in the workshop instructions if following this approach.

__1. Download the product installation files.

__a. Go to the RTC all downloads page for your version of RTC. As example for 4.0 at https://jazz.net/downloads/rational-team-concert/releases/4.0?p=allDownloads. The file sizes will vary from what is shown next.

__b. Scroll down to the **Web Installers** section and download the highlighted file. This will be used to install the server (but not the client). You will need a jazz.net id. There are other options for download and install if you prefer. This workbook will use this method.



Web Installers

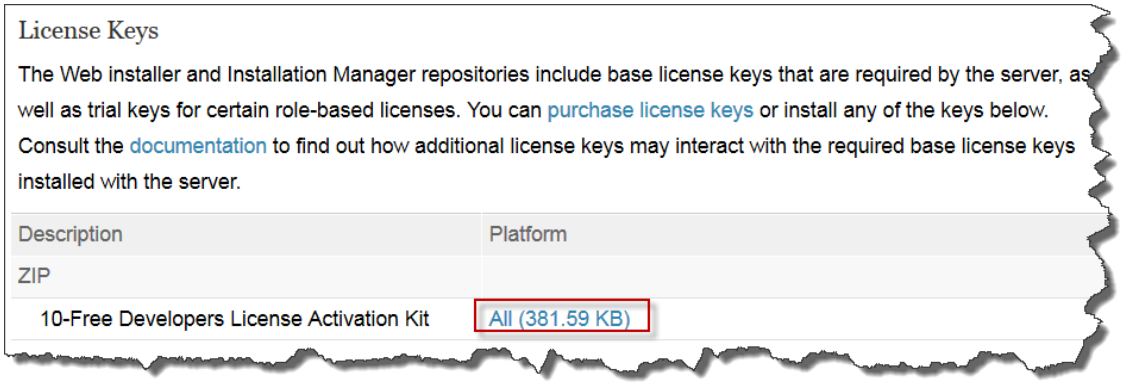
Show MD5 download verificatio

These downloads contain the IBM Installation Manager and a launchpad which is configured to pull the product-specific installation bits from jazz.net. Simply unzip the download, run launchpad.exe (windows) or launchpad.sh (unix), and click the launchpad links to run the Installation Manager for each product you need to install. See the nex section if you need to perform an Installation Manager product install in an environment where you cannot establish network connection to jazz.net.

Tip: *Installation will fail on Windows if downloads are extracted in a directory whose path name is longer than 100 characters. Refer to technote 1455480 for additional details about this limitation.*

Description	Platform
ZIP	
Jazz Team Server and CCM, QM, and RM Applications; trial licenses for Rational Team Concert, Rational Quality Manager, and Rational Requirements Composer; Clients for Eclipse and Microsoft Visual Studio IDEs; Windows shell integration; MSSCCI integration; Build System Toolkit; and ClearCase and ClearQuest Synchronizers	Windows x86 (114.01 MB) Linux x86 (115.91 MB) Solaris-64 (117.61 MB) AIX PPC (122.12 MB) Linux for System z (111.09 MB) Linux on Power (115.89 MB)

__c. Scroll down to the **License Keys** section and download the highlighted file.

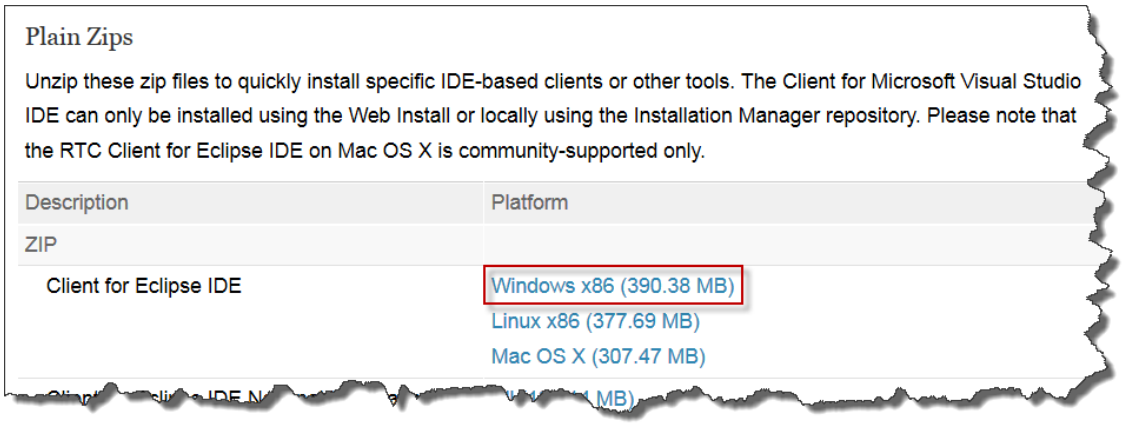


License Keys

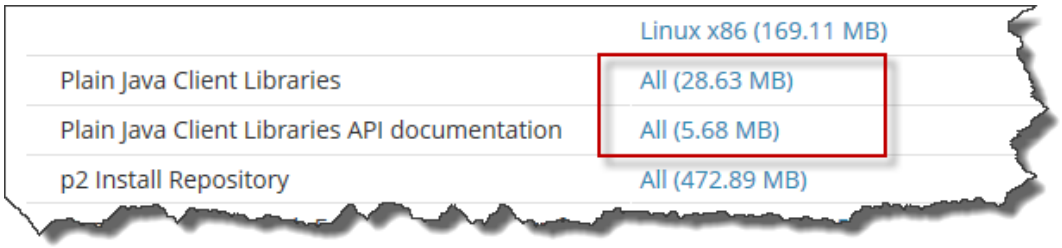
The Web installer and Installation Manager repositories include base license keys that are required by the server, as well as trial keys for certain role-based licenses. You can purchase license keys or install any of the keys below. Consult the documentation to find out how additional license keys may interact with the required base license keys installed with the server.

Description	Platform
ZIP	
10-Free Developers License Activation Kit	All (381.59 KB)

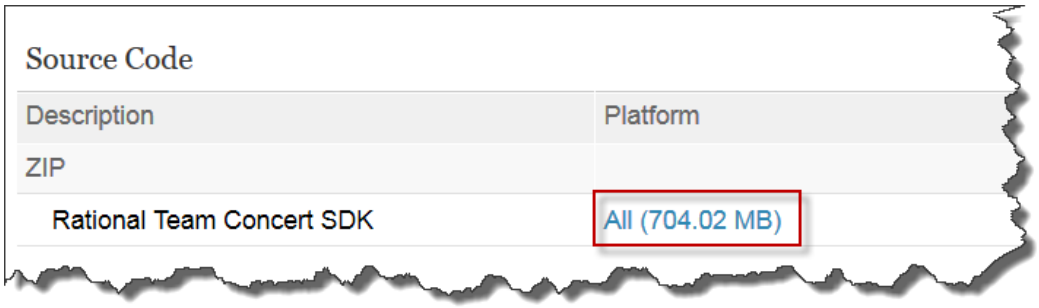
__d. Scroll down to the **Plain Zips** section and download the highlighted client zip file.



Also download the highlighted zip files for the Plain Java Client Libraries.



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

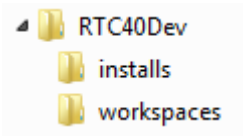


__e. Go to the feature based launches wiki page at https://jazz.net/wiki/bin/view/Main/FeatureBasedLaunches.

__f. Download the attached `com.ibm.team.dev.launch_0.3.3.201004231417.jar` file. Make sure the file name is correct and has no other extension. If the download changed or added a different file name extension, rename the file.

__2. Set up a directory structure to contain your extensions development and test environment.

__a. Many people like to isolate their extensions development environment from their normal application development environment. This helps avoid blocking your application development work (which may be your day job) by a buggy extension you have created and deployed (during your extra time). This workshop will assume the following folder structure on the C: drive.



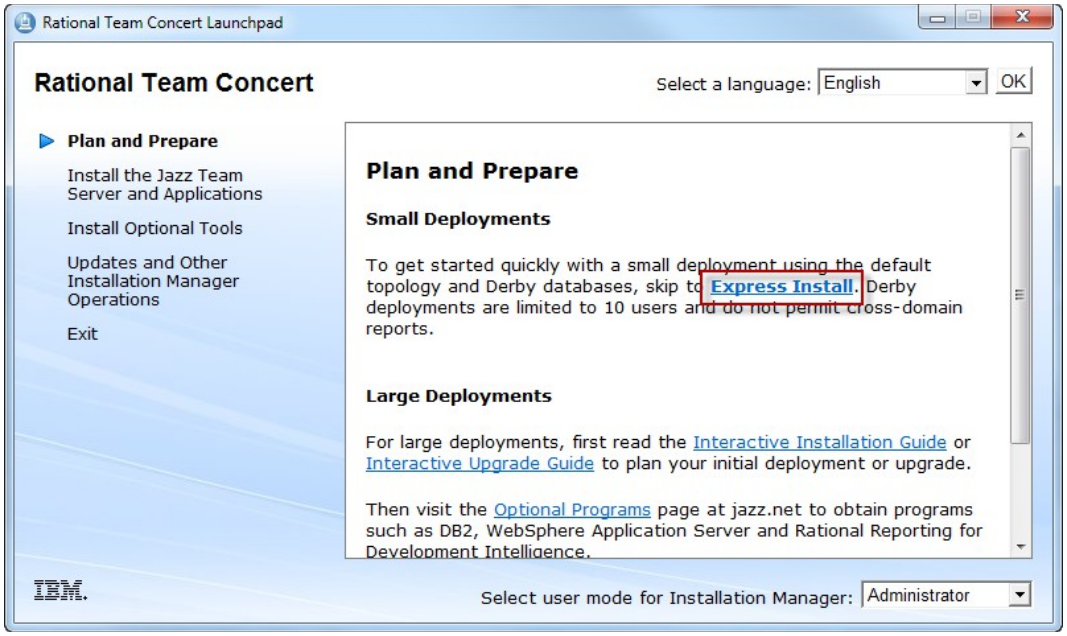
The **RTC40Dev** root folder will contain all your work for this workshop. The **installs** folder will be the target of the product installations you are about to perform. The **workspaces** folder will contain your Eclipse workspace(s) and other related folders.

__3. Install the RTC Eclipse client and a test server.

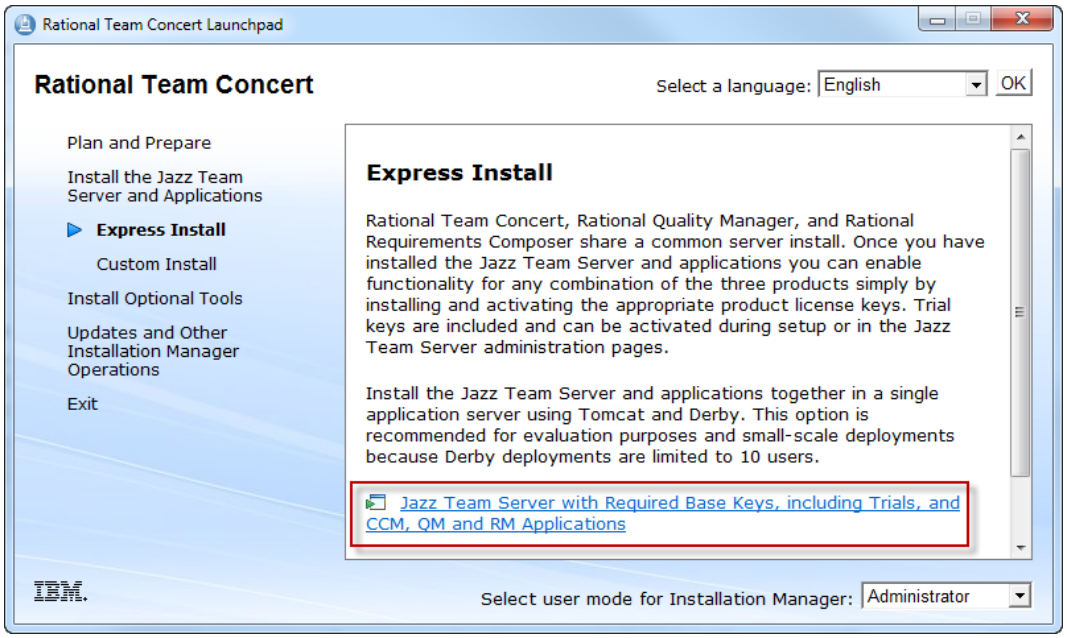
__a. Unzip the **Web Installer** download into a temporary directory and run launchpad.exe.

__b. Install the **Jazz Team Server and CCM Application** to `C:\RTC40Dev\installs\JazzTeamServer`.

__c. Select Express Install



__d. on the next screen select **Jazz Team Server with Required Base Keys, including Trials, and CCM, QM and RM Applications**



__e. Provide the required password for administrative access and your Jazz.net user name and password.

__f. If you do not already have Installation Manager installed, it will be installed at this time. After the install completes you can exit Installation Manager and the Launchpad.

During the installation, you will need to change some items from their defaults. All the other default values are fine; in particular, be sure to install both the “Jazz Team Server and CCM Application” and the “Required Base License Keys, Including Trials....” installation packages. You can remove the applications Requirements Management and Quality Management, this workshop focuses on the Change and Configuration Management Application

If the product is already installed you will be prompted select to continue installation.

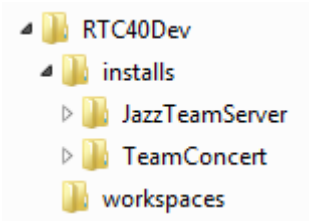
__g. If you are on Windows 7, change the **Shared Resources Directory** to be outside the **Program Files** or **Program Files (x86)** directories. These directories are virtualized and if any part of the server is installed into a virtualized directory, the server would have to be run as an administrator. Note that even if you are logged into Windows 7 as an administrator, the default when starting an application is to not run it as an administrator. You can put it anywhere you want, for example into `C:\IBM\IBMIMShared` as long as it is not virtualized.

__i. Change the **Installation Directory** to `C:\RTC40Dev\installs\JazzTeamServer`.

__ii. Review the installation packages. You may de-select Requirements Management and Quality Management as these are not needed for the workshop.

__h. Unzip the **Client for Eclipse IDE** zip file to `C:\RTC40Dev\installs\TeamConcert`. Do not use Web Installer or any other Installation Manger method to install the client. You need a plain Eclipse layout for this workshop and not a layout that optimizes disk space via Installation Manger's area for shared features and plugins.

__i. Your `C:\RTC40Dev` folder will look pretty standard at this point. Much like setting up a sandbox or demo environment.



__4. Add the feature based launches capability to the RTC Eclipse client.

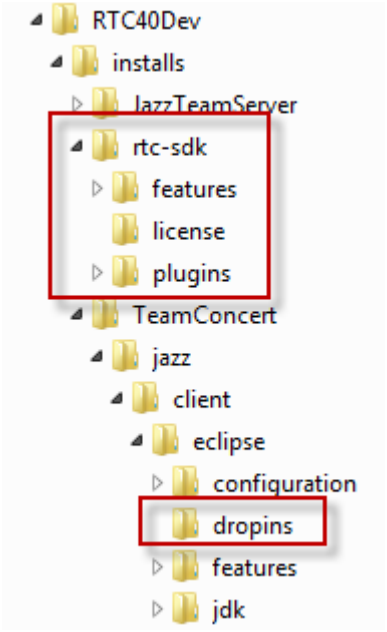
__a. Copy the feature based launches download file `com.ibm.team.dev.launch_0.3.3.201004231417.jar` into the folder `C:\RTC40Dev\installs\TeamConcert\jazz\client\eclipse\dropins`. The dropins folder might not exist and then needs to be created by you.

__b. Note that some users on Linux have reported that the file permissions on the jar placed in the dropins folder are set to 755 and that the feature based launches would not show up in the RTC Eclipse client until the permissions were changed to 644.

__5. Unzip the development time files.

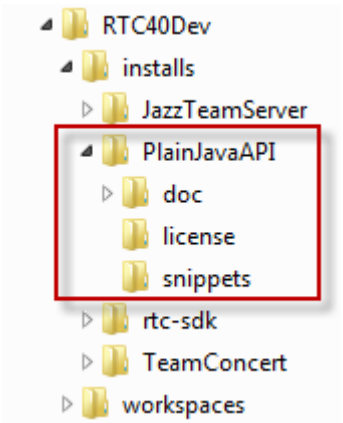
__a. Unzip the RTC SDK zip file into `C:\RTC40Dev\installs\rtc-sdk`. This zip file has path lengths longer than 250 characters and may cause trouble for some extractor tools on Windows. One zip extractor tool that works is 7Zip. You or your extraction tool will need to create the rtc-sdk folder. It is not contained in the zip file.

__b. Your `C:\RTC40Dev` folder will now look a bit different.



__c. Unzip the RTC Plain Java Client Libraries files into `C:\RTC40Dev\installs\PlainJavaAPI`.

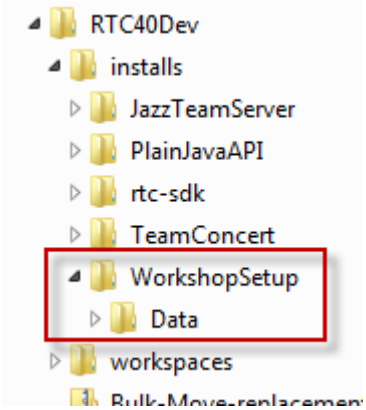
__d. Unzip the RTC Plain Java Client Libraries API Documentation files also into `C:\RTC40Dev\installs\PlainJavaAPI`. This result in this final folder structure.



__6. Install the Workshop Setup tool.

__a. Along with this lab document(s), you should have received or downloaded the file WorkshopSetup-V4-YYYYMMDD.zip.

__b. Unzip this file to `C:\RTC40Dev\installs\`. Your `installs` folder should now finally look like below.



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.

__7. Start the Eclipse RTC client.

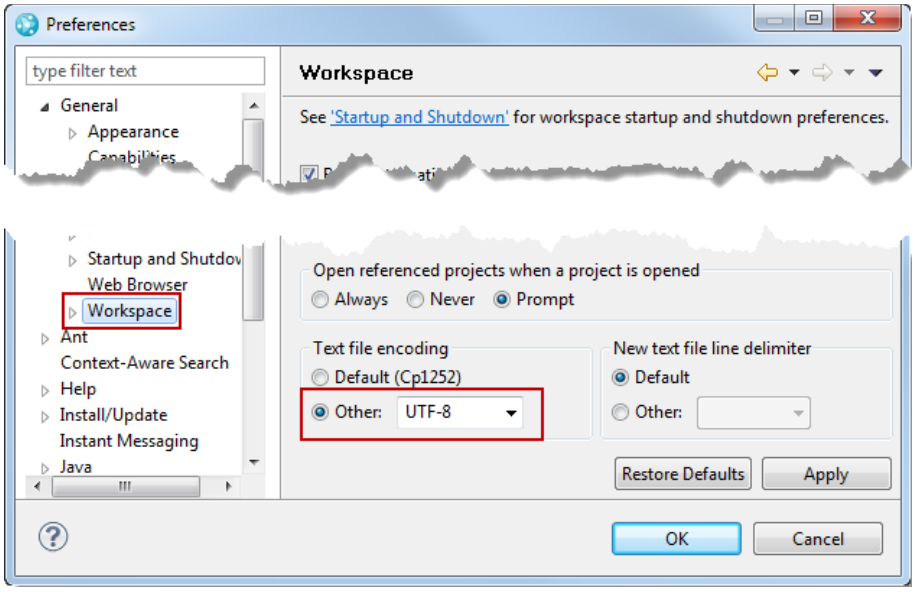
__a. Start the RTC Eclipse client (`C:\RTC40Dev\installs\TeamConcert\jazz\client\eclipse\eclipse.exe`).

__b. When prompted, select an Eclipse workspace. These instructions will use `C:\RTC40Dev\workspaces\Dev1\WS`. Note that it is “\Dev1\WS” and not “Dev1WS”. Either would work, but by default when you launch a runtime or debug session the Eclipse workspace for the launched process is created as a sibling to your workspace folder. By using the “\Dev1\WS” technique, this runtime workspace folder is created as a peer to “WS” inside the “Dev1” folder. The makes it easier to have other isolated development workspaces such as “ C:\RTC40Dev\workspaces\Dev2\WS” without any collisions between launches that have the same name. Alternatively, you can specify a launch's workspace location, but isolating them using this technique is easier to remember.

__c. Minimize the **Welcome** via this () button near the top of the window.

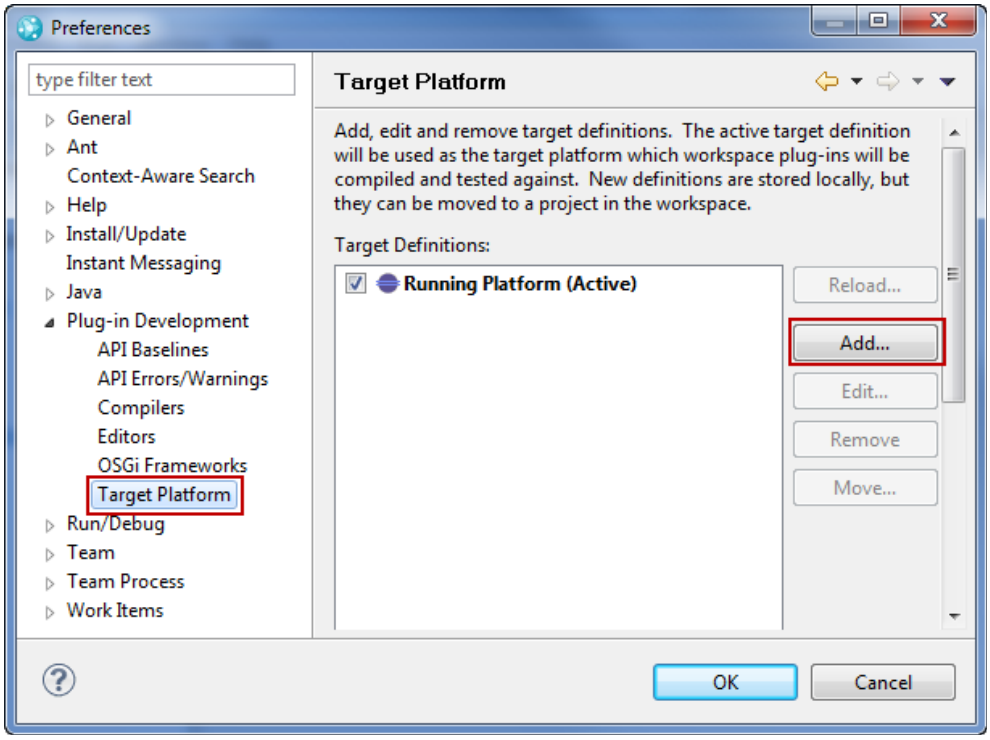
__8. Switch Text file encoding to UTF 8

__a. From the menu bar, select **Window > Preferences**. In the **Preferences** dialog, select **General > Workspace**. In the **Text file encoding** section select **Other** and select the encoding **UTF-8**. This is important to be able to run the launches for debugging.

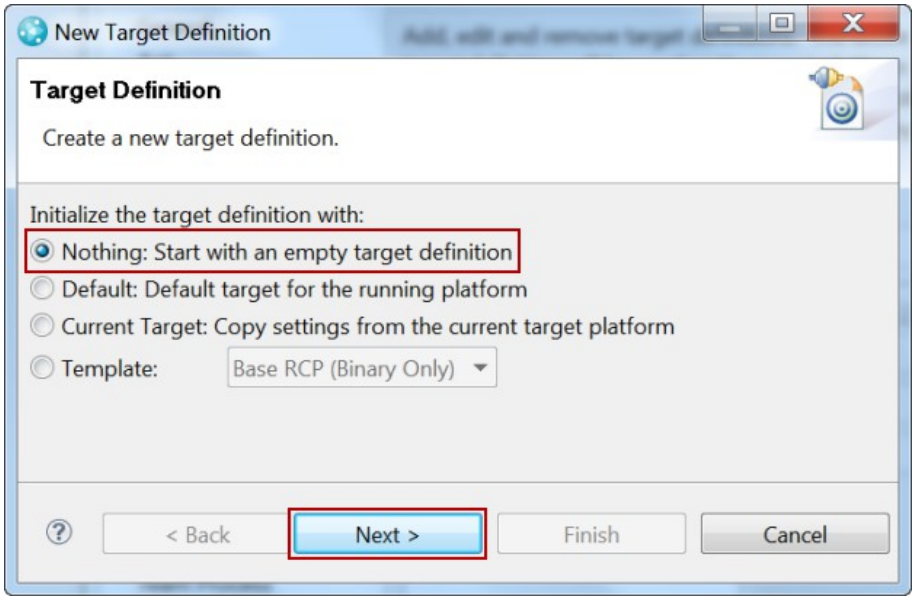


__9. Create a new target platform.

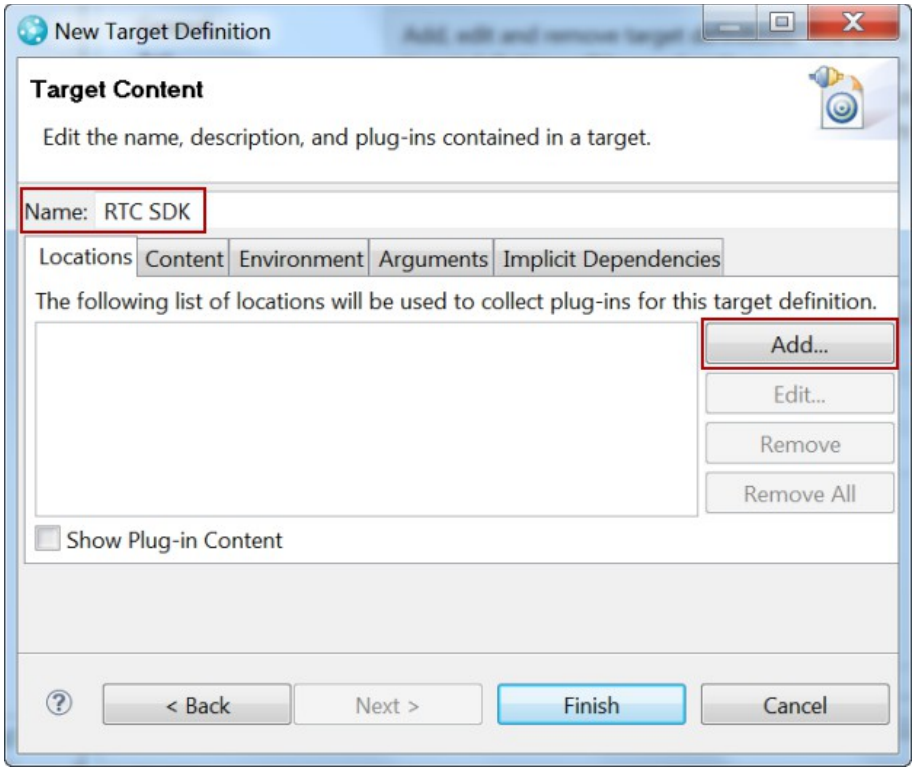
__a. From the menu bar, select **Window > Preferences**. In the **Preferences** dialog, select **Plug-in Development > Target Platform** and then click **Add...**



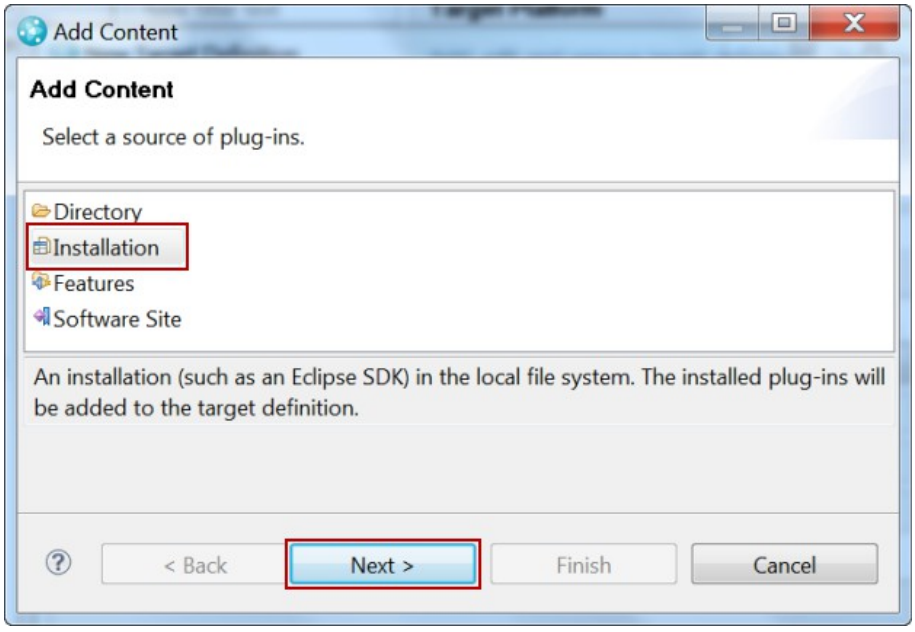
__b. In the **New Target Definition** wizard, select **Nothing: Start with an empty target definition** and then click **Next**.



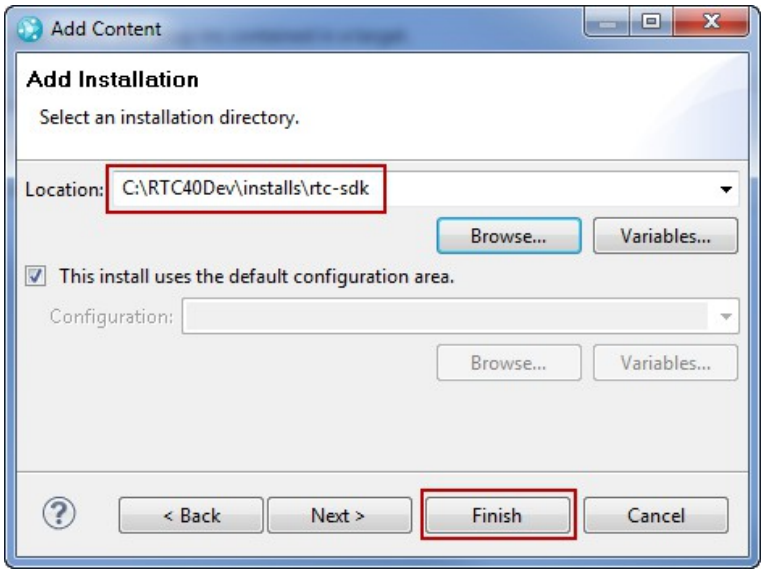
__c. On the second page of the wizard, enter *RTC SDK* as the **Name** and click **Add...**



__d. In the **Add Content** wizard, select **Installation** and then click **Next**.

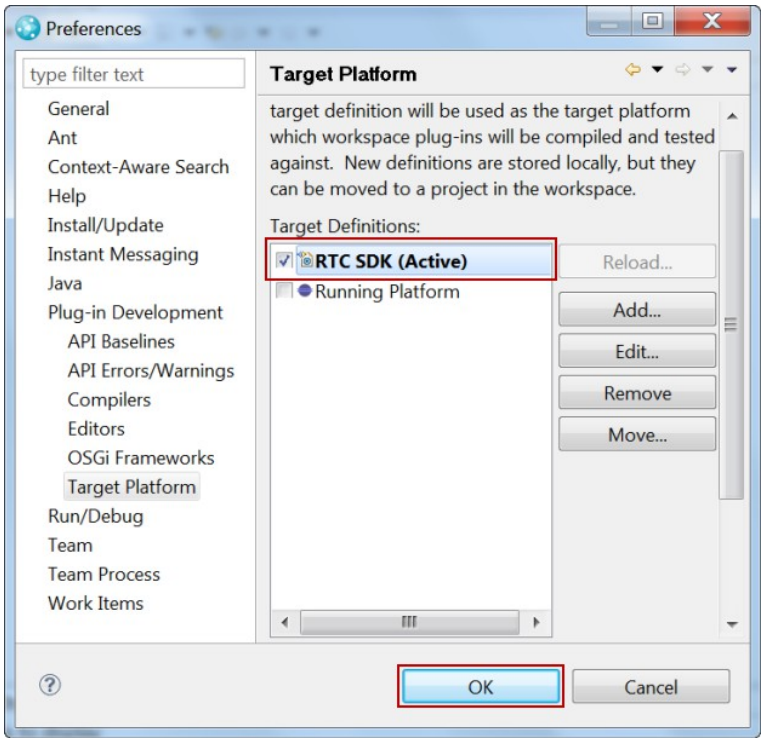


__e. On the second page of the wizard, enter `C:\RTC40Dev\installs\rtc-sdk` as the **Location** and then click **Finish**.



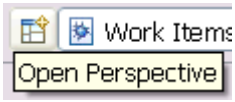
__f. After the operation completes, click **Finish** in the **New Target Definition** wizard.

__g. Back on the **Preferences** dialog, select the new **Target Definition** and then click **OK**.

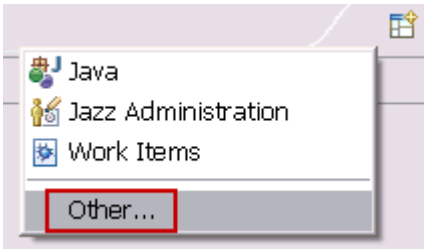


__10. Open the Plug-in Development perspective.

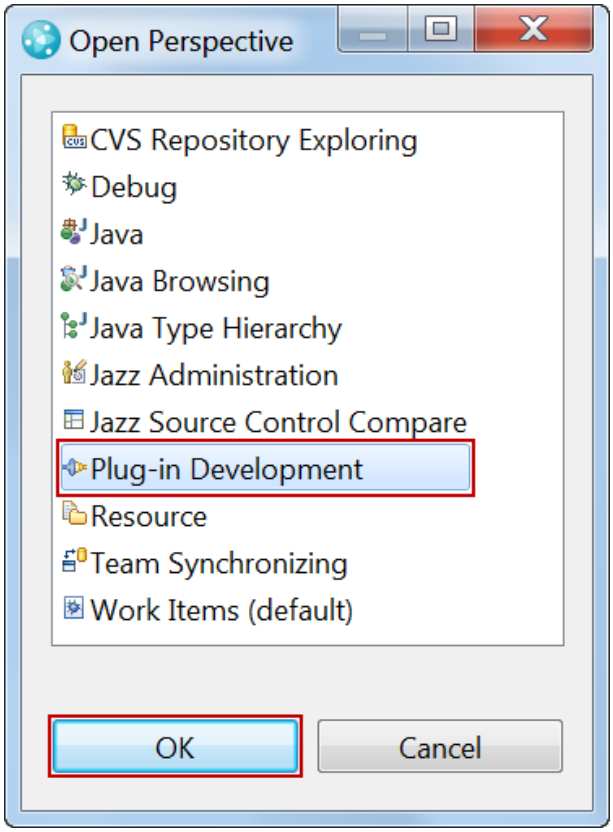
__a. In the toolbar toward the right, click the **Open Perspective** button.



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

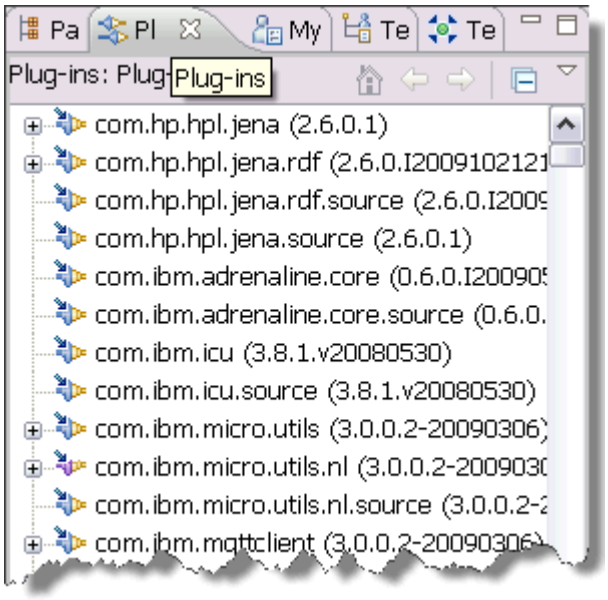


__c. In the **Open Perspective** dialog, select **Plug-in Development** and then click **OK**.

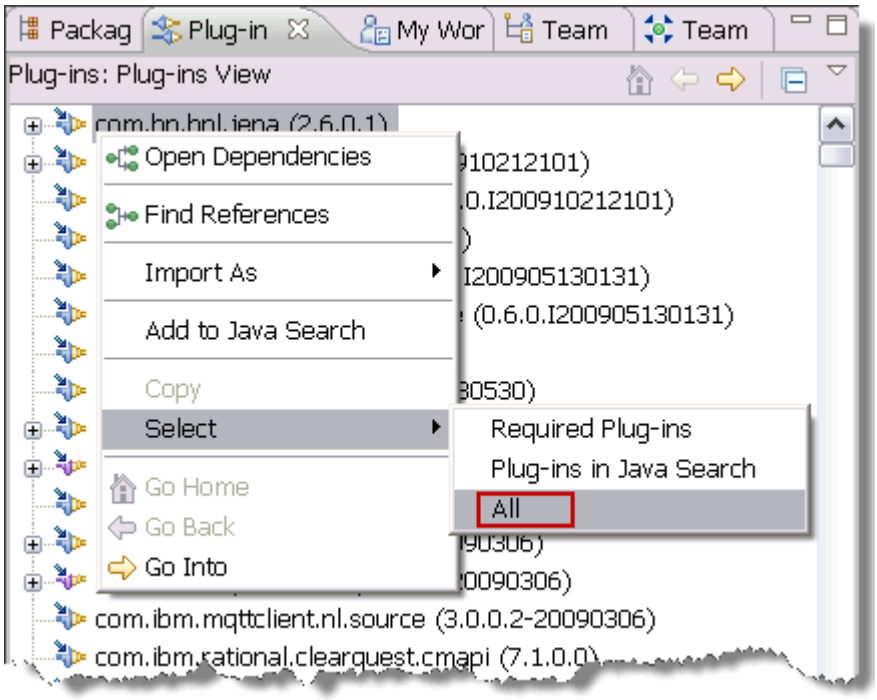


__11. Add RTC source code to Java search.

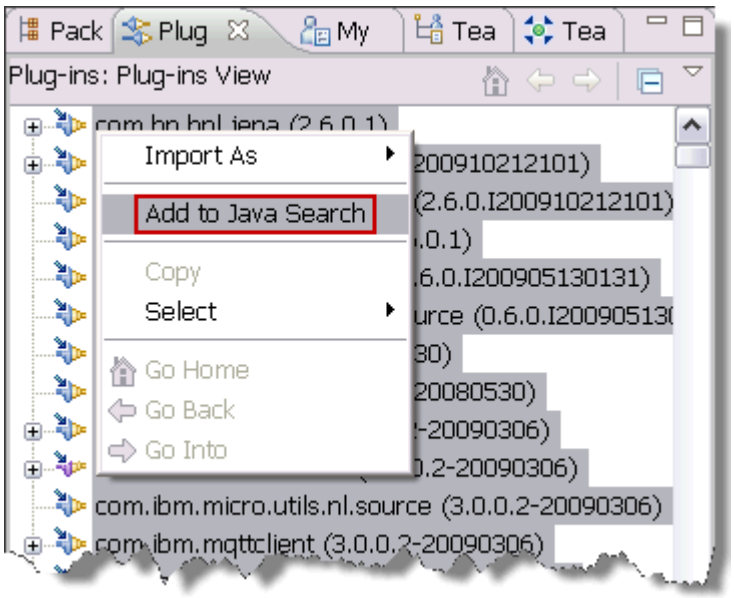
__a. On the left, select the **Plug-ins** view.



__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 the RTC Tomcat Server

You will now setup the RTC Tomcat server. This server will be used in two roles:

1. This RTC is used as your RTC Development server that provides you with streams, SCM data and a capability to version your code
2. This RTC is used as a test server for the final deployment of the extension you will create in this workshop

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

In this section you will also import the workshop repositories into this server.

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

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.

__12. Setup to run the server in debug mode.

__a. Open Windows Explorer and navigate to `C:\RTC40Dev\installs\JazzTeamServer\server`.

__b. Open the **server.startup.bat** file with either Notepad or Wordpad.

__c. Find the following line that starts with the following. It is near the bottom of the file.

```
set JAVA_OPTS=%JAVA_OPTS% -Djava.awt.headless=true
```

__d. After that line add these lines.

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

__e. Be sure to keep all the other options. 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.

__13. Complete setup of the server.

The description and repositories for this workshop for simplicity reason provide data for JTS and all applications (CCM, QM, RM).

- The extensions are only for the CCM application.

- During the setup you can skip configuring the data warehouse for the applications to save some time.

__a. Open a Windows Explorer and navigate to `C:\RTC40Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file (the same file you just edited).

__b. After the server has finished starting (the "INFO: Server startup in nnnnn ms" message is displayed in the console), start your browser and enter the URL https://localhost:9443/jts/setup.

__c. Login with `ADMIN` as both **User ID** and **Password**.

__d. Use the **Express Setup** option.

__i. Click Next.

__ii. On the **Configure Public URI** page set the **Public URI Root** to https://localhost:9443/jts

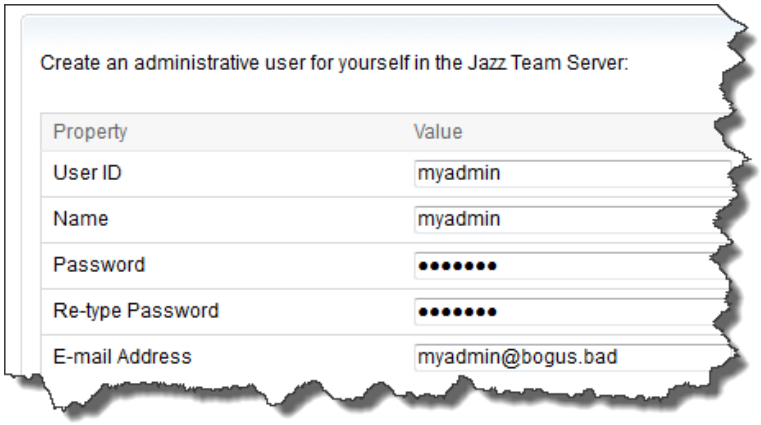
- There will be a warning when using this value, but this is not a real server and only needs to be accessible by you from your machine.

Test the connection, accept the warning in step 2 and click next.

__iii. On the **Create Administrative User** page create a new administrator user

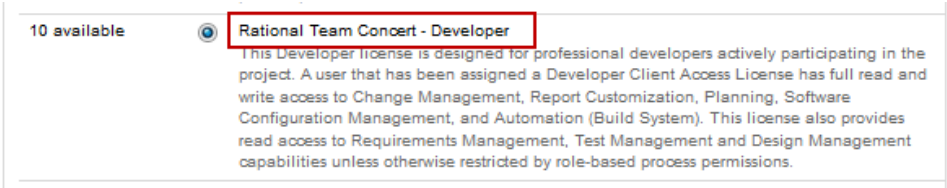
- As User ID, name, password and re-type password enter **myadmin**
- As e-mail enter myadmin@bogus.bad

Your screen should look as follows. Click next.



__iv. Wait for the Express Setup to finish. Click Next.

__v. On the **Assign Licenses** page make sure to Activate a RTC Developer trial license, if it is not already activated and assign a license to myadmin. Then click Finish.



__vi. There is no need to create a lifecycle project.

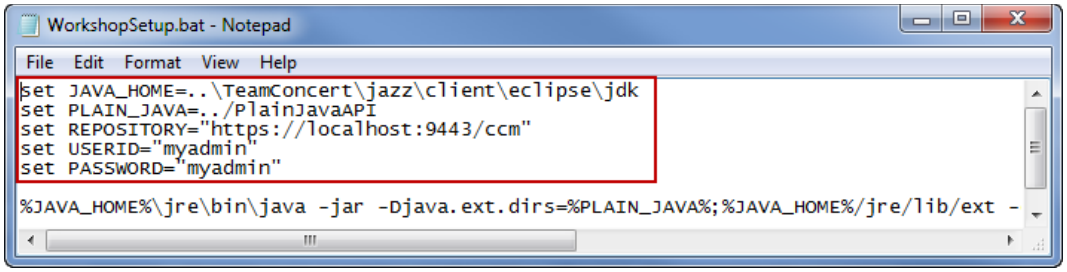
__14. You will now setup the workshop repository.

This step creates a project and uploads data into the Jazz SCM system of the project, that you will use later in the workshop.

__a. Browse to the folder `C:\RTC40Dev\installs\WorkshopSetup`. Make sure the folder exists and contains a file named `WorkshopSetup.bat`.

> **Other Operating Systems**
>
> A file WorkshopSetup.sh file for Unix operating systems is shipped with the zip file. You might have to chmod the file to make it executable.

__b. Open the file and review its content. It should look similar to below.



If you followed the instructions above, you should be able to run the WorkshopSetup without issues. The file sets required information to run the setup such as the JAVA_HOME folder of a JDK, where to find the Plain Java Client Libraries and the login information for the repository.

By default the JAVA_HOME file points to the JDK shipped with the RTC Eclipse client, if at all possible use this setting. If not, make sure JAVA_HOME is set to a **Java Development Kit** version 1.6 or higher.

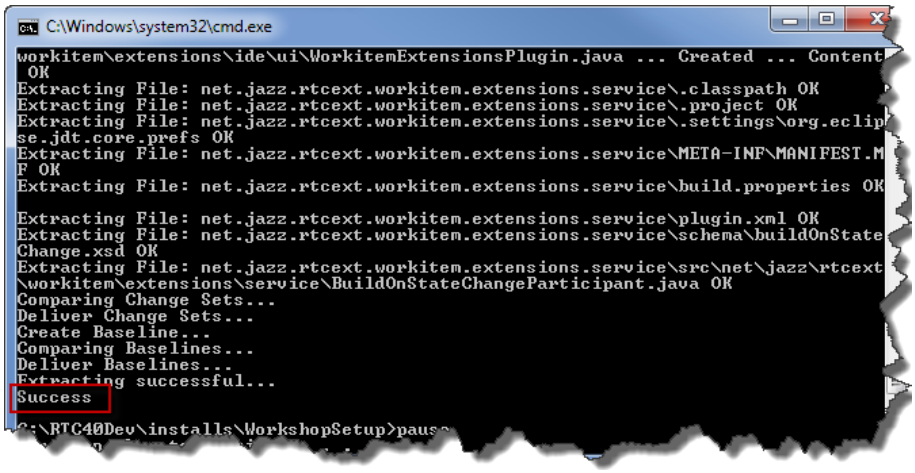
> **JDK Required**
>
> The file to launch the workshop setup assumes a **JDK** (Java Development Kit). You can not use a JRE. A compatible JDK is shipped with the Eclipse Client. Make sure to have a Java with a **lib/ext** folder.

If your setup is different, edit `WorkshopSetup.bat` to match your environment. Dependent on your environment you can choose other options to set the environment variables in the file. If you change the file save the change before running it.

Make sure to the paths match your setup. E.g. if you use a different folder for the workshop, enter the correct path.

__c. Run `C:\RTC40Dev\installs\WorkshopSetup\WorkshopSetup.bat`.

Make sure the setup is executed and shows a success and close the shell.



In case of errors, carefully read the error message, check the paths, especially to your Java JDK and make sure it is available.

Troubleshooting

Check that you are using a JDK, check the paths.

If needed replace the JAVA_HOME statement by your absolute paths and make sure your paths actually point to the required folders.

Make sure you have /lib/ext folder.

Make sure the Plain Java Client Libraries are reachable in the folder you specified.

If this step does not perform successfully, you can not follow the labs. You can however manually import the configuration files and the Lab code from the subfolders in `C:\RTC40Dev\installs\WorkshopSetup\data`.

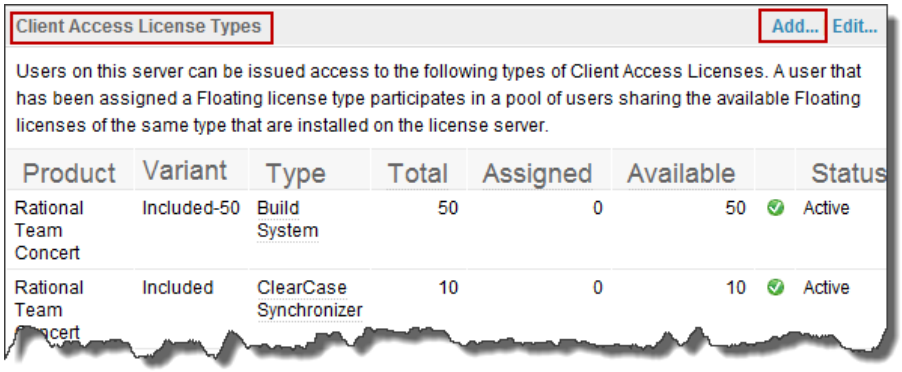
__15. Import the 10 Free Developer CALs. The license assignments in the repository will be preserved (myadmin has a Developer CAL).

__a. Locate the `RTC-Developer-10-C-License-4.0.zip` file and remember where you placed this file. You will next upload it to your server.

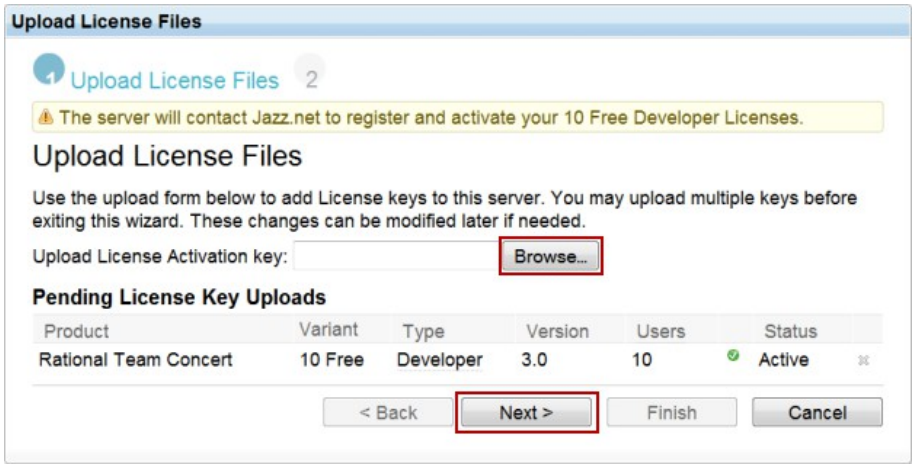
__b. Open or return to your browser and open this URL: https://localhost:9443/jts/admin#action=com.ibm.team.repository.admin.manageLicenses

__c. When prompted, enter `myadmin` for both the **User ID** and **Password**.

__d. In the **Client Access License Types** table, click **Add...**



__e. In the **Upload License Files** dialog, use the **Browse** button to locate the `RTC-Developer-10-C-License-4.0.zip` file. The file will be uploaded and the **Next** button will activate. Click **Next** and jazz.net will be contacted to register your free licenses.



__f. Read the license that is presented then select **I accept the terms in the license agreement.** Then, click **Finish**.

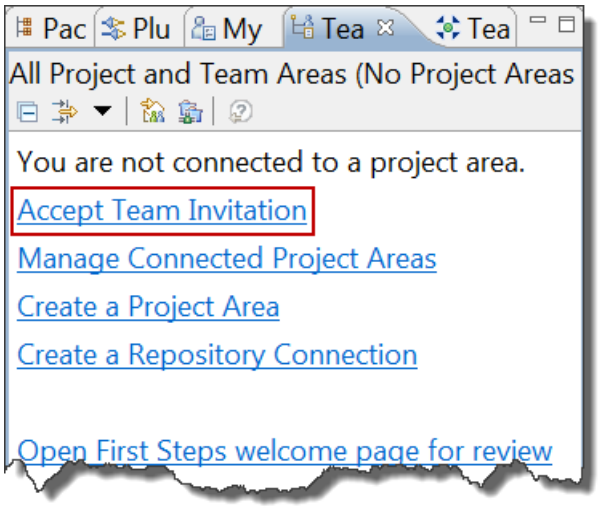
__g. The **Client Access License Types** table will now show your 10 free **Developer** CALs in addition to the trial developer CALs. The other CALs are still in place. In addition, the assignment of a Developer CAL to the ADMIN id has been upgraded to one of the free Developer CALs. The trial developer CALs are no longer assignable.

1.4 Complete Setup of Your RTC Eclipse Client

__16. Return to the RTC Eclipse client you already have running. If you shut it down earlier, start it again (`C:\RTC40Dev\installs\TeamConcert\jazz\client\eclipse\eclipse.exe`) using the `C:\RTC40Dev\workspaces\Dev1\WS` workspace. Also, make sure you are in the **Plug-in Development perspective** that you opened earlier.

__17. Connect to the project area.

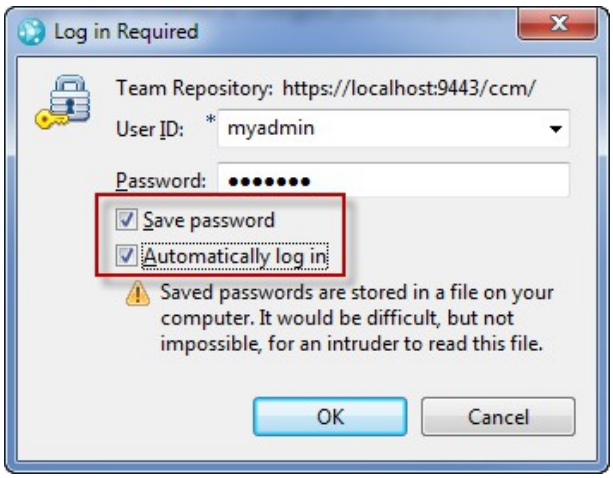
__a. On the left, switch to the **Team Artifacts** view and click the **Accept Team Invitation** link.



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

```
teamRepository=https://localhost:9443/ccm/
userId=myadmin
userName=myadmin
projectAreaName=RTC Extension Workshop
```

__c. When prompted, make sure `myadmin` is entered for both the **User ID** and **Password**. Also, check the **Save password** and **Automatically log in** check boxes. Then click **OK**.

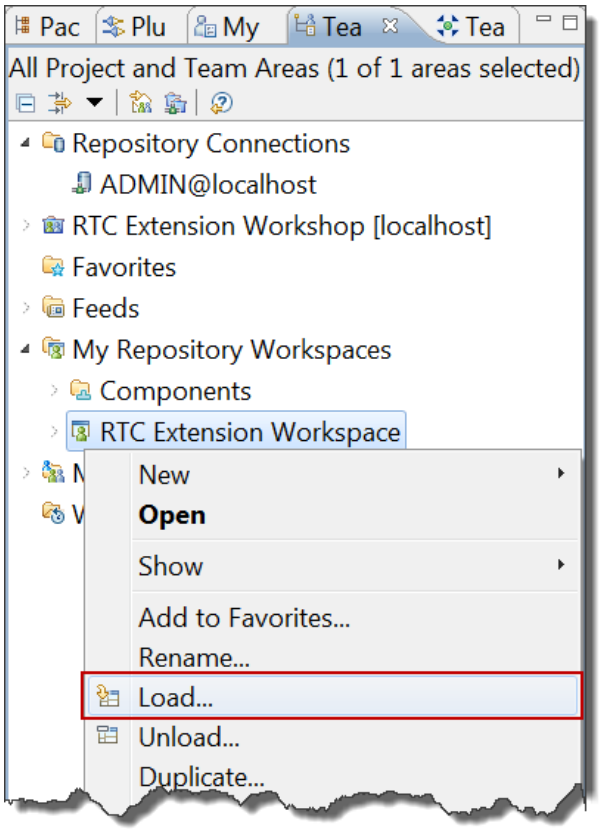


__d. If prompted with a **Repository Connection Certificate Problem**, select the **Accept this certificate permanently** radio button and then click **OK**.

__e. Close the project area editor that opens.

__18. Load the workshop repository workspace.

__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, make sure **Find and load Eclipse projects** is selected and then click **Finish**.

__c. Verify that there are now four 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 (component in this context will be defined at the top of lab 2). You will use these in subsequent labs. The third (**RTC Extension Workshop Configuration**) contains Eclipse launch configurations. In the rest of this lab you will learn how to use these launches. The fourth 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 **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.

__19. Gather remaining configuration files for Jetty based launches.

__a. The **RTC Extension Workshop Configuration** project also contains some configuration files for use by the Jetty based launches. There are two files that need to be included but are not part of what was just loaded. You will now copy them from your server installation in order to make sure they match the version of your server and SDK.

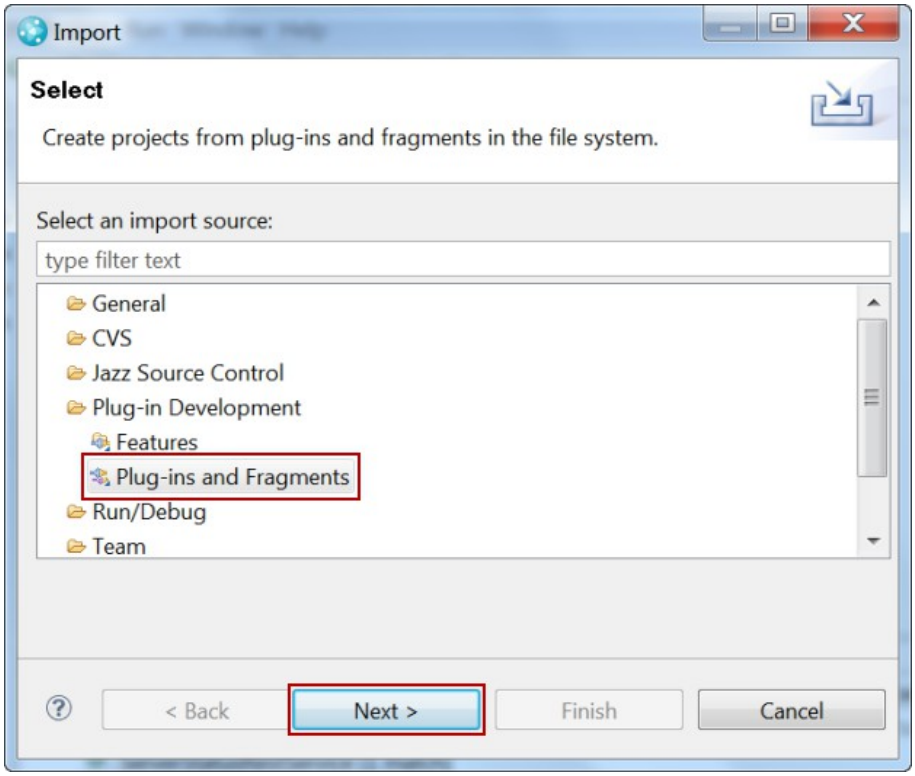
__b. The two files you need are **services.xml** and **scr.xml** from your server's ccm application configuration. You will find them in the `C:\RTC40Dev\installs\JazzTeamServer\server\conf\ccm` folder.

__c. You can drag or copy them from the Windows Explorer into the **conf\jazz** folder in the **RTC Extension Workshop Configuration** project in the **Package Explorer** view.

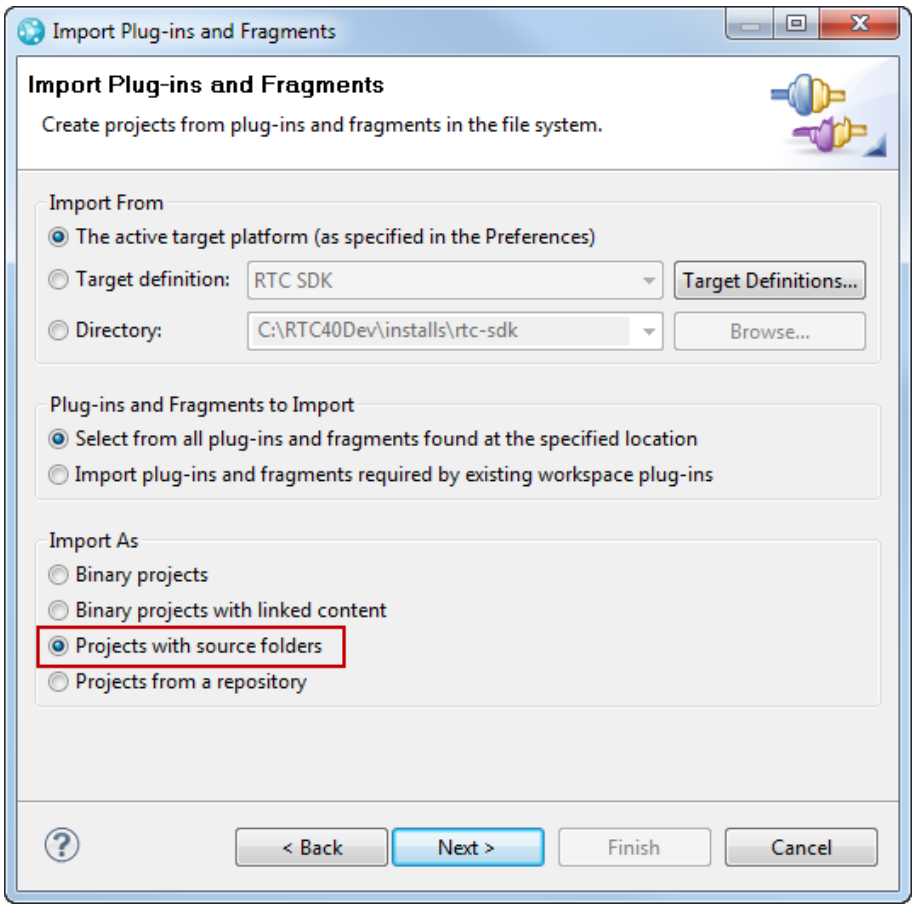
__d. This will give you two unresolved local changes in the **Pending Changes** view. You do not need to do anything with these.

__20. Import plugins for Jetty based launches. There are three plugins you will need to import (one from the RTC SDK and two from your installed server) for use with the Jetty based launches that you will try out later in this lab.

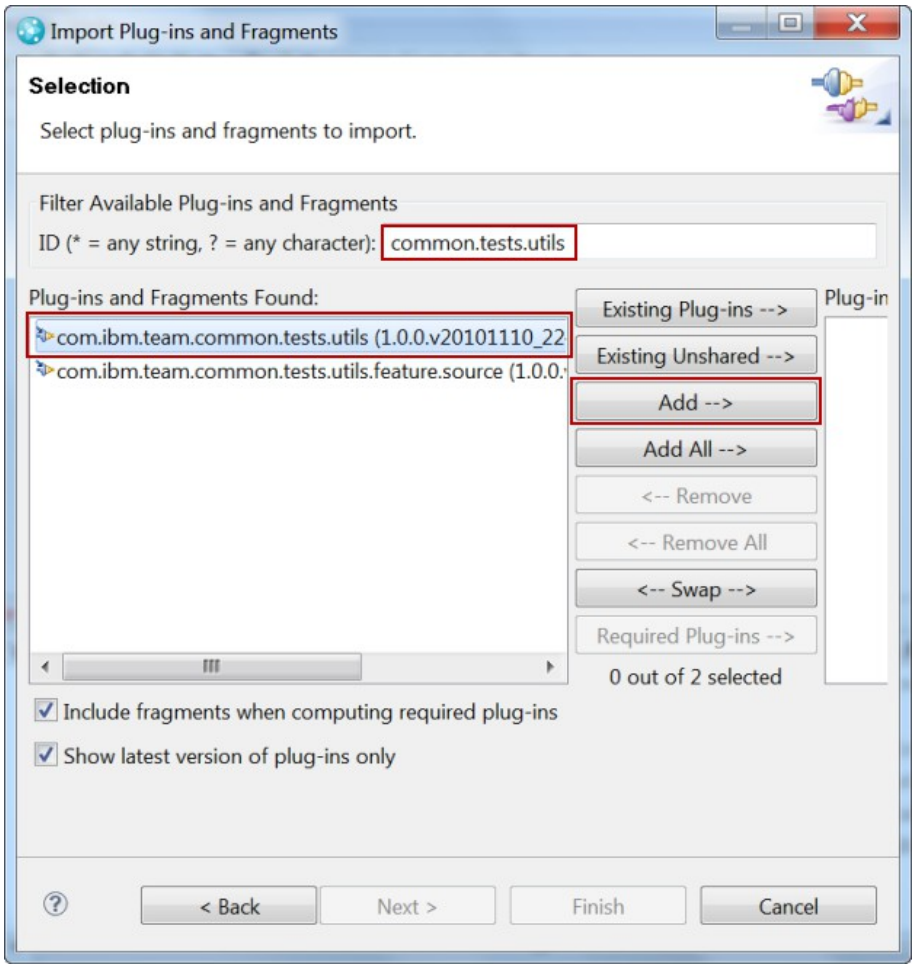
__a. First, import the JUnit test plug-in that contains the database creation code. From the menu bar, select **File > Import…** and then in the **Import** wizard, select **Plug-in Development > Plug-ins and Fragments** as shown here and then click **Next**.



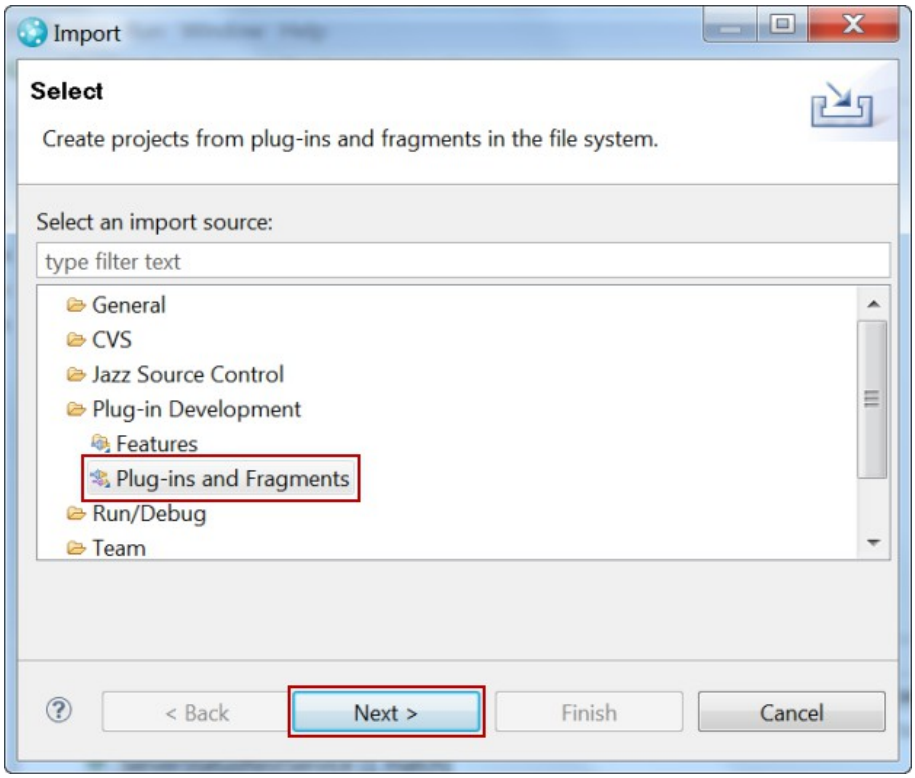
__b. On the second page of the wizard, make sure your selections match those shown here. The only one you should have to change is highlighted. Then, click **Next**.



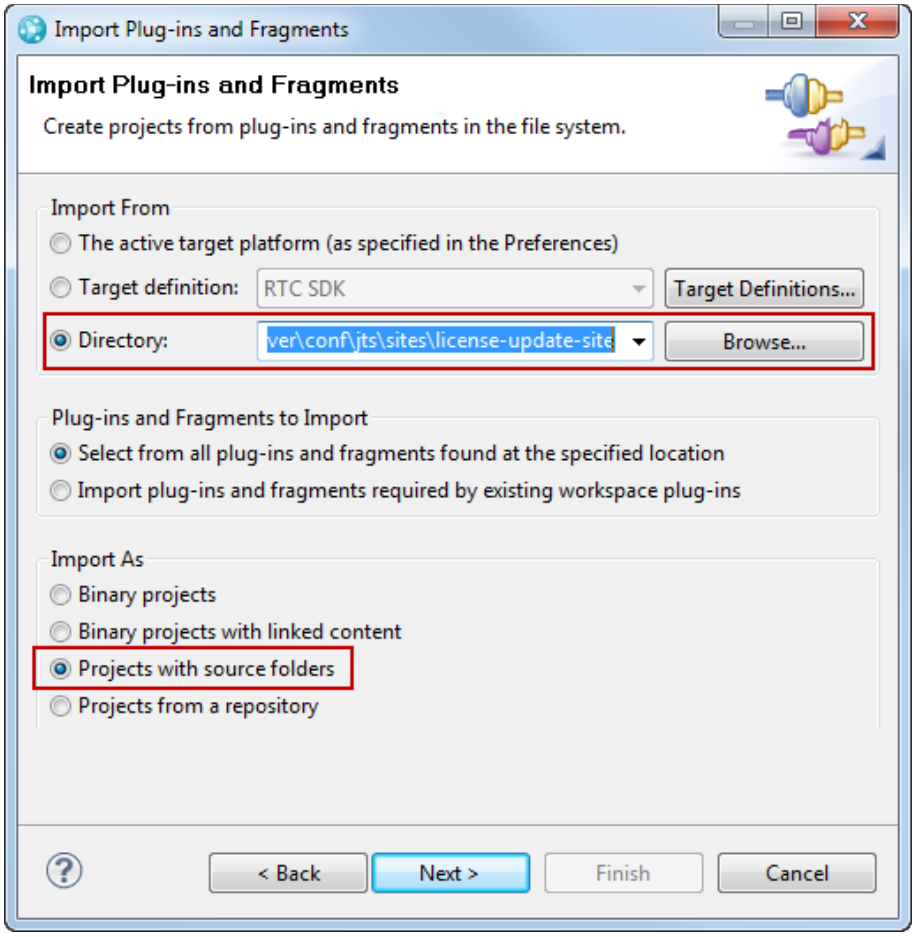
__c. On the third page of the wizard, enter `common.tests.utils` into the **ID** field. This will filter the plug-ins list. Select the **com.ibm.team.common.tests.utils** plug-in in the list, click **Add -->** and then click **Finish**.



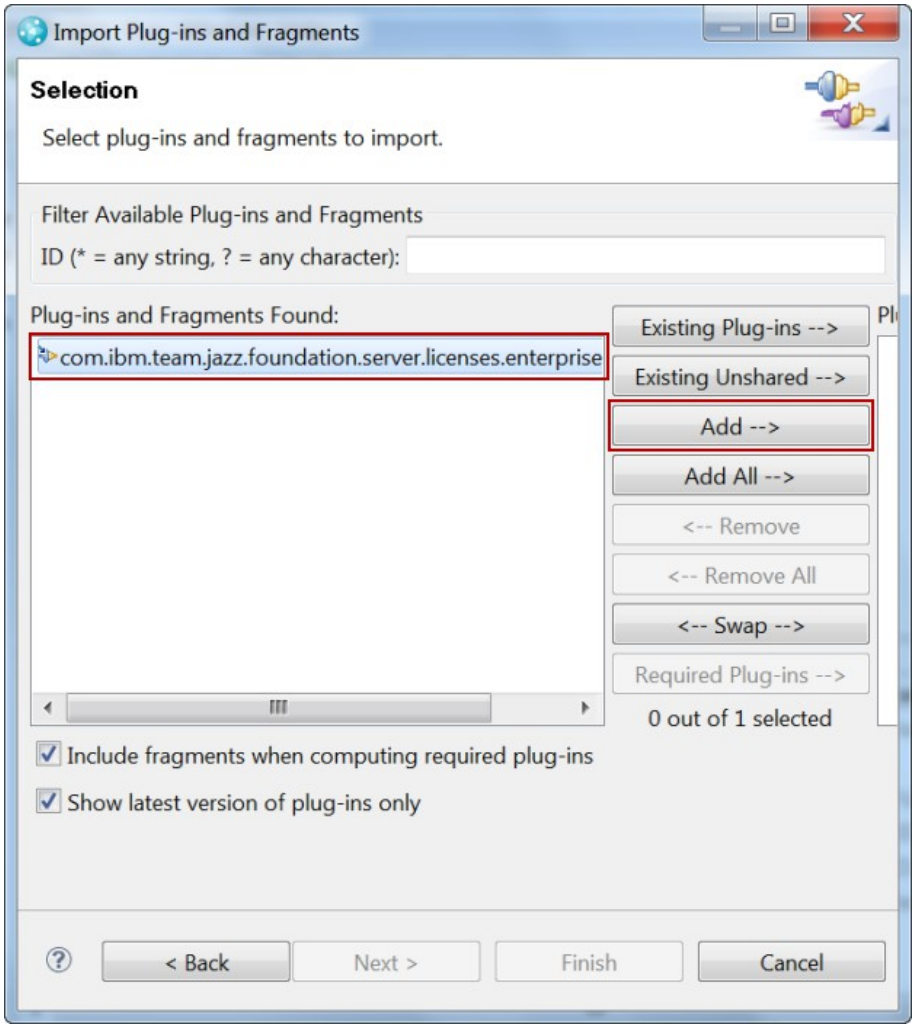
__d. Next, import the server license from the server installation. This will override the development time server license you would otherwise be using in a Jetty launch with the permanent server license. It is likely that the development license has expired. As before, from the menu bar, select **File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



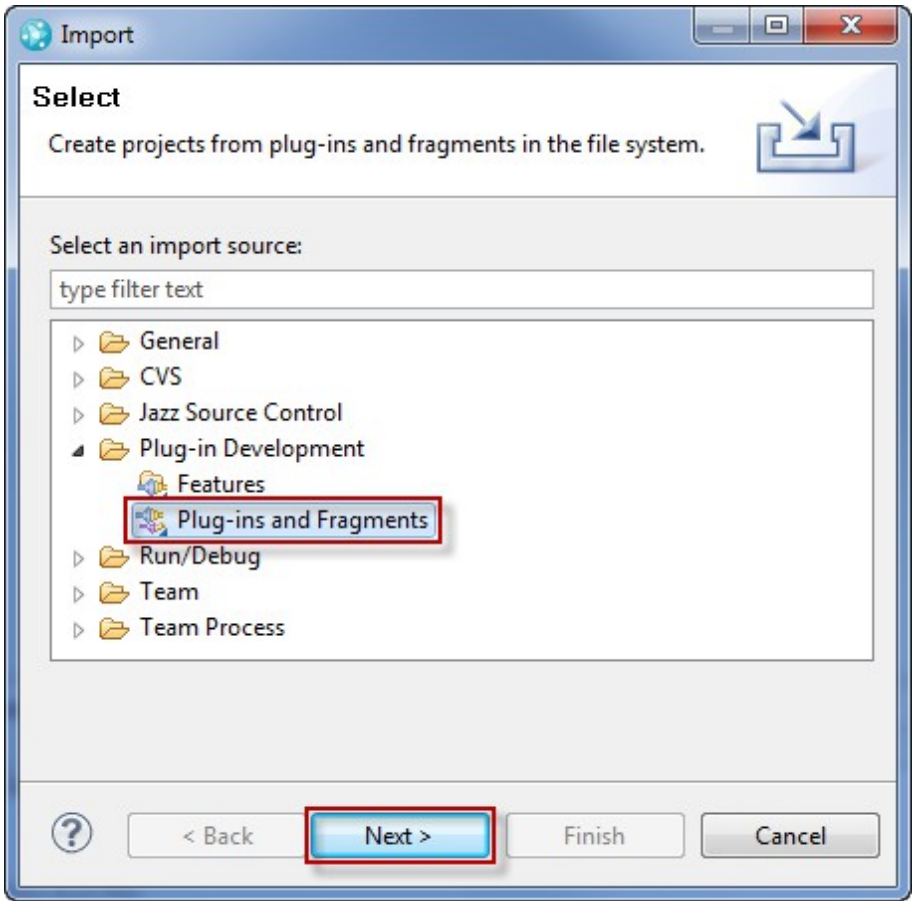
__e. This time on the second page of the wizard, make sure your selections match those shown here and then click **Next**. The major difference from last time is the selection of a different place to import from. The **Plug-in Location** field should be set to (use the **Browse…** button to find it):
`C:\RTC40Dev\installs\JazzTeamServer\server\conf\jts\sites\license-update-site`



__f. On the third page of the wizard, select the **com.ibm.team.jazz.foundation.server.licenses.enterprise-ea** plug-in in the list. Then click **Add -->** and finally click **Finish**.



__g. Finally, import the client access licenses (CALs) from the server installation. 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**.



__h. 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:\RTC40Dev\installs\JazzTeamServer\server\conf\jts\sites\clm-activation`.

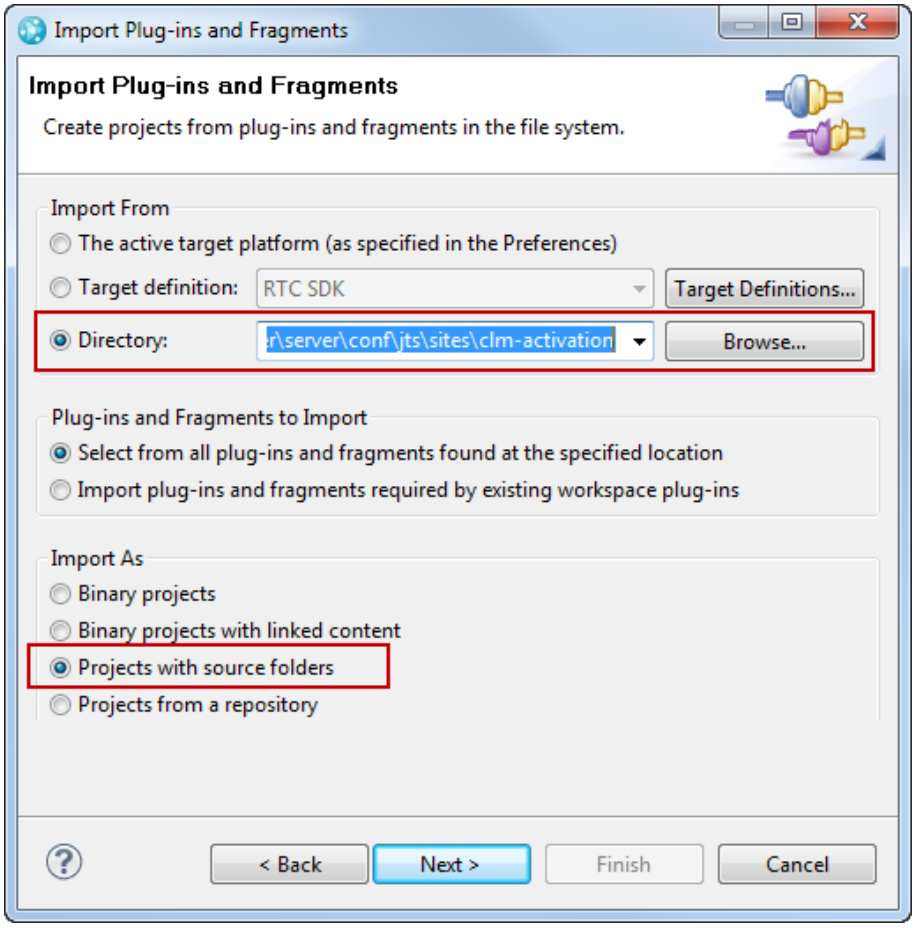
License File Names Might Change

The file names might be subject to change in future releases. If you don't find the files, look out for other names.

RTC installed from ZIP File

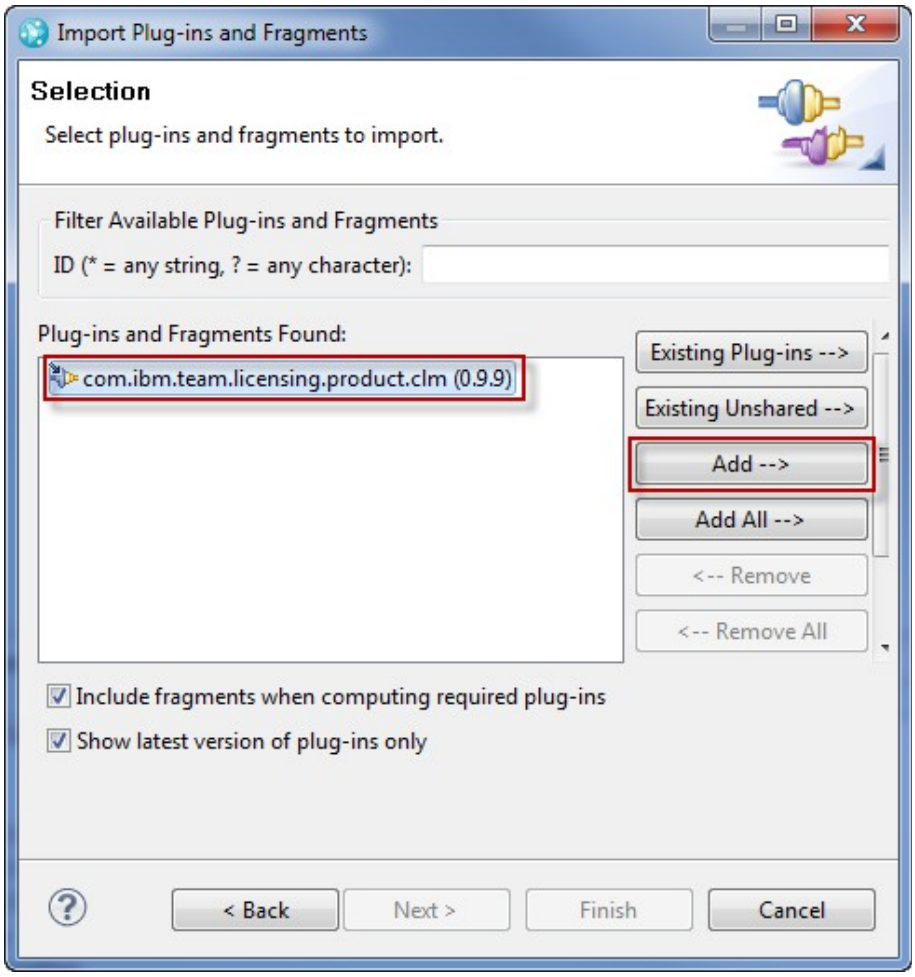
If you installed the zip version of RTC, this plugin will not exist. In this case look for the folder: C:\RTC40Dev\installs\JazzTeamServer\server\conf\jts\sites**rtc-standalone-activation**

If you can not find any of these folders, you probably forgot to install the “Rational Team Concert Required Base License Keys, Including Trials” installation package back at step 3.b. You will need to return to the server installation and install this package into the same package group as your server. You can then return to this step. Later, you will want to confirm that your myadmin user has a developer CAL.



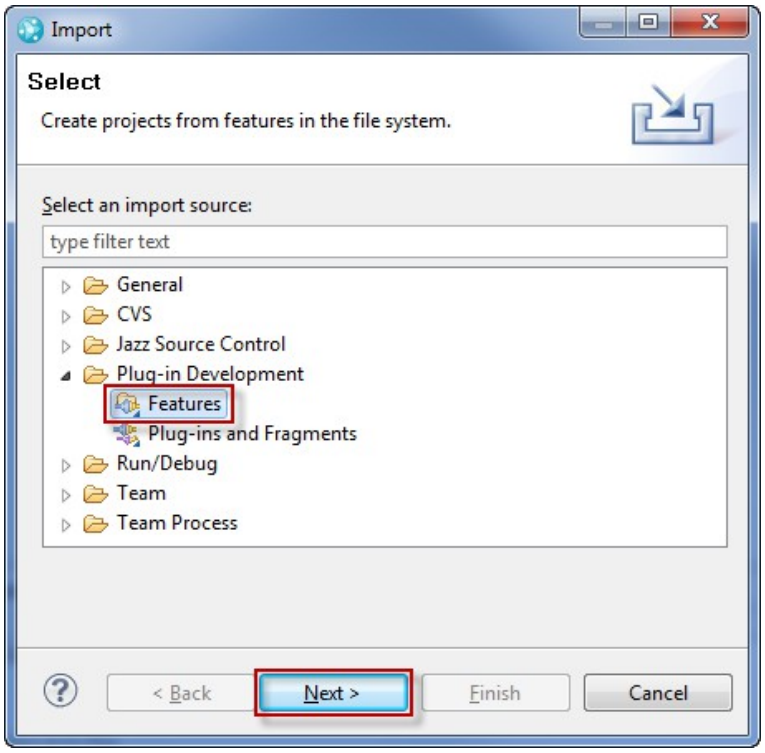
__i. On the third page of the wizard, select the **com.ibm.team.licensing.product.clm** plug-in in the list.

In case you installed the RTC standalone ZIP version choose **com.ibm.team.licensing.product.rtc-standalone.**
Then click **Add -->** and finally click **Finish**.



__21. Import a feature to make launching the RTC Eclipse client much easier. In the simple zip file installation of the RTC Eclipse client, there is an umbrella feature that includes all the RTC Eclipse client features. This feature is not in a client installed via Installation Manager nor is it in the RTC SDK. It is, however, very convenient for launching a RTC Eclipse client for debug.

__a. From the menu bar, select **File > Import…** and then in the **Import** wizard, select **Plug-in Development > Features** as shown here and then click **Next**.



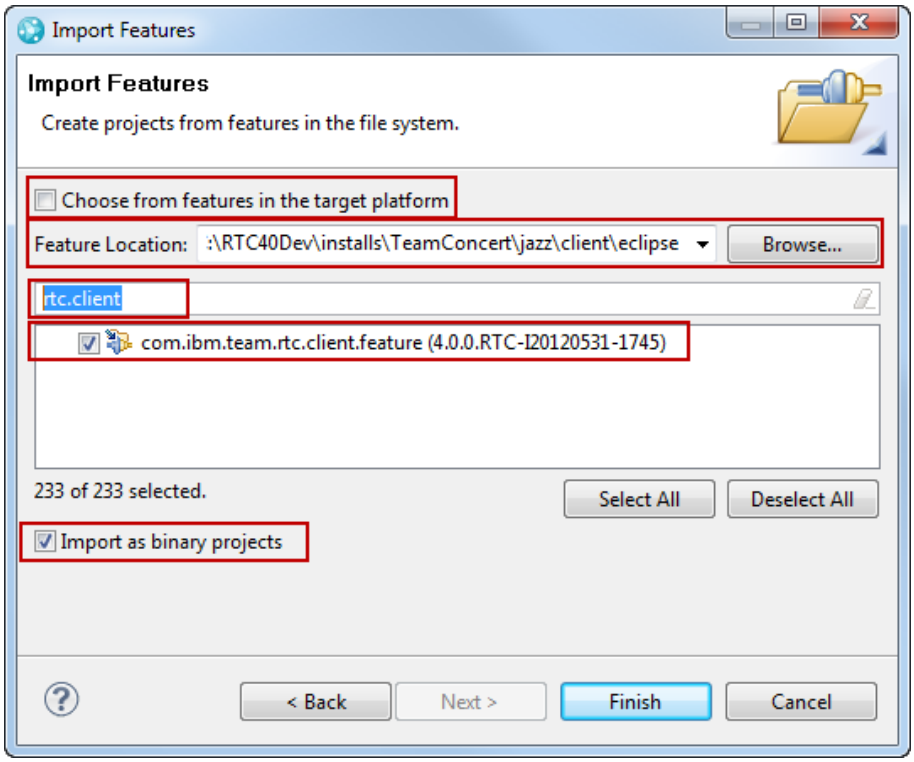
__b. On the second page of the wizard

__i. Deselect the **Choose from features in the target platform** checkbox.

__ii. The **Feature Location** field should be set to (use the **Browse…** button to find it): `C:\RTC40Dev\installs\TeamConcert\jazz\client\eclipse.`

__iii. Click **Deselect All**.

__iv. Type `rtc.clien` to narrow down the selection. Scroll down the list to the **com.ibm.team.rtc.client.feature** and check it.



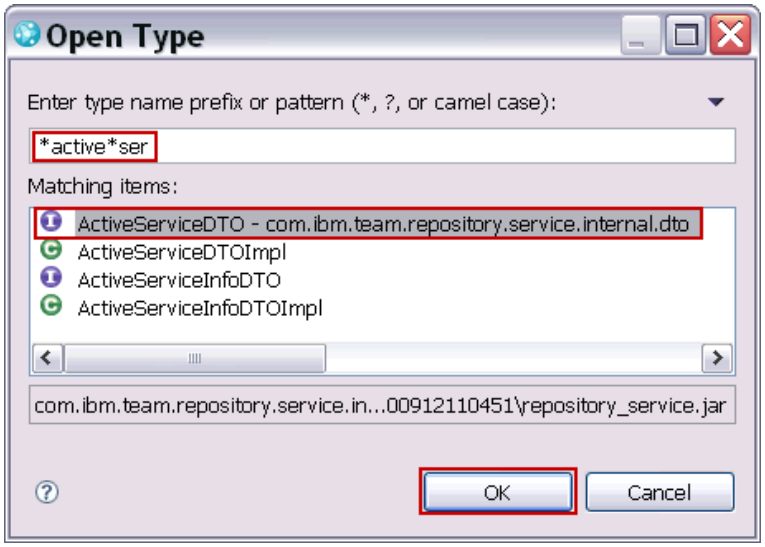
__v. Click **Finish**.

1.5 Test connecting the Eclipse debugger to Tomcat

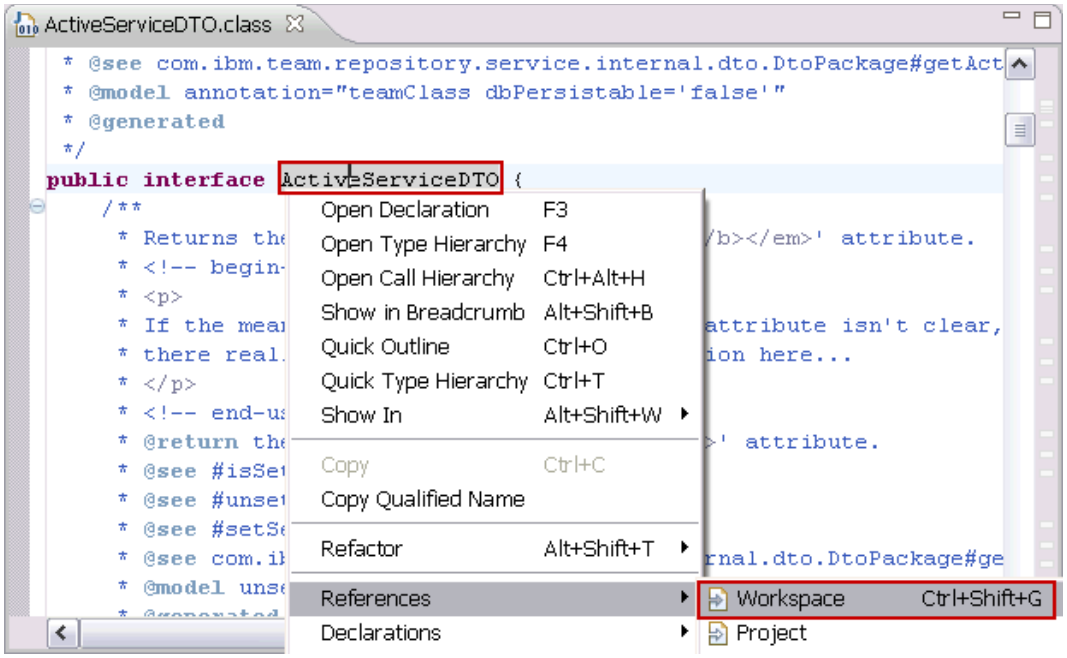
__22. Set a breakpoint to be used to verify the debugging connection.

__a. In your RTC Eclipse client (in the **Plug-in Development** perspective you opened earlier), select **Navigate > Open Type…** from the menu bar.

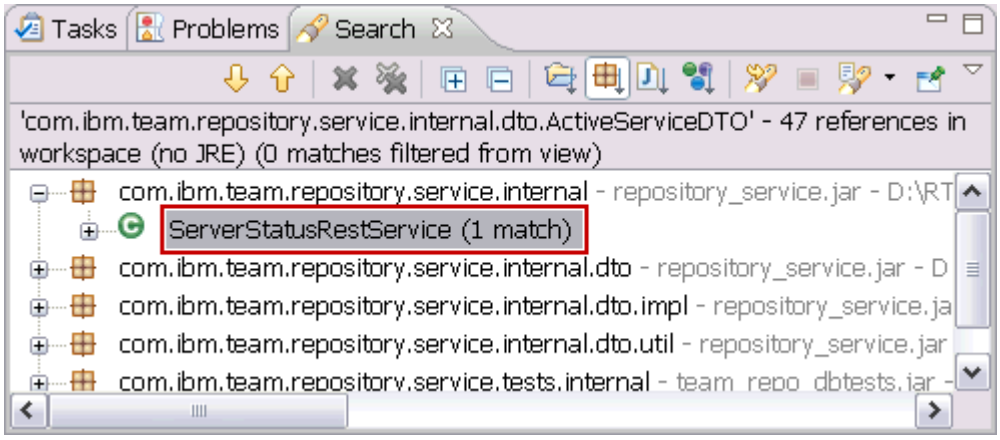
__b. In the **Open Type** dialog type `*active*ser` in the pattern field. Several types will appear (depending on the version). Select the **ActiveServiceDTO** interface as shown here and then click **OK**.



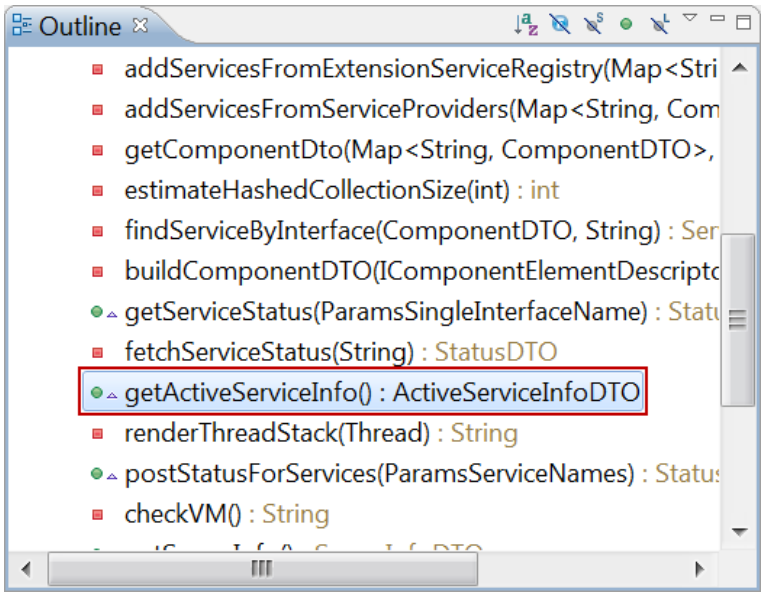
__c. When the Java editor opens on the class, the class name will be highlighted. Right click the class name and select **References > Workspace**.



__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.



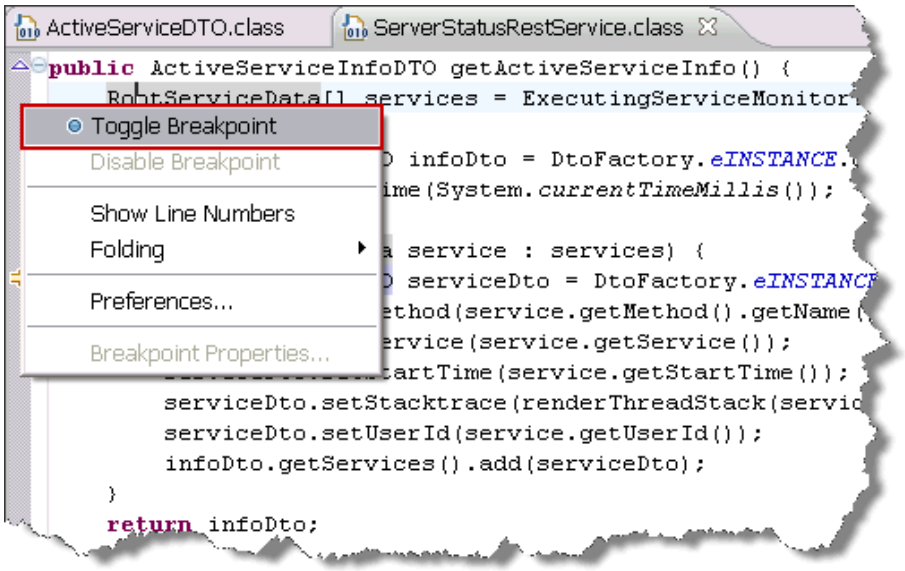
__e. The **Outline** view now shows the structure of the ServerStatusRestService class. In the **Outline** view, click the **getActiveServiceInfo()** method.



__f. The editor is now showing the **getActiveServiceInfo()** method. Set a breakpoint on the first line of the method. Right click in the shaded area to the left of the first line to get the menu.

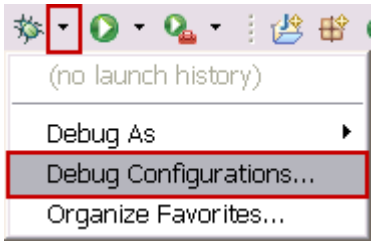
Production Code Might Change

The screen shot below is from the 4.0 SDK. In newer versions the code might look different from the screen shot below.

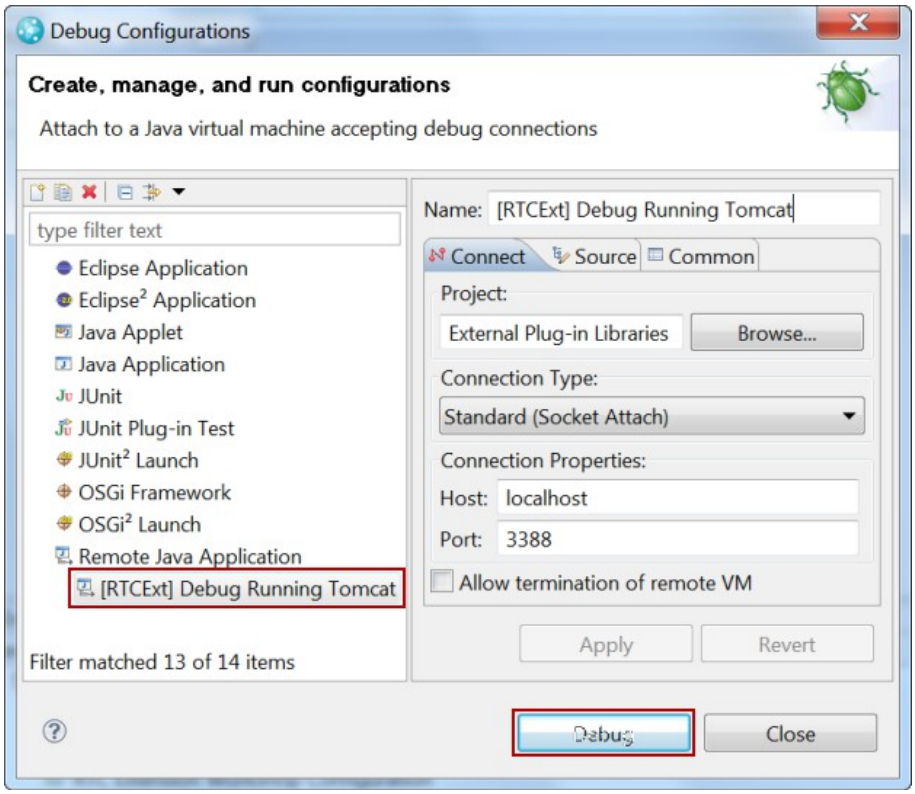


__23. Attach the Eclipse debugger to the RTC server.

__a. From the **Debug** toolbar icon dropdown select **Debug Configurations…**



__b. In the **Debug Configurations** dialog, expand the **Remote Java Application** launch type, select the **[RTCExt] Debug Running Tomcat** launch configuration and then click **Debug**. If you had to use a different debug port when starting Tomcat, adjust the **Port** value here before debugging. Also note that if you switch to the **Common** tab that the **Save as** location is set to inside one of the projects you loaded (\RTC Extension Workshop Configuration\launches). This will be true for all the launches you use with this workshop. You may also notices some other launches that start with [Standard]. These come from the test plugin you imported from the RTC SDK. You will not use them in this workshop.

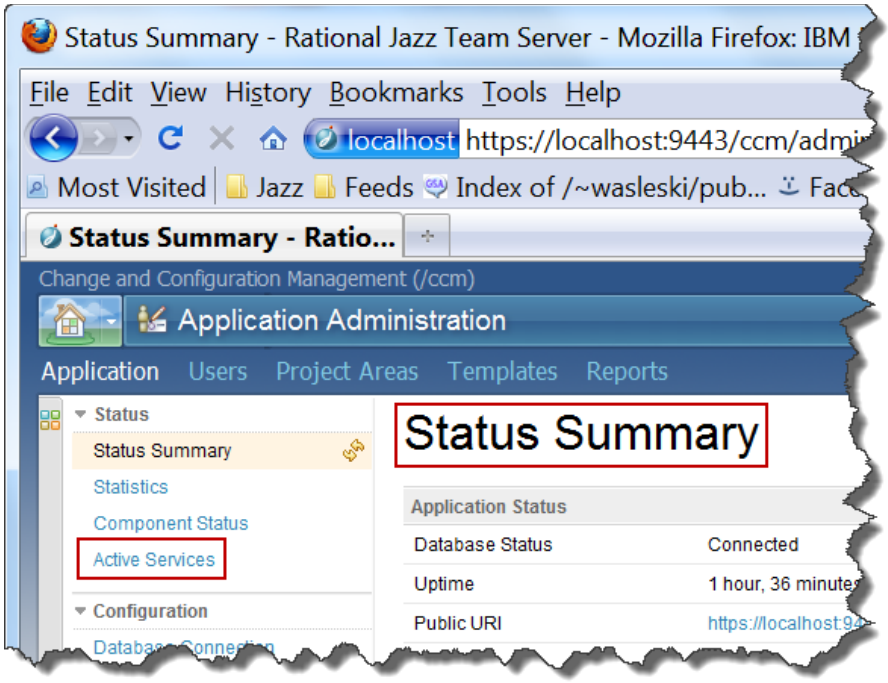


__24. Use the RTC Web UI to trigger the breakpoint.

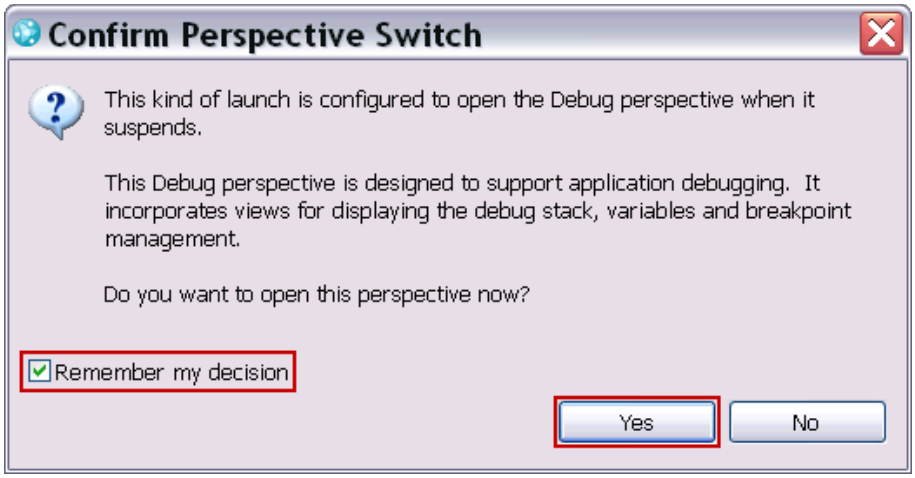
__a. Open your browser (if it is not already open) and enter the URL https://localhost:9443/ccm/admin.

__b. If prompted, login with `myadmin` as both **User ID** and **Password**.

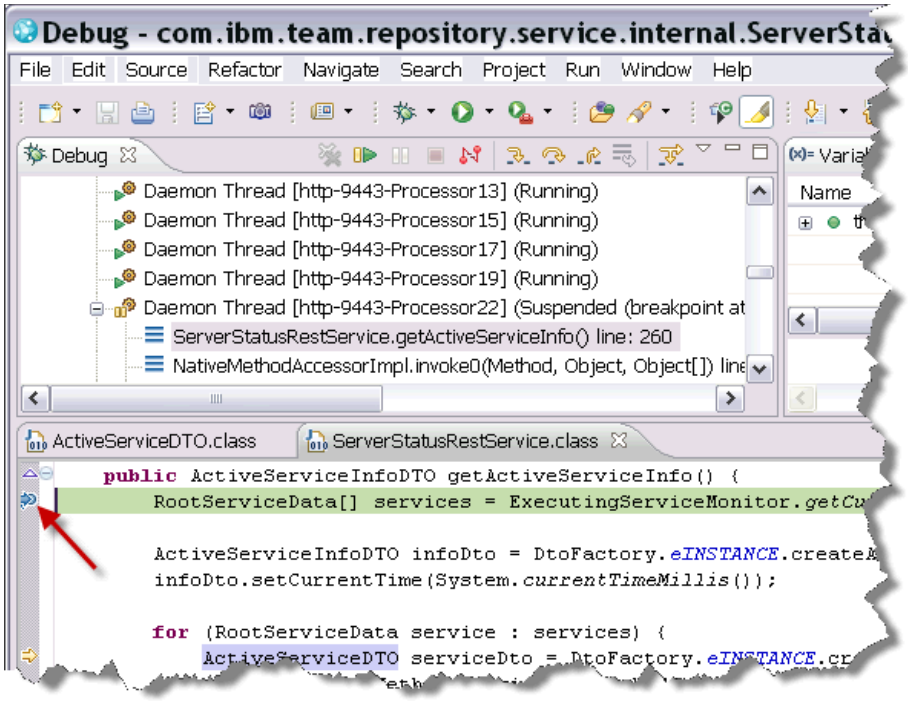
__c. When the **Status Summary** page appears, click the **Active Services** link on the left.



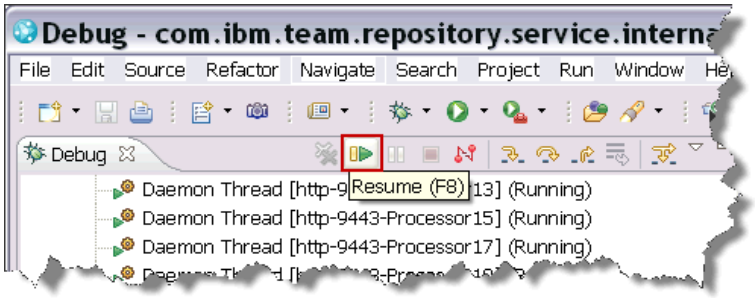
__d. The breakpoint will trigger and the RTC Eclipse client should come to the foreground (or flash in the Windows taskbar if minimized). If you are prompted to switch to the Debug perspective, click the **Remember my decision** checkbox if you wish, and then click **Yes**.



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

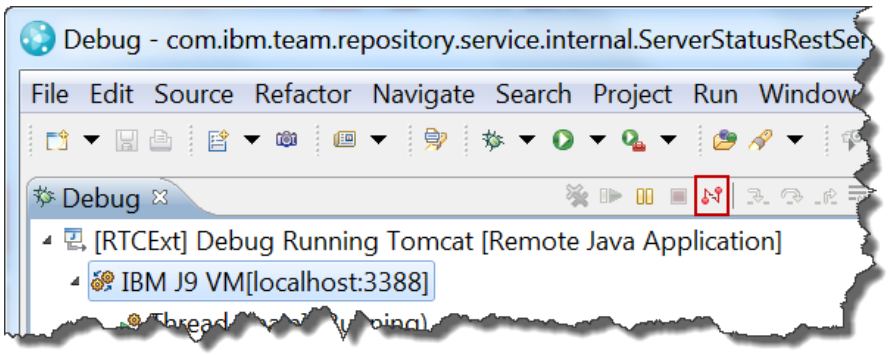


__f. Click the **Resume** toolbar button to resume execution of the server.



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

__h. Disconnect the debugger from the running Tomcat server by clicking the **Disconnect** toolbar button.



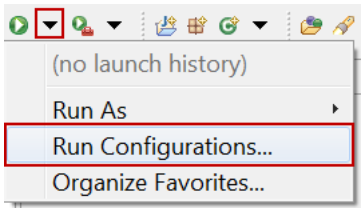
1.6 Test the Jetty Based Server Launch

	As mentioned earlier, you will now 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 that this launch runs the server as one application at the "\jazz" context root and not as separate JTS and CCM applications. This is generally fine for development and you do have the Tomcat server with split applications for final testing.

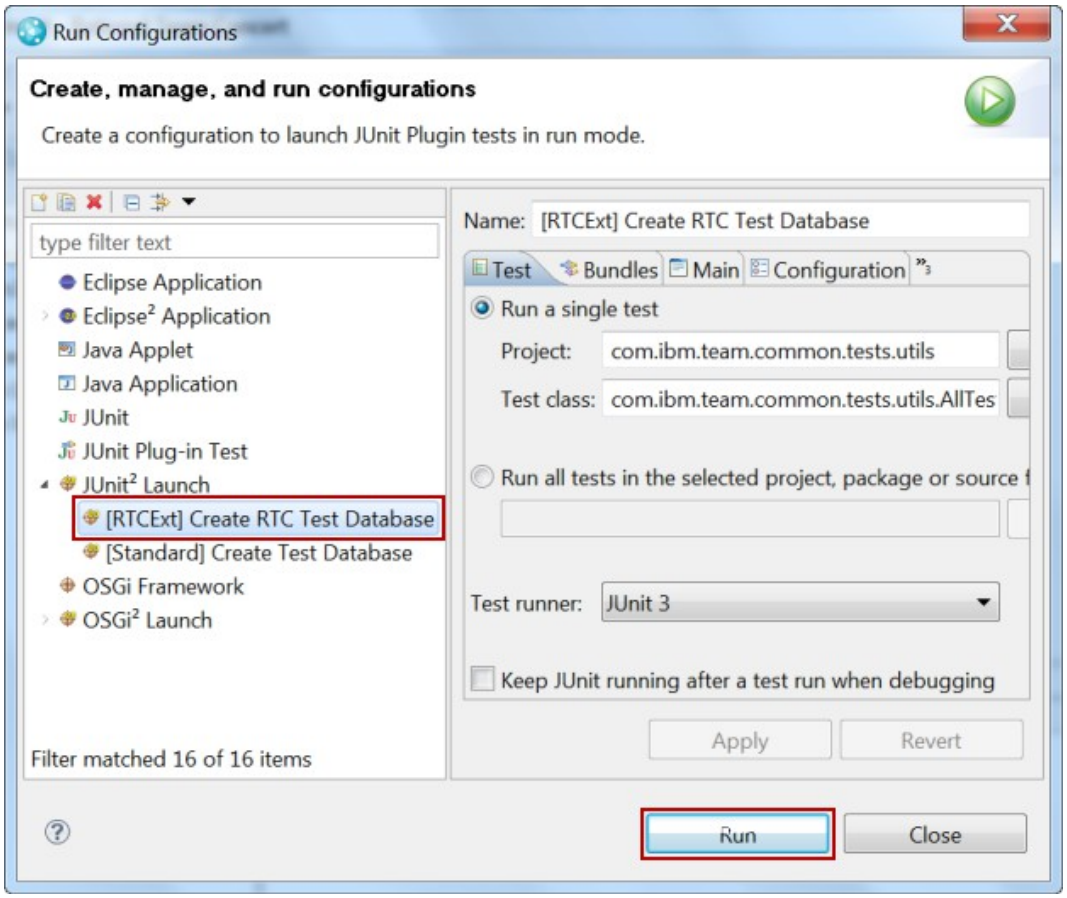
__25. Create the development time repository database. Note that this process will create a "server" folder as a sibling of your Eclipse workspace. The database and eventually its indexes will be contained within this folder. If you ever want to delete the database and indexes and recreate them, you can simple delete the server folder and rerun this process.

__a. The database creation test you are about to run uses a Jetty server during initialization of the database. Unfortunately, that server must run at the same ports as the Tomcat server you currently have running. You will need to temporarily stop the Tomcat server. You will be able to restart it after the database is created. This will not be a problem when running your Jetty test server. It and the Tomcat server will use different ports. So, return to the Windows Explorer and navigate to `C:\RTC40Dev\installs\JazzTeamServer\server` and run the **server.shutdown.bat** file. Wait for the server to stop and then proceed with the next step.

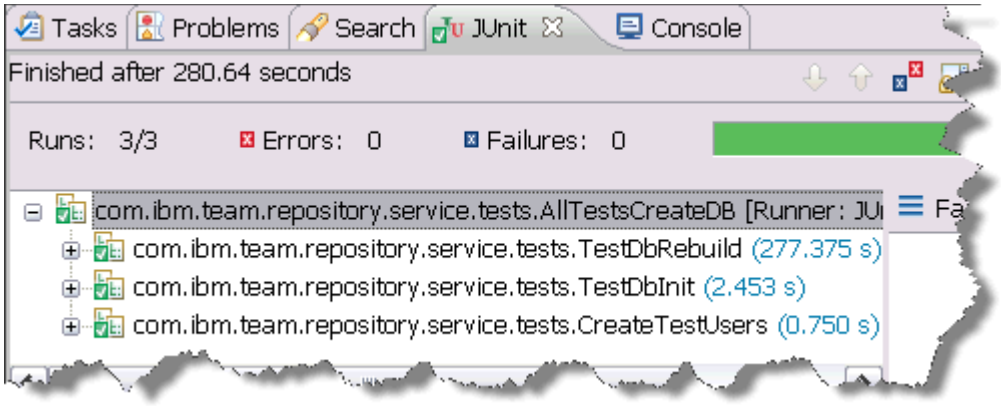
__b. Select **Run Configurations…** from the dropdown menu off the **Run** toolbar icon.



__c. On the **Run Configurations** dialog, select **JUnit² Launch > [RTCExt] Create RTC Test Database** and then click **Run**. Note that if you switch to the **Bundles** tab, you will see that three of the bundles you imported earlier (the test bundle that creates the database and the licenses) are included in this launch. You will learn about adding your own bundles to launches in subsequent labs.



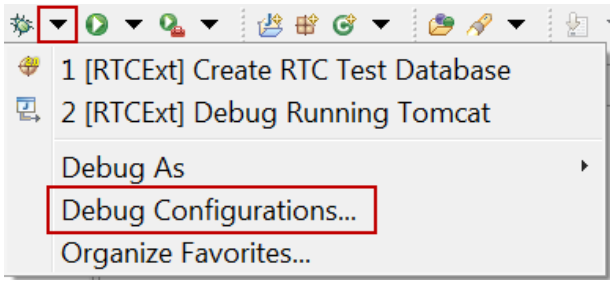
__d. This may take a while to run. The **Console** view will appear and show quite a bit of output. The **JUnit** view will also be active. When the database creation is complete, the **JUnit** view will show success. Note that the **Console** view will show some exceptions. The important thing is that the **JUnit** view shows success. If it fails, make sure you have shut down RTC and retry until it succeeds.



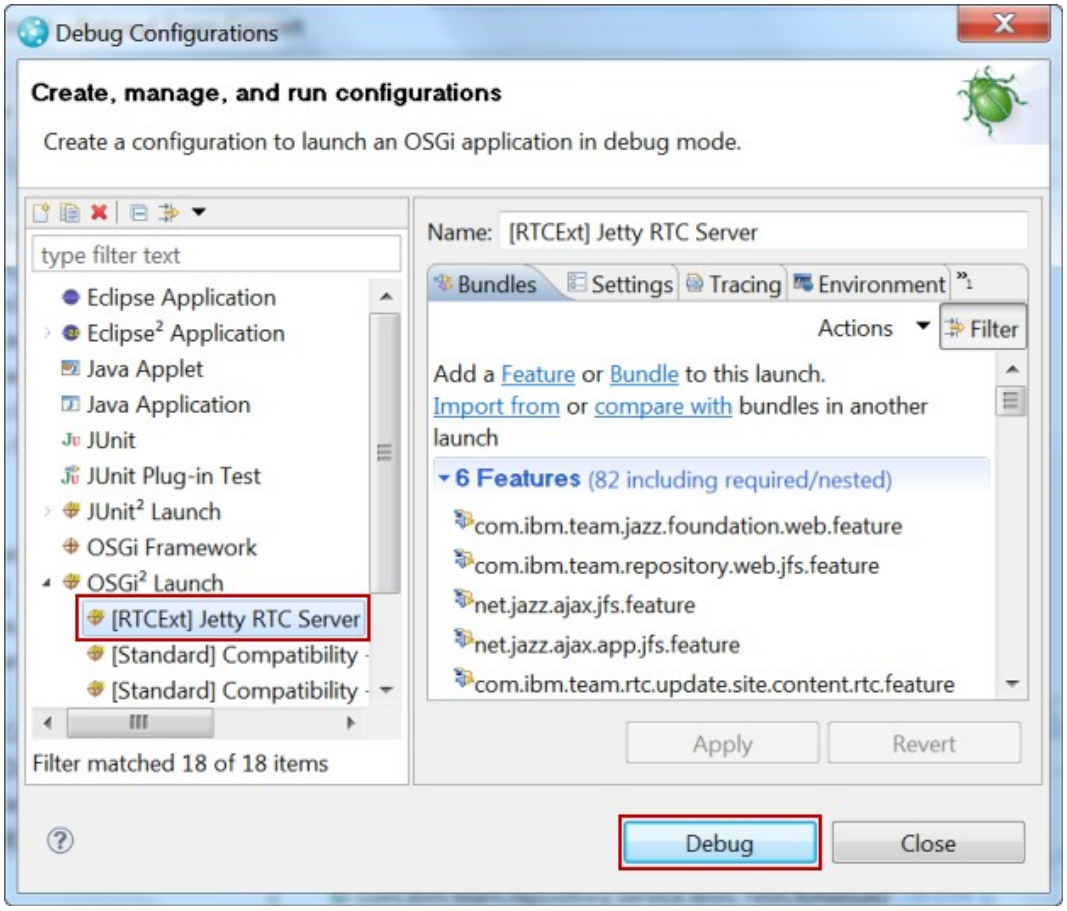
__e. You can now restart your Tomcat server. Return to the Windows Explorer and navigate to `C:\RTC40Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file.

__26. Launch the Jetty server for debug.

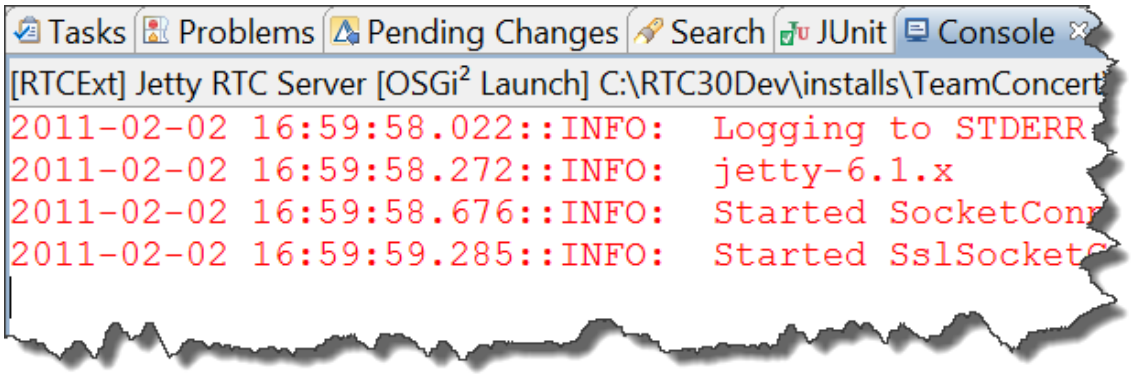
__a. Select **Debug Configurations…** from the dropdown menu off the **Debug** toolbar icon.



__b. In the **Debug Configurations** dialog, select **OSGi² Launch > [RTCExt] Jetty RTC Server** and then click **Debug**. Note that on this **Bundles** tab, that two of the bundles you imported earlier (the licenses) are included in this launch. You will learn about adding your own bundles to launches in subsequent labs. Also notice the list of **System Properties**. Many of these will be familiar to you if you have ever administered a Jazz server (location of the repository, index locations, public URL, etc). There are also two Jetty properties for setting the ports. The primary port you will used with this server is 7443 rather than 9443.



__c. Switch to the **Console** view. Log messages will appear indicating that the Jetty server has started. You might see Framework Manager exceptions. These can be ignored.



__27. 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).

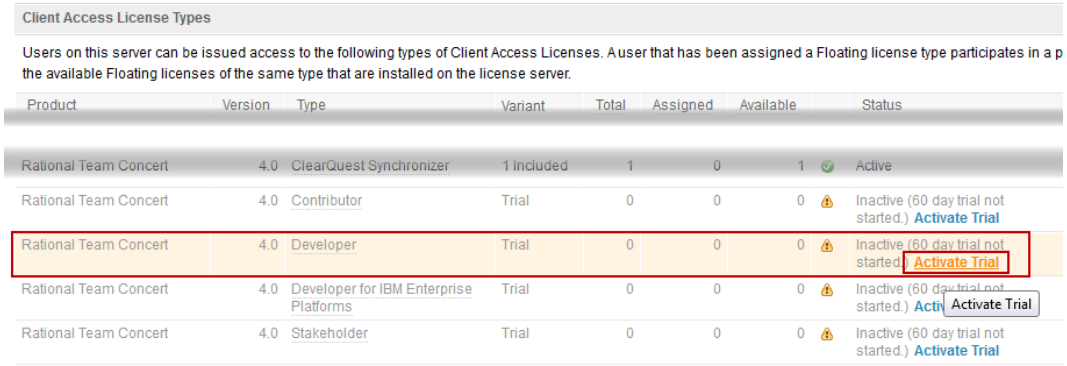
__b. Log in with **`TestJazzAdmin1`** as both the **User ID** and **Password**. For this workshop we will use the `myadmin` user for the Tomcat server and this other administrator id for the Jetty launched server. This will hopefully make things a little less confusing in that it will be more clear as to which server is being used. This new administrator id was created along with the database you created earlier. There are several other ids that were created then too.

__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. Dependent o

__28. Activate the Licenses. In the 4.0 products you need to activate the trial licenses to be able to use them. You will now perform this step in the Jazz Team Server Administration pages. In newer versions the developer trial licenses are already activated. And you can continue after verifying the fact.

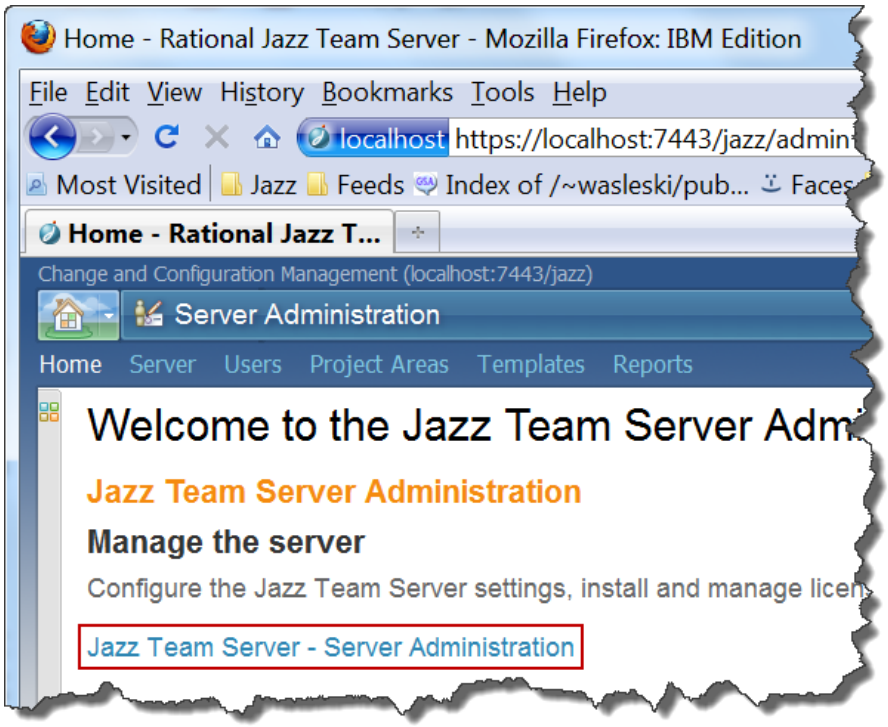
__a. Navigate to the **License Key Management** page at https://localhost:7443/jazz/admin#action=com.ibm.team.repository.admin.manageLicenses

__b. Find the trial license entry for “Rational Team Concert 4.0 Developer” and activate the trial license.

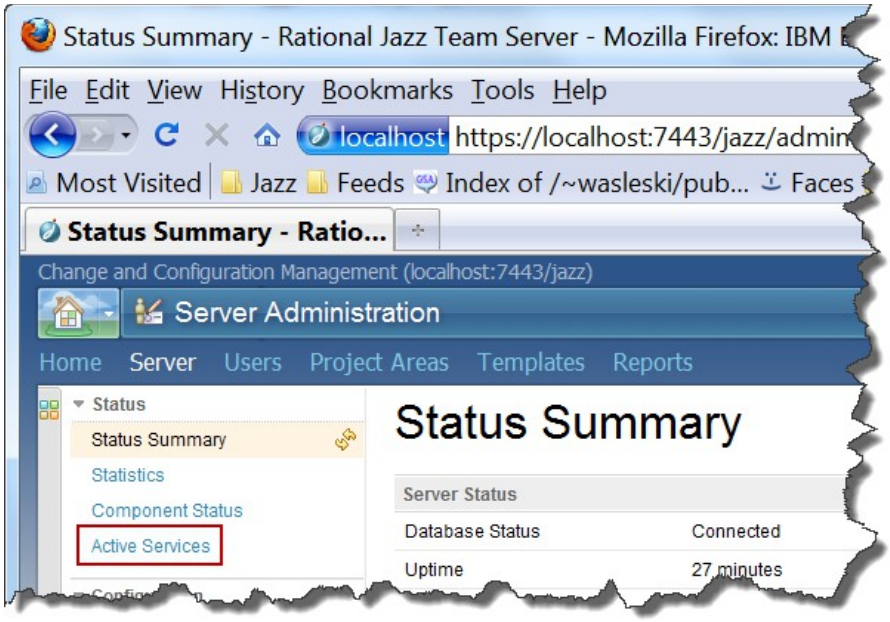


__29. Trigger the breakpoint set earlier.

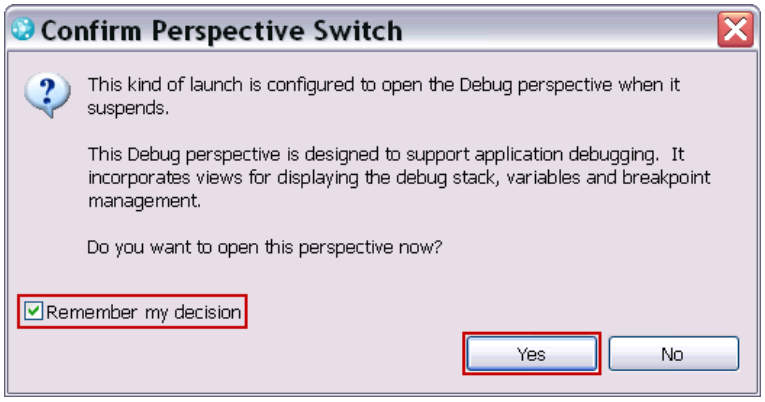
__a. Click the **Jazz Team Server - Server Administration** link under **Manage the Server**.



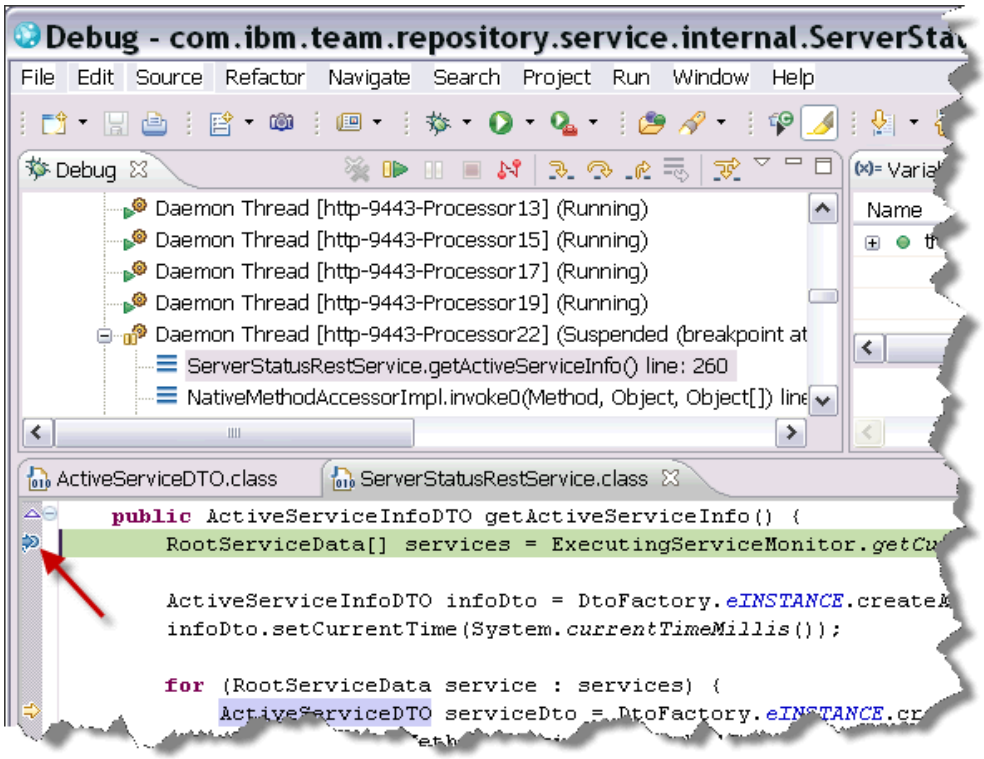
__b. As before, click the **Active Services** link on the left.



__c. 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**.



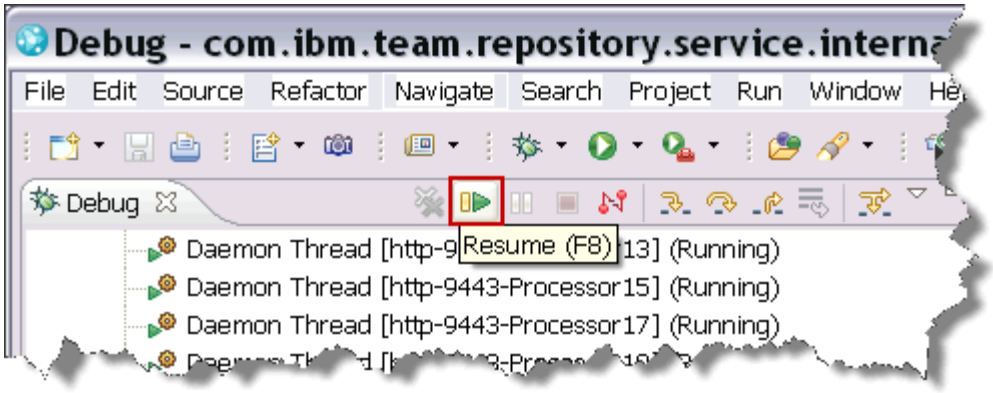
__d. You will now be in the **Debug** perspective stopped at the breakpoint you set earlier.



Make sure to remove all terminated launches to have a better overview about what runs.

Make sure not to run several server instances in parallel. They will have conflicting ports so one instance will not run. If the system does not behave as expected, please check how many instances run. You can stop all debug sessions and start again, making sure only one is launched.

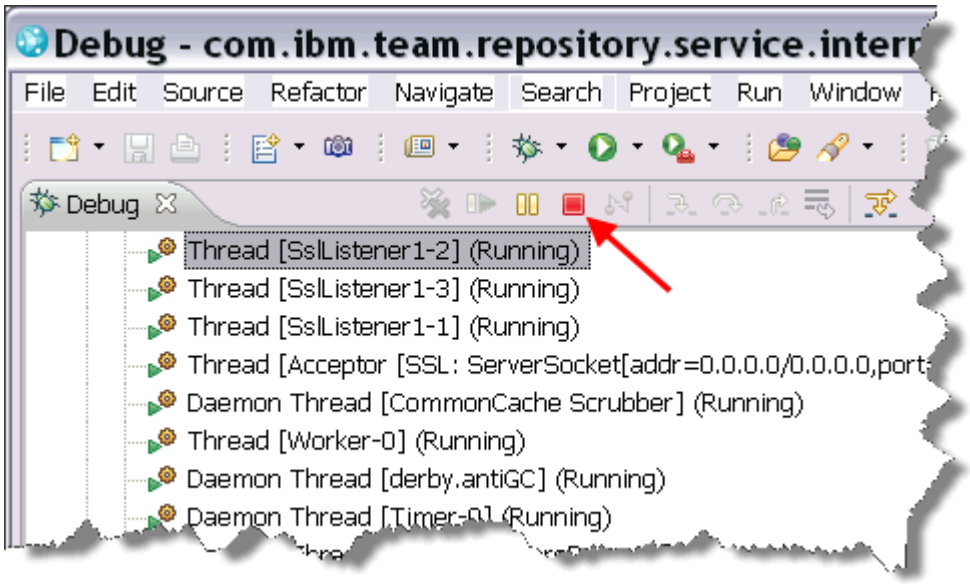
__e. Click the **Resume** toolbar button to resume execution of the server.



__30. 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.

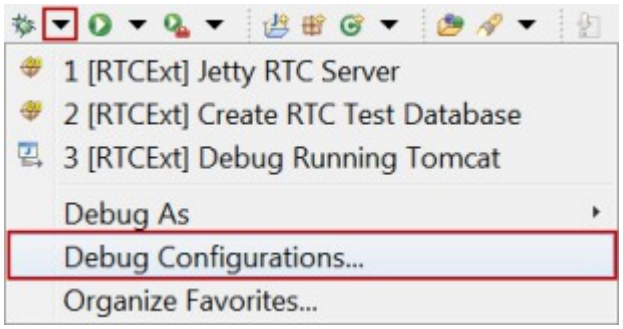


1.7 Test the RTC Eclipse Client Launch

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 test a launch for that in this section.

__31. Launch the RTC Eclipse client under debug.

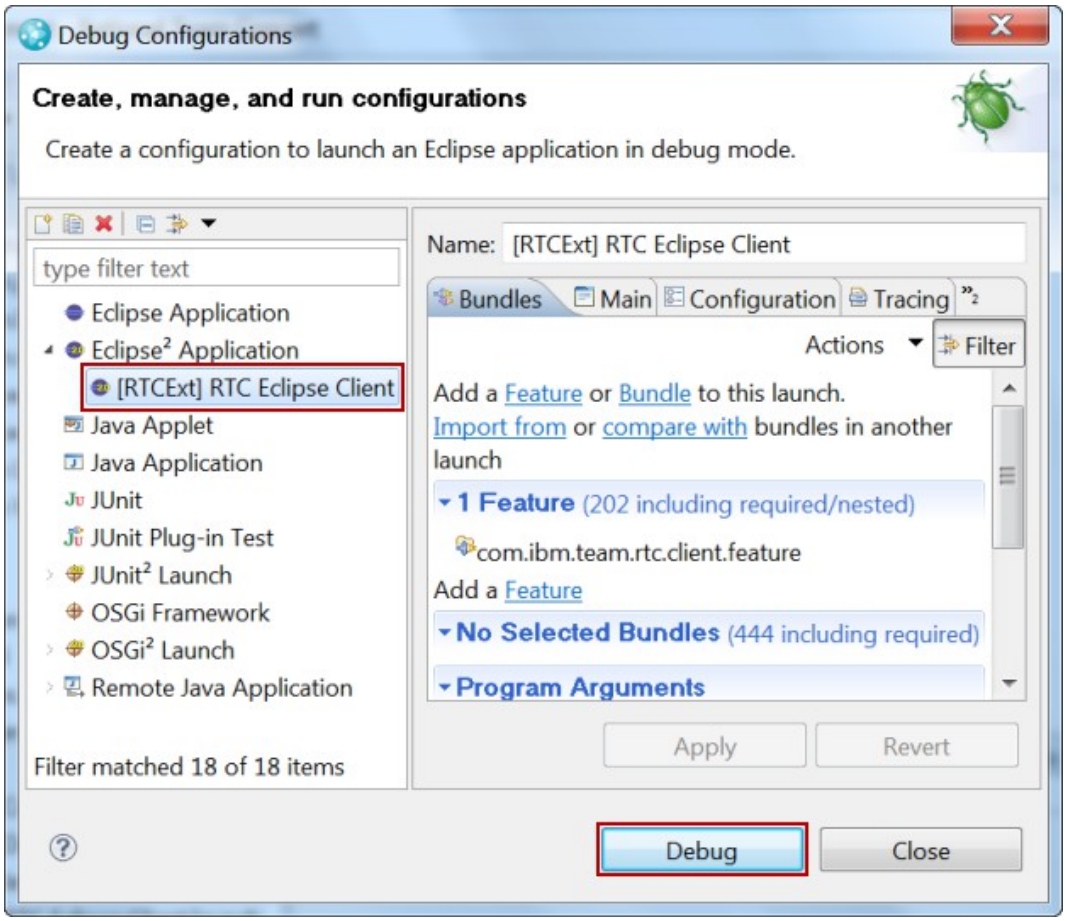
__a. Select **Debug Configurations…** from the dropdown menu off the **Debug** toolbar icon.



__b. In the **Debug Configurations** dialog, select **Eclipse² Application > [RTCExt] RTC Eclipse Client** and then click **Debug**. Note that on this **Bundles** tab, that the feature you imported earlier (the RTC client feature) is included in this launch. If you switch to the **Main** tab, you will notice two important settings.

__i. First, the launch is configured to prompt you to see if you want to clear the Eclipse workspace being used by the launched client (not the one you are in now) before launching. Usually you will answer no (and you can change the settings to not clear at all if you wish) but occasionally you will find it useful. You will not see the prompt for clearing the workspace the first time you use this launch since the workspace does not yet exist.

__ii. Second, the product being launched is the **com.ibm.team.concert.product**.



__32. The RTC Eclipse client will launch and you can use it as you normally would.

__a. If you hit a client side breakpoint, your original RTC Eclipse client will surface to handle the debugging.

__b. 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.

__33. Close the RTC Eclipse client you just launched under debug.

__34. Shutdown unless proceeding to lab 2.

__a. Close your RTC Eclipse client (the original one where you loaded code from the RTC server running under Tomcat).

__b. Return to the Windows Explorer and navigate to `C:\RTC40Dev\installs\JazzTeamServer\server` and run the **server.shutdown.bat** file.

You have completed lab 1. You now have a complete develop and debug environment for extending RTC. You have several 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:\RTC40Dev\installs\JazzTeamServer\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 workshop there will be common, service and rich client UI (Eclipse) plug-ins. 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

Note: dependent on the version of RTC you are using, you might see warnings on some of the API classes used e.g.

Discouraged access: The type ITeamBuildService is not accessible due to restriction on required library

C:\RTC406Dev\installs\rtc-sdk\plugins\com.ibm.team.build.common_3.1.700.v20140116_0309.jar

You can ignore this warnings.

__35. If your RTC development environment is not open, navigate to `C:\RTC40Dev\installs\TeamConcert\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.

__36. Return to the lab two code.

__a. In lab one, you loaded a repository workspace. Along with the launches that you used in lab one, this also loaded the lab two code. Return to the **Package Explorer** view. Verify that the two projects that define the common (**net.jazz.rtcext.workitem.extensions.common**) and service (**net.jazz.rtcext.workitem.extensions.service**) parts of your component are present. In the rest of this lab you will learn about the various parts of this initially simple participant.

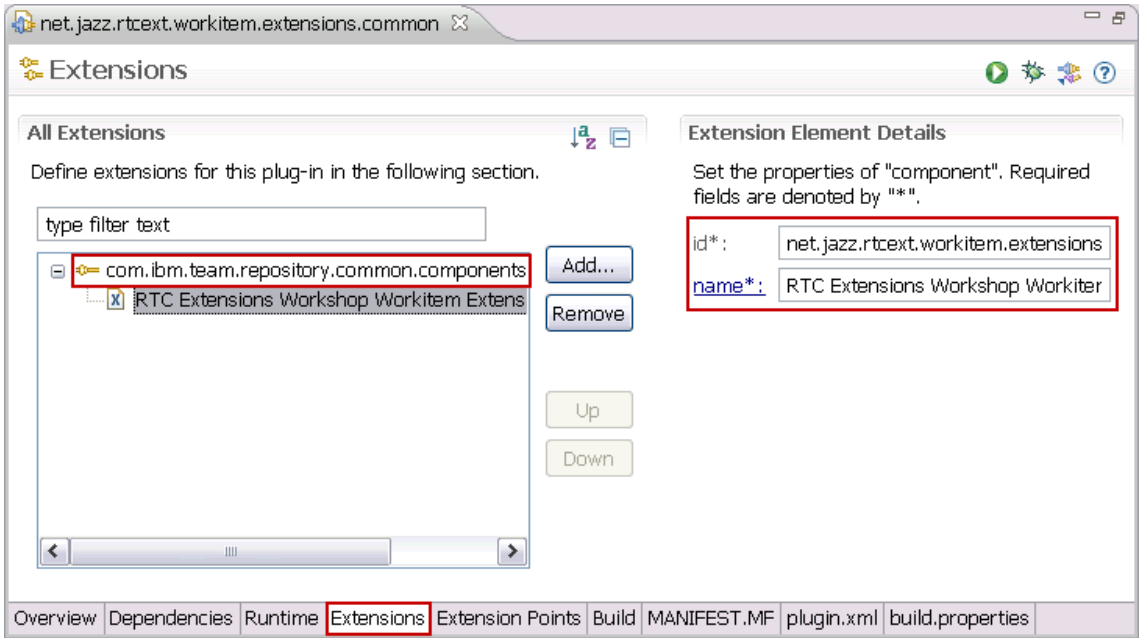
__b. You will also notice in the **Pending Changes** view, that there are incoming change sets and baselines. Do not accept them. You will make use of them in later labs. If the **Pending Changes** view is not open, select **Window > Show View > Other…** from the menubar, type `pending` into the filter field and then double click the **Pending Changes** entry.

__37. 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/helios/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.

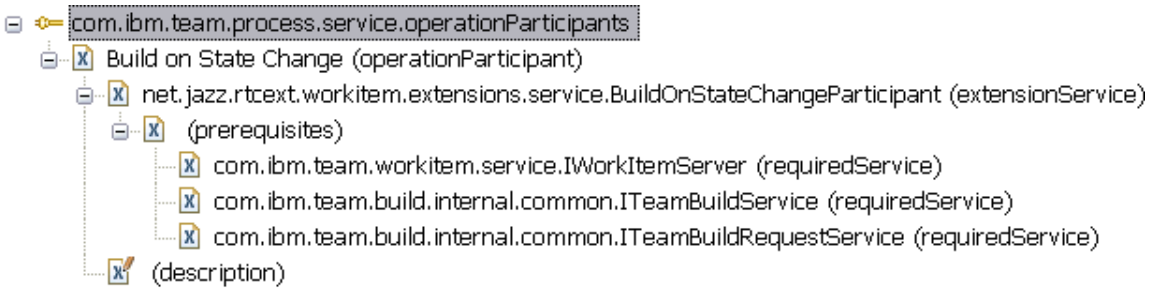
```
/**
 * The component id is used to identify the component to Jazz. It is also
 * used by service definitions to identify which component they belong to.
 */
public static String COMPONENT_ID ="net.jazz.rtcext.workitem.extensions";
```

__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.

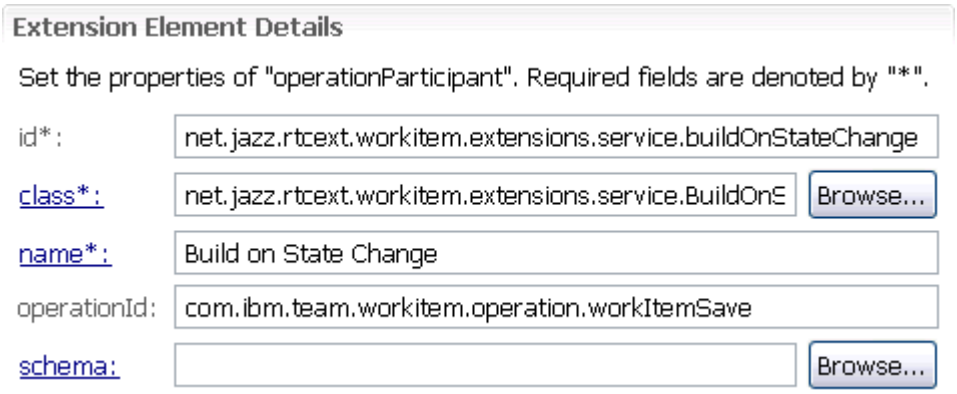
```
/**
 * The extension id is used to identify the operation participant to Jazz.
 * It is also included in instantiations of the participant in process
 * definitions.
 */
public static String EXTENSION_ID =
        "net.jazz.rtcext.workitem.extensions.service.buildOnStateChange";
```

__38. Understanding the service plug-in Eclipse project.

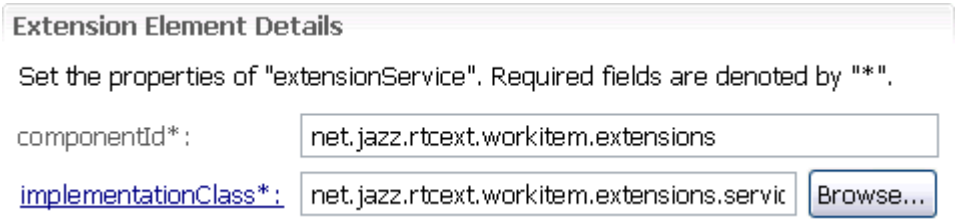
__a. In the **Package Explorer** view, expand the tree for the service project (**net.jazz.rtcext.workitem.extensions.service**) and double click the **plugin.xml** file. Once again, there is a set of standard Eclipse plug-in definitions. Also, the most interesting part is once again on the **Extensions** 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 text in parenthesis on each line is the name of the xml element for that line. The raw xml can be seen on the **plugin.xml** tab of the editor.



__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.



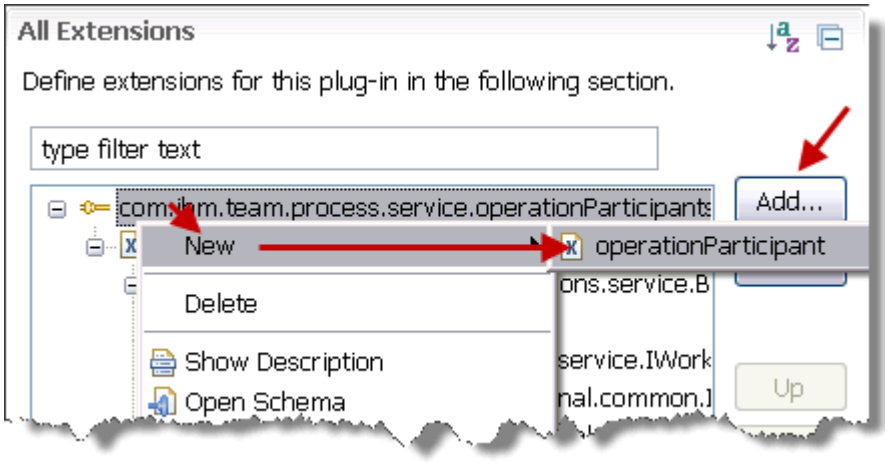
__c. Select the **net.jazz.rtcext.workitem.extensions.service.BuildOnStateChangeParticipant (extensionService)** element on the left and the right side of the editor will look like this. Note that this element is optional. It is only required if the participant will require services from other components. 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.



__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.

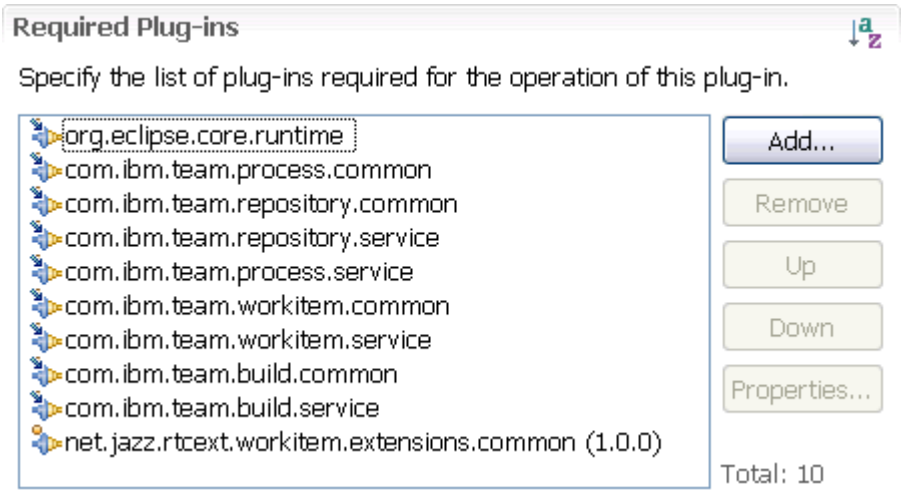
__f. Up to now, all the work you would do to create this definition is possible from this one place using the **Add…** button and the **New >** cascade menu from the various element's pop-up menus.



__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.



__39. Understand the code within the service plug-in Eclipse project

__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. 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 is 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 for 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))) {
```

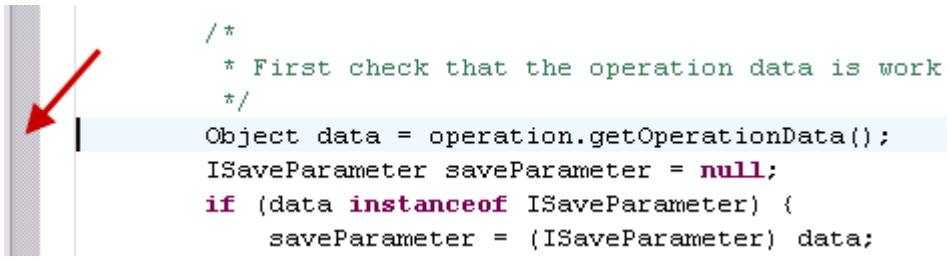
__iii. Is the work item of the type in which the participant is interested? Right now the work item type id is hard coded to the Story type from the Scrum template. You will change that later.

__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. 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.

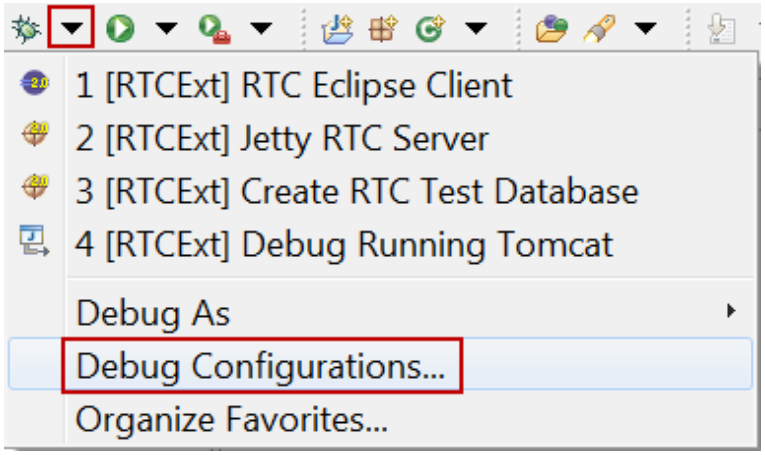
__h. So there you have a pretty simple participant that boils down to a few simple status checks in the run method and four lines of code to request a build. There is one more thing to do before leaving this editor. That is, set a breakpoint at the first line of the run method. You will step through 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.



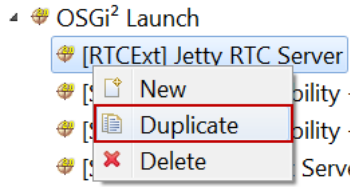
2.2 Launch the Server for Debug Using Jetty

__40. Create the launch configuration.

__a. From the **Debug** toolbar dropdown, select **Debug Configurations…**

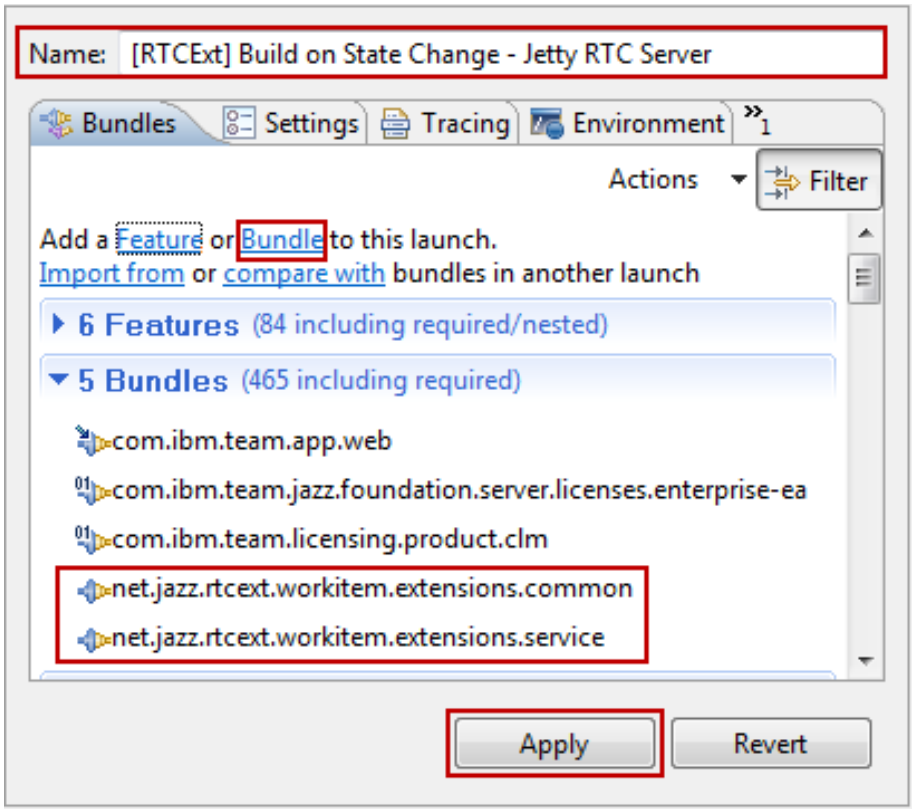


__b. In the **Debug Configurations** dialog, expand the **OSGi² Launch** tree and right click the **[RTCExt] Jetty RTC Server** configuration and then from the popup menu, select **Duplicate**. Note that you are not changing the existing launch but creating a copy of it. You should keep the original 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 `[RTCExt] Build on State Change - Jetty RTC Server`.

__d. Add your participant's two bundles to the configuration. Click on the **Bundle** link and in the **Add Bundle** dialog, type `rtcext` 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.

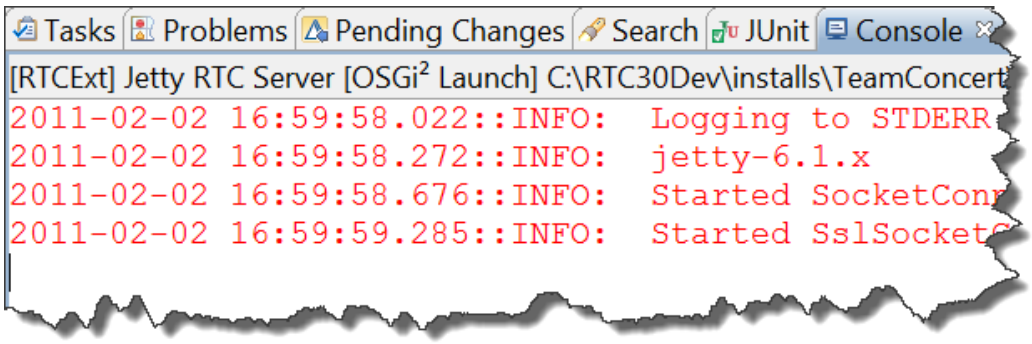


__e. Click **Apply** to save your changes but do not close the dialog.

__41. Launch the server.

__a. Click **Debug** at the bottom of the **Debug Configurations** dialog.

__b. As in lab 1, the **Console** view will show a few log messages indicating that the Jetty server is up and running.

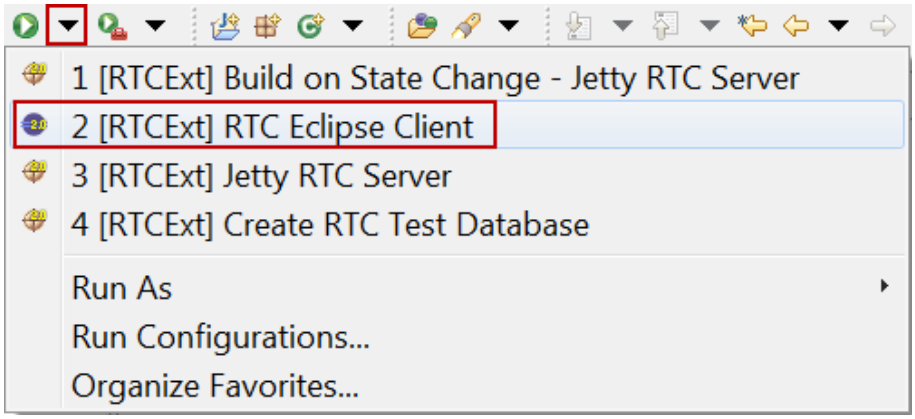


__c. The next time you want to debug this server configuration, you will be able to click a shortcut to it on the dropdown of the **Debug** toolbar icon. You will not need to open the **Debug Configurations** dialog.

2.3 Launch an RTC Client and Connect to the Server

__42. 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.

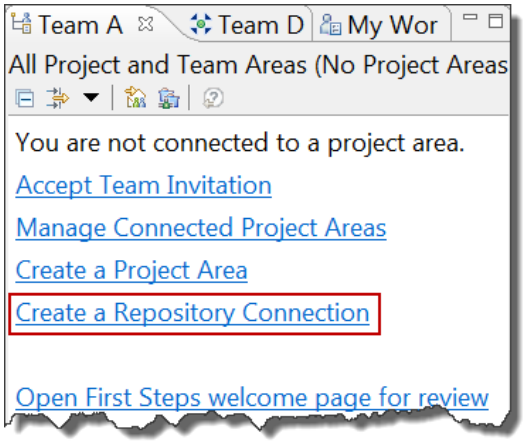


__b. The RTC Eclipse client will start up and should look familiar. If you are prompted to clear the runtime workspace, click **Yes** (you will usually click **No**, but this time start fresh).

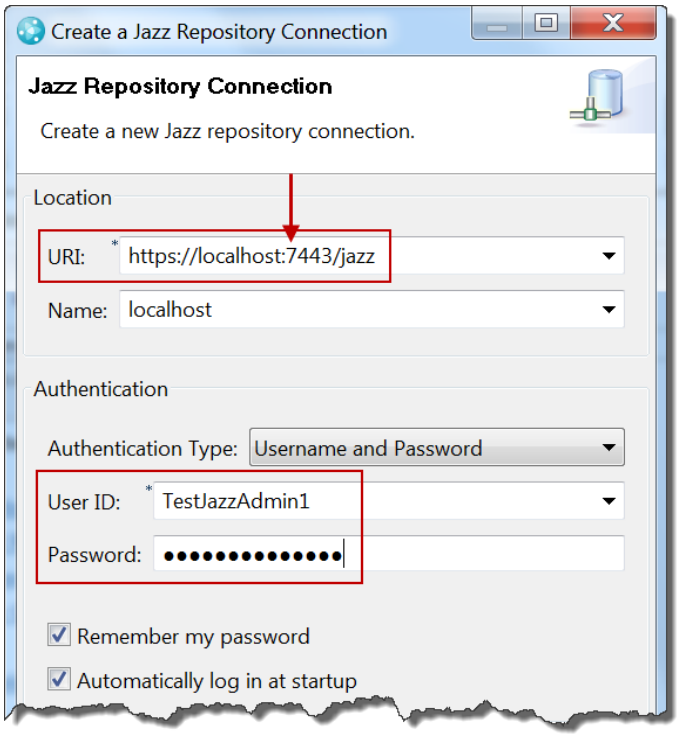
Minimize the **Welcome** screen via this () button near the top or right of the window.

__43. Connect to the debug server.

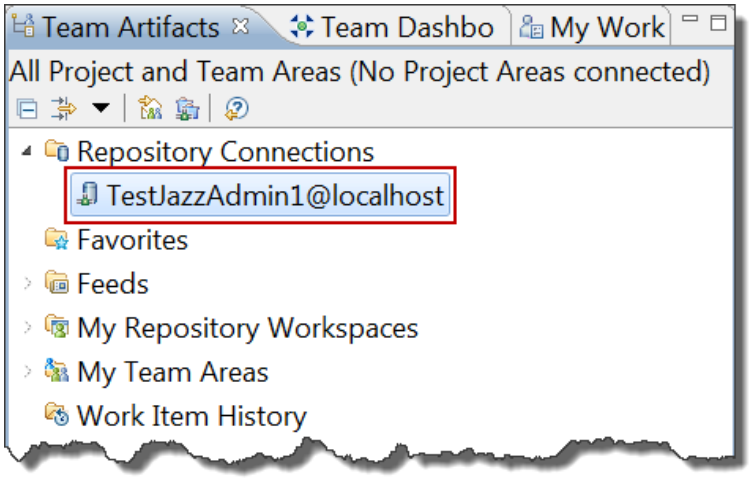
__a. You will be in the **Work Items** perspective and the **Team Artifacts** view will be on the left. In the **Team Artifacts** view, click the **Create a Repository Connection** link.



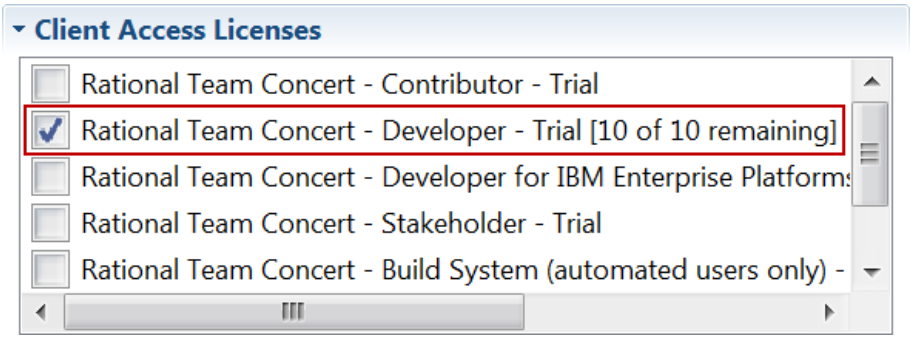
__b. In the **Create a Jazz Repository Connection** wizard, set the **URI** to https://localhost:7443/jazz and the **User ID** and **Password** fields to `TestJazzAdmin1`. Note that it is a '7' and not a '9' in the URI. Then, click **Finish**. Note that "\jazz" is the correct context root and not "\ccm". Recall from lab one that this launch runs the server as one application at the "\jazz" context root and not as separate JTS and CCM applications. This is generally fine for development and you do have the Tomcat server with split applications for final testing (a later lab).



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



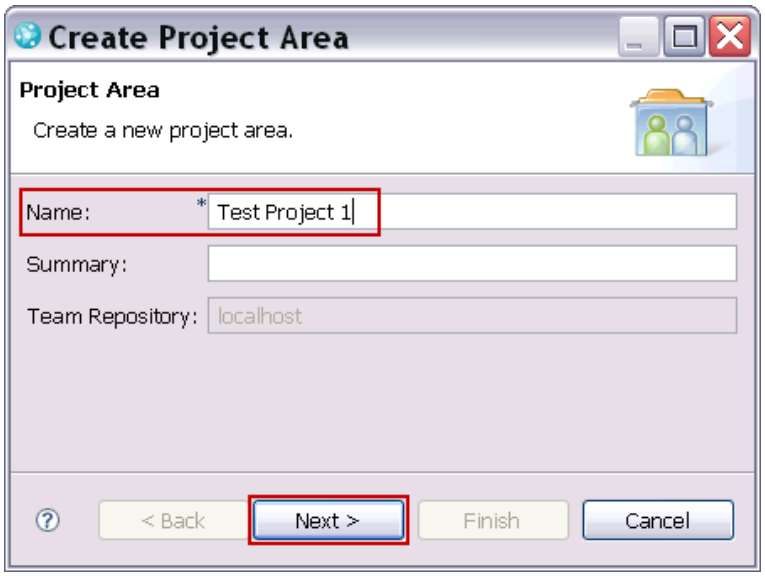
__d. Right click your repository connection and from the pop-up menu select **Open My User Editor**. In the user editor that opens, find the **Client Access Licenses** section at the lower right and make sure the **Rational Team Concert – Developer** checkbox is selected. Save and close the editor. You will use the `TestJazzAdmin1` user id for several operations that require a developer client access license.



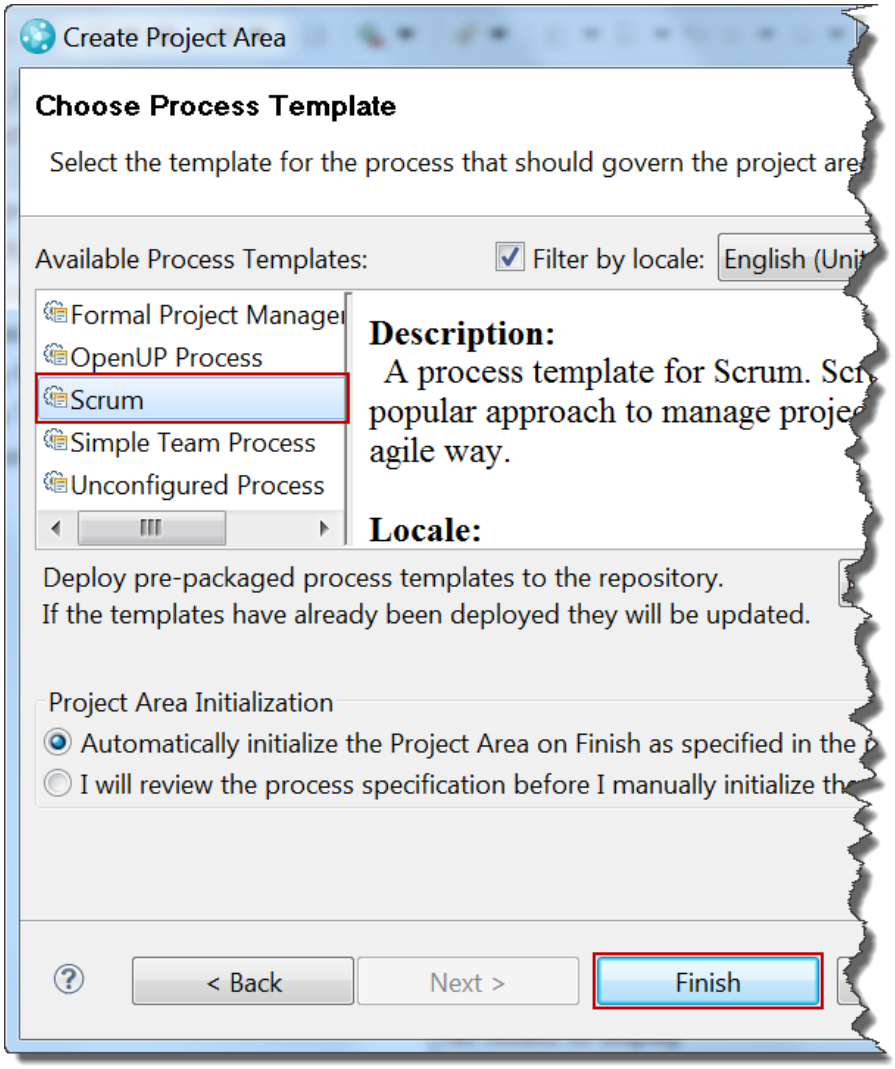
2.4 Edit the Process to Use the Participant

__44. 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**.

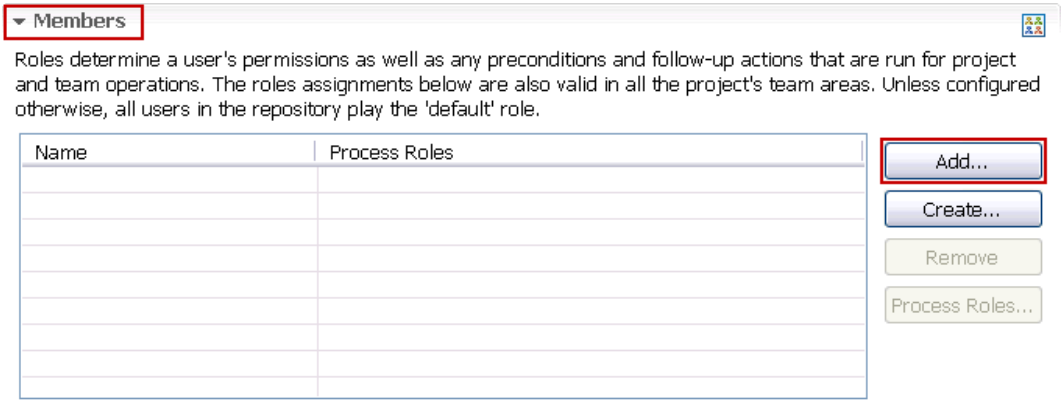


__b. On the second page of the wizard, click the **Deploy Templates** button. This operation may take a bit of time. When it completes, you will be on the next page of the wizard. Select **Scrum** on the left and then click **Finish**. When the operation completes and the project area editor opens, leave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

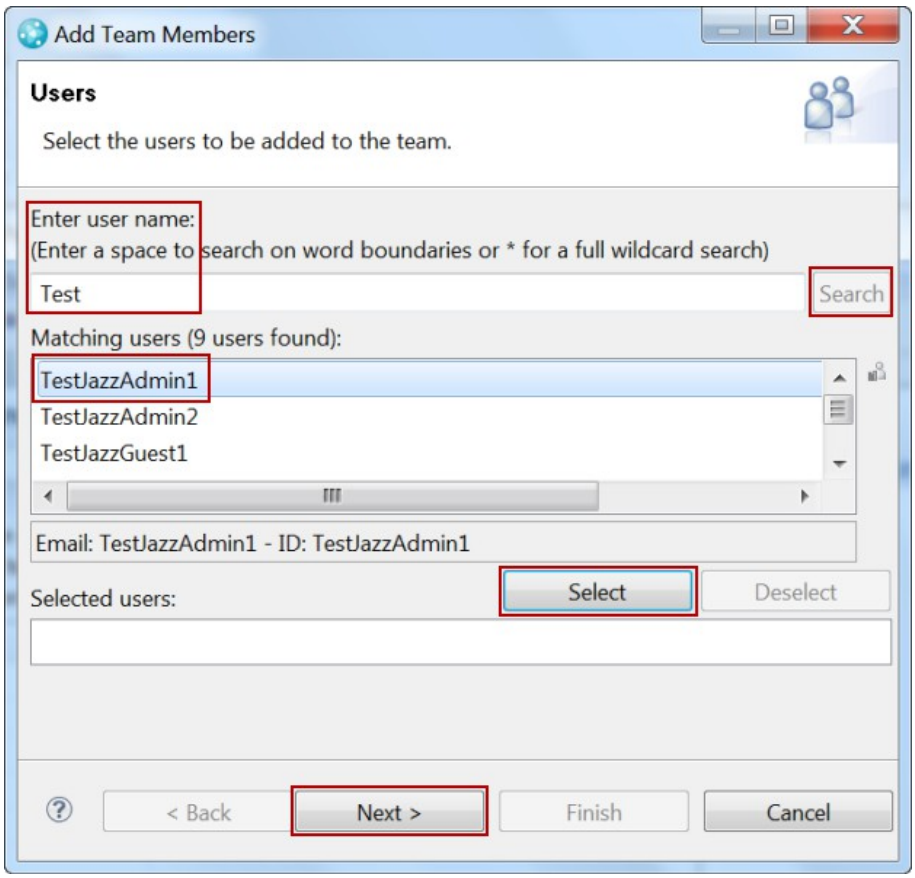


__45. Add `TestJazzAdmin1` as a member of the project area.

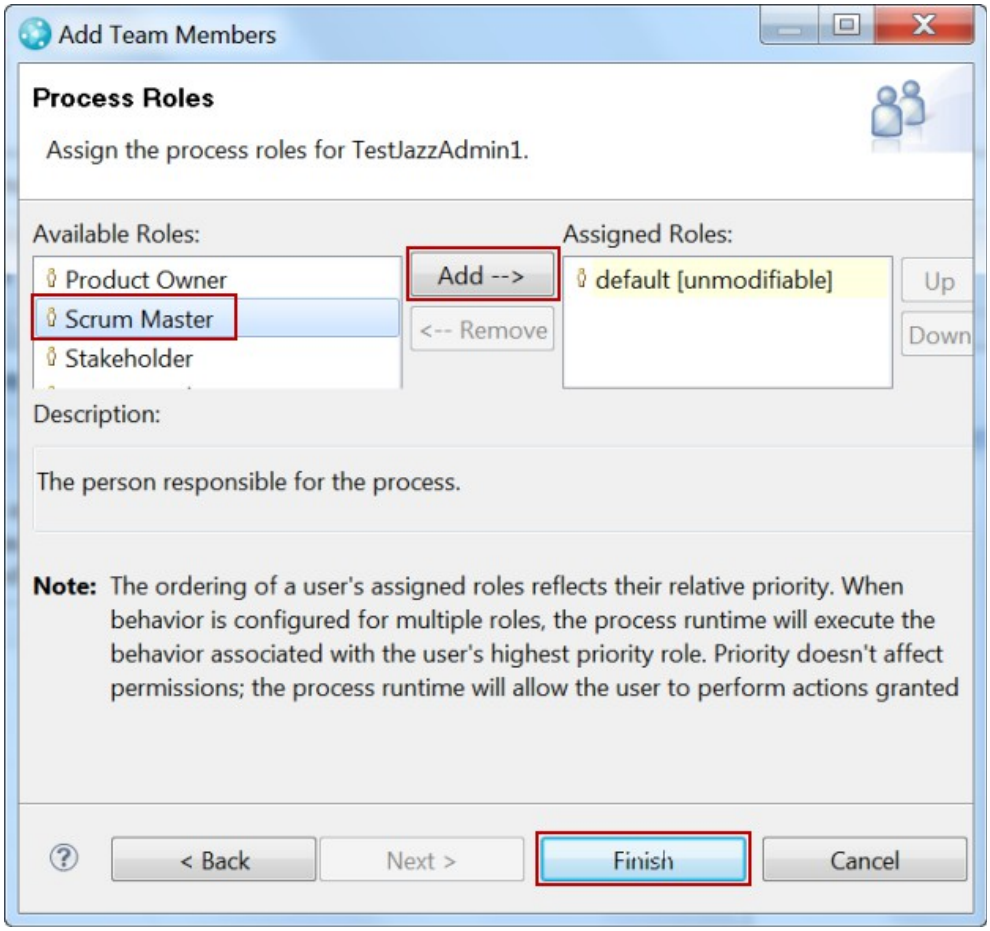
__a. On the **Overview** tab of the project area editor, expand the **Members** section and click **Add…**



__b. In the **Add Team Members** wizard, type `Test` into the **Enter user name** field and then click **Search**. Then, select `TestJazzAdmin1` in the **Matching users** list, click **Select** (moves `TestJazzAdmin1` to **Selected users**) and then click **Next**.



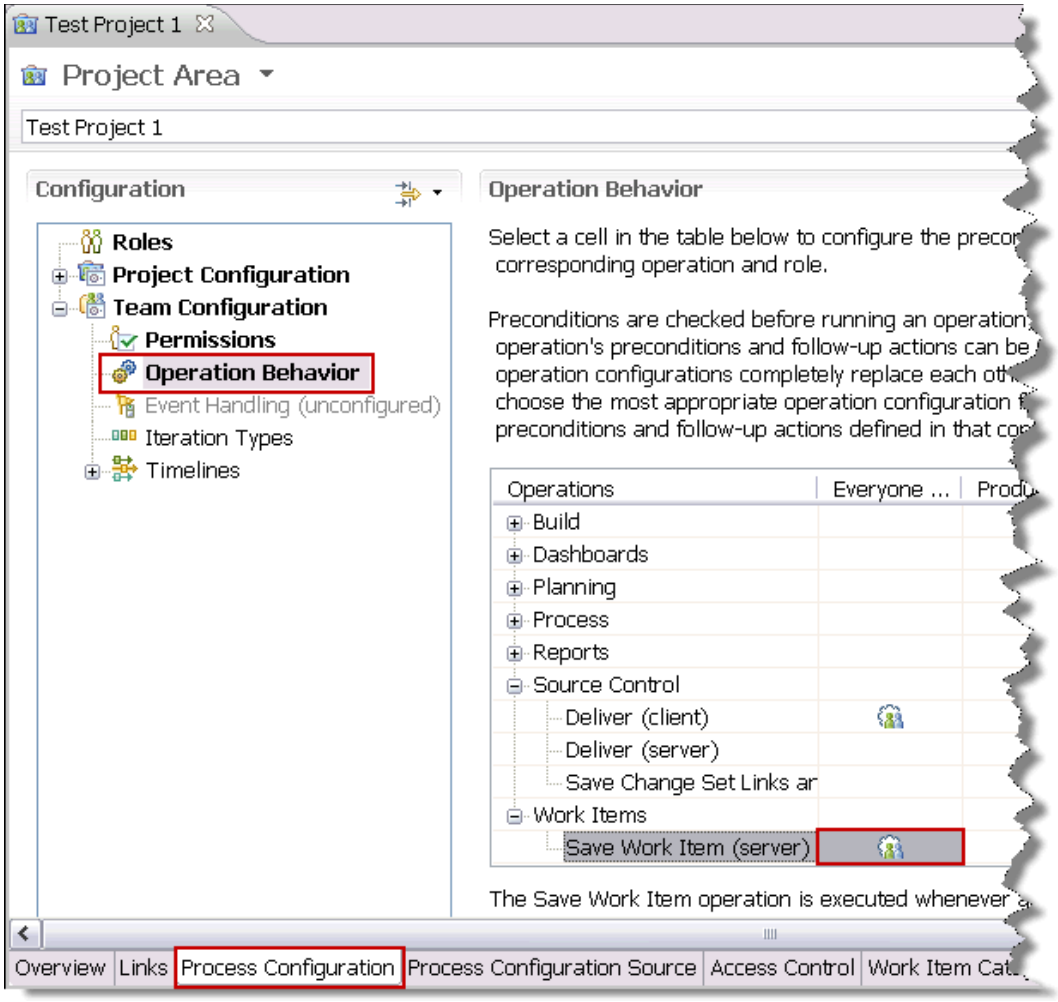
__c. On the second page of the wizard, select **Scrum Master** on the left, click **Add -->** (moves the selection to the right) and then click **Finish**.



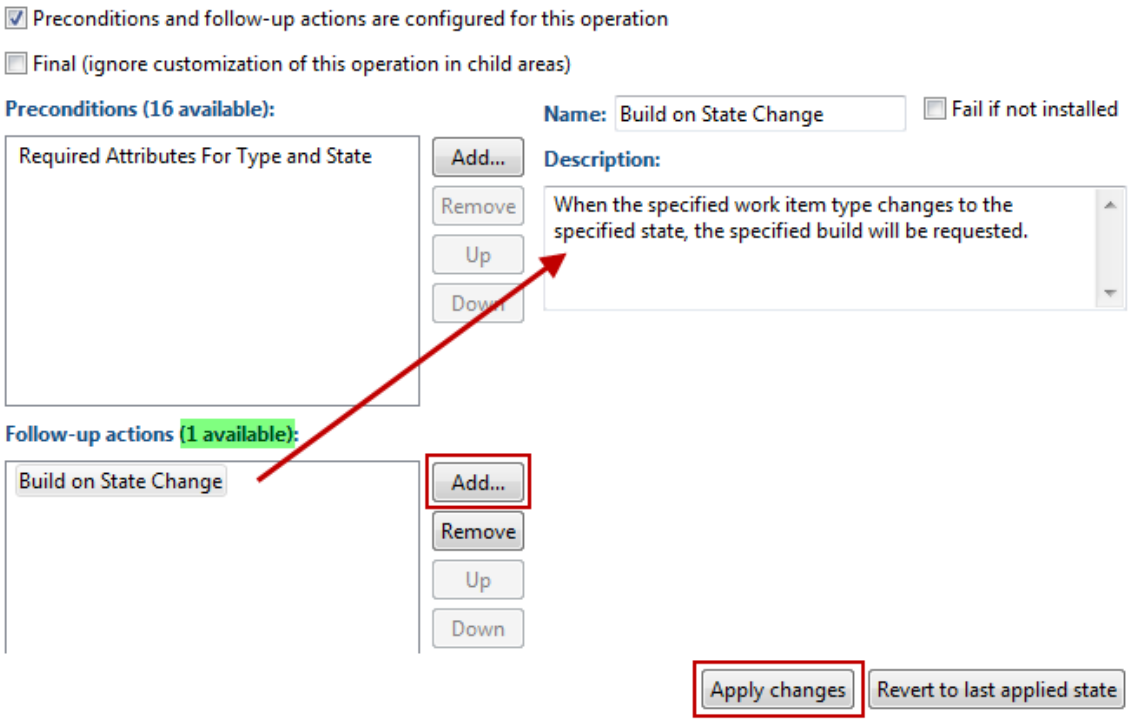
__d. Back on the project area editor's **Overview** page, click **Save** (at the upper right) but leave the editor open for the next step.

__46. 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.



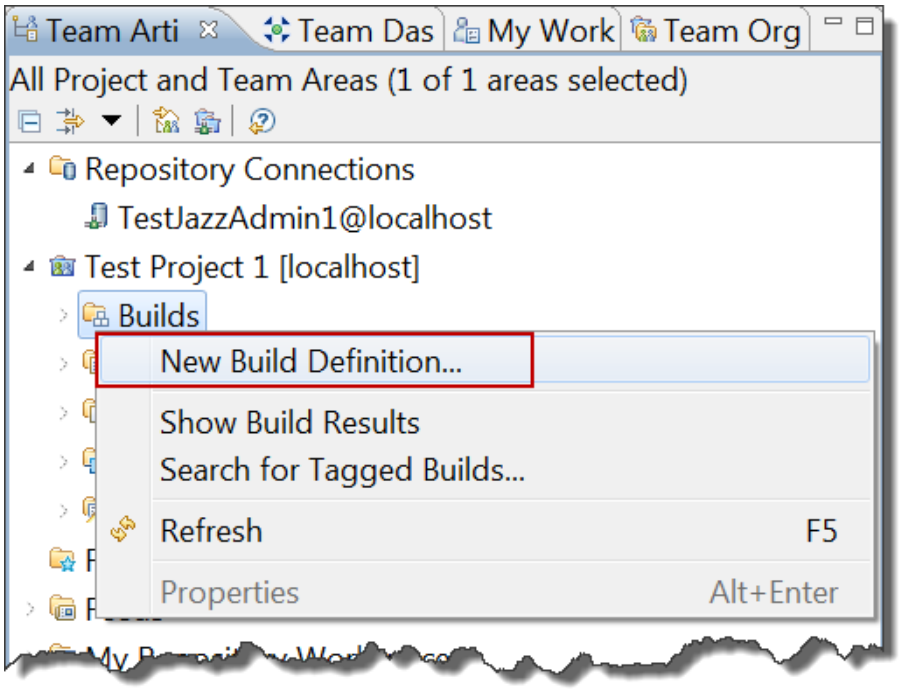
__c. Make sure you have saved your changes, otherwise the next steps will fail.

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

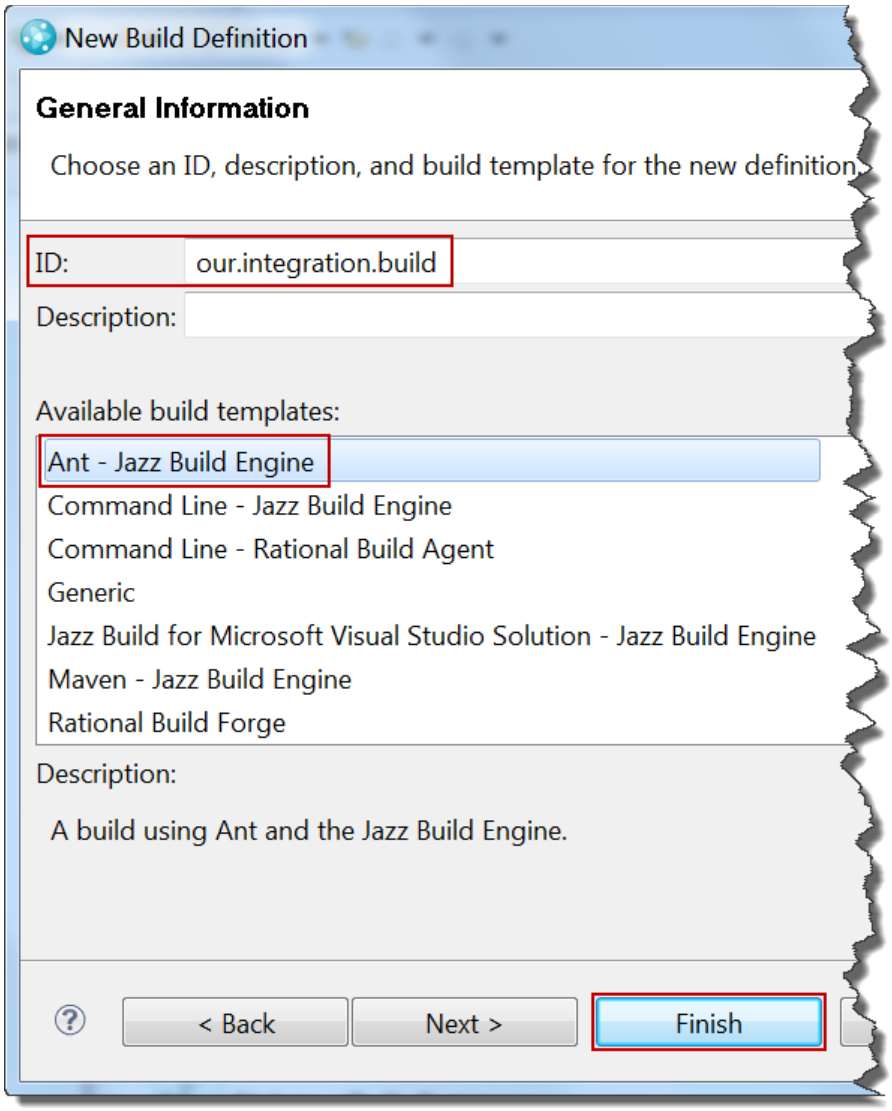
2.5 Trigger the Participant

__47. 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.

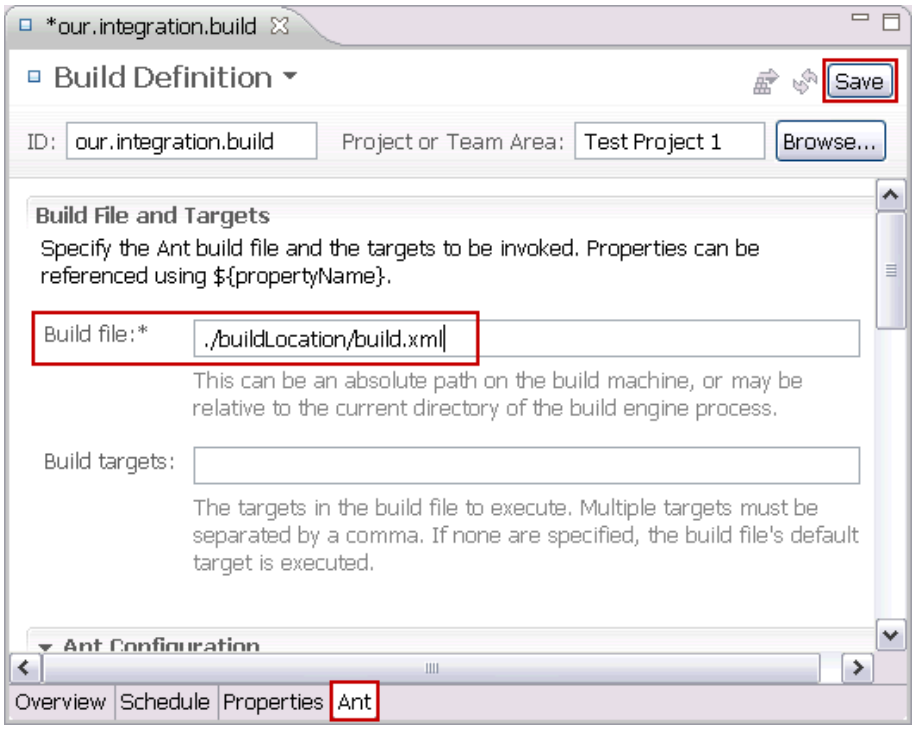
__a. In the **Team Artifacts** view, expand the **Test Project 1** node, right click **Builds** and then click **New Build Definition…**



__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**.



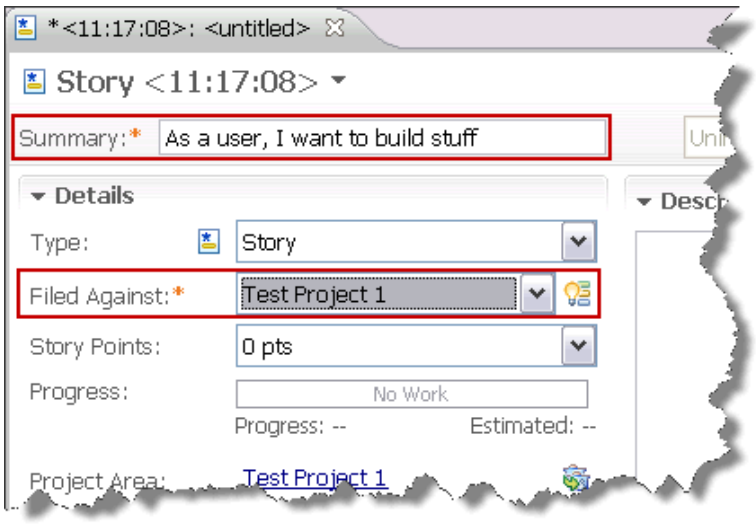
__c. In the build definition editor that opens, switch to the **Ant** tab, and enter a path for the **Build** file and then click **Save**. You may now close the editor. Note that the build file does not exist and any path will work for the current purpose. If you wish, you can use the **Build file** 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.



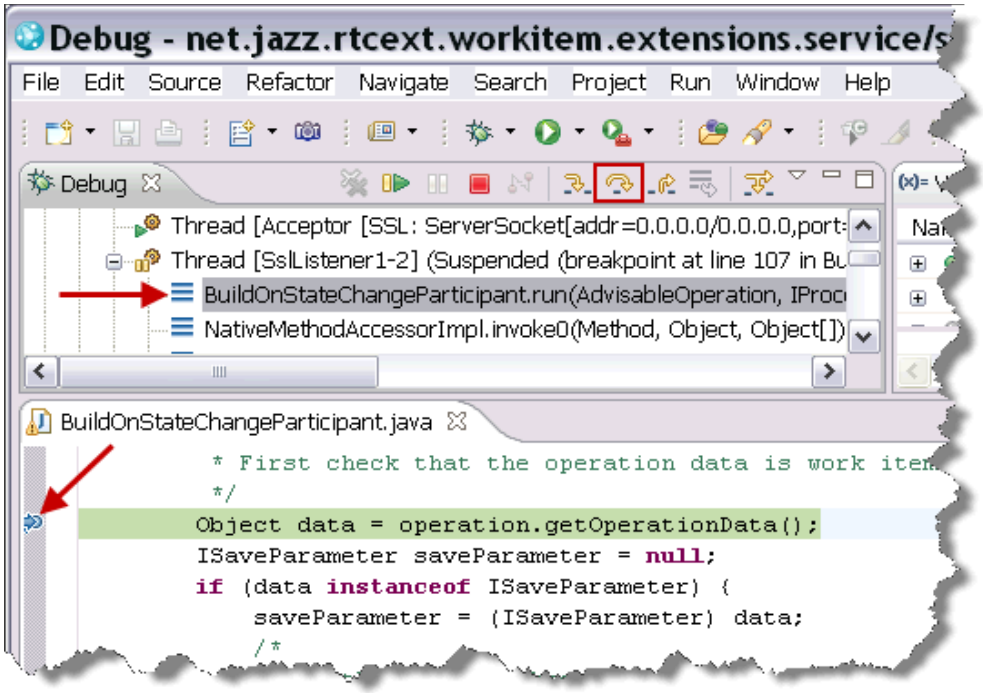
__48. Create a Story work item.

__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.



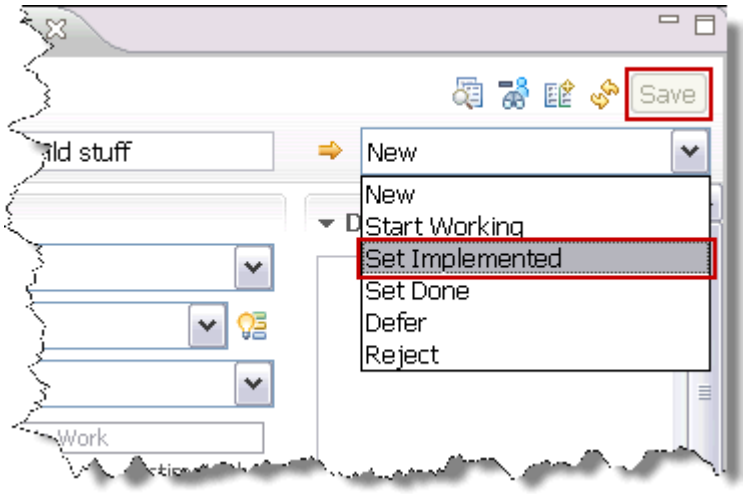
__c. The breakpoint you set earlier is now hit. The RTC Eclipse client in which you were studying the code will now surface (if asked about switching to the debug perspective, click **Yes**). If it does not surface, you probably minimized it earlier. In this case, it will be flashing in the Windows taskbar. 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 ( ).

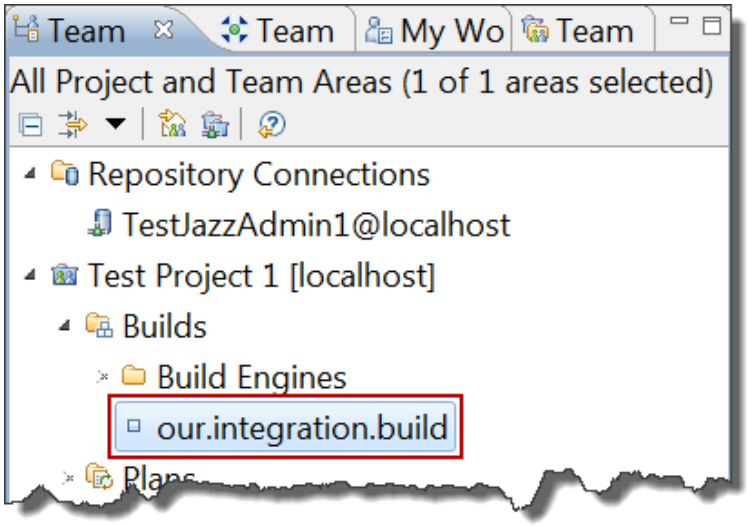
__49. Move the Story to the Implemented state.

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

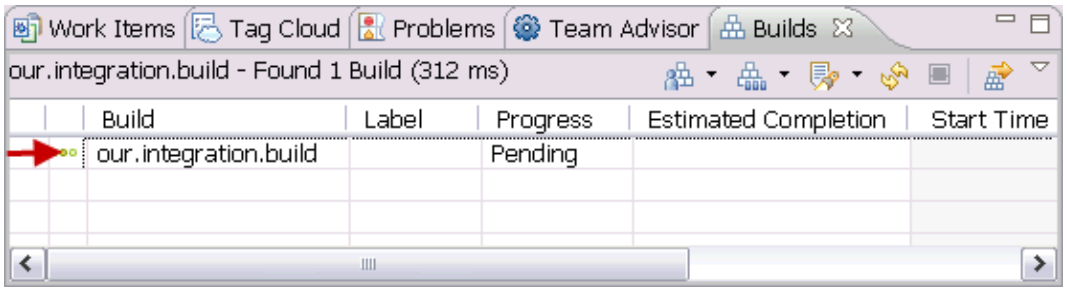


__b. Once again the breakpoint is hit and your debugger surfaces (or you need to click it in the Windows taskbar). Step through the code again. When you get to the call to the build method, use the Step Into button ( ). You can then step through the four lines that request the build and then click the 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.



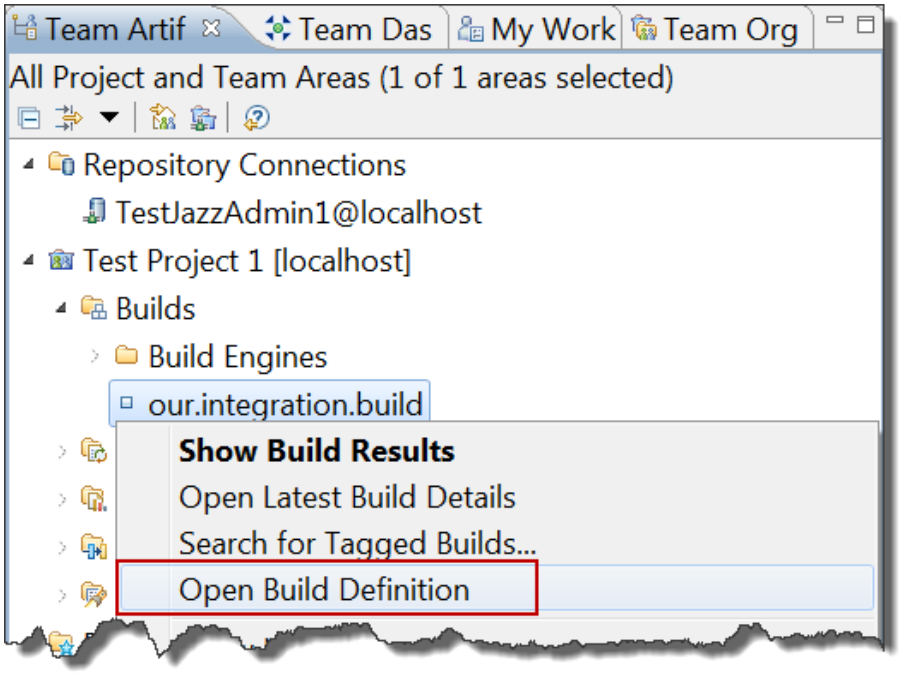
__d. The **Builds** view opens showing the build request the participant just submitted.



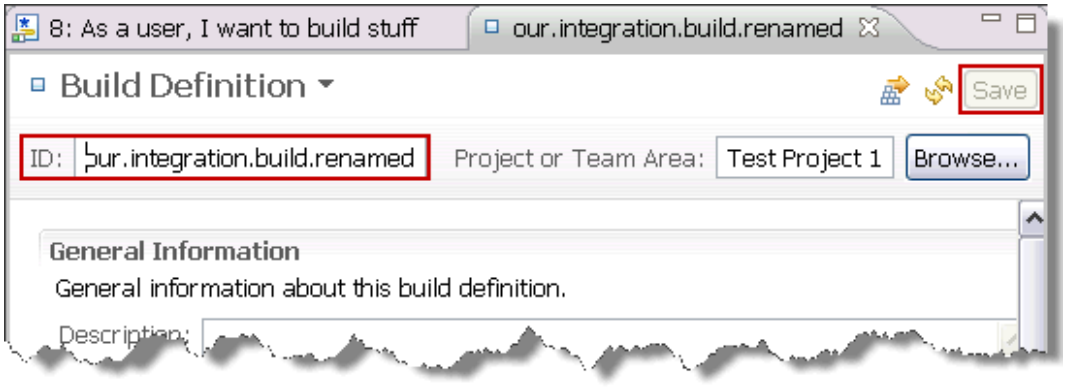
2.6 Rename Build Definition and Try Again

__50. Rename the build definition.

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

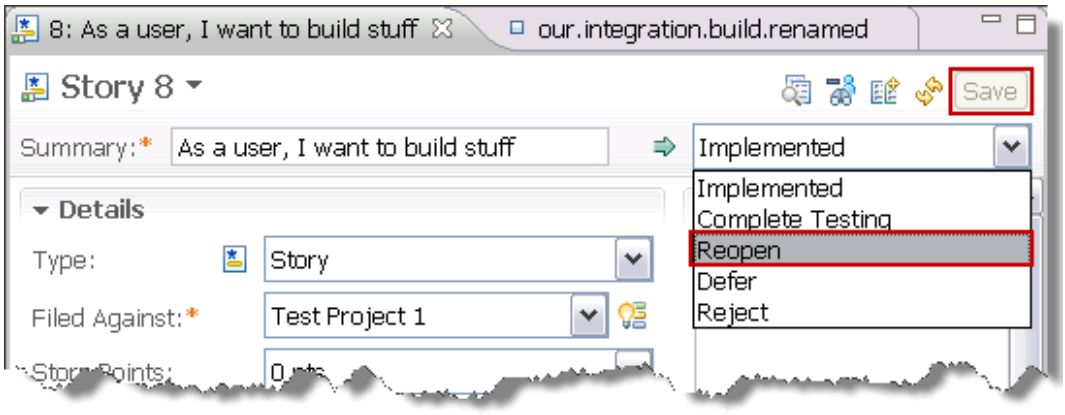


__b. In the build definition editor change the **ID** to `our.integration.build.renamed` and then click **Save**. Do not close the editor as you will want to rename it back soon.

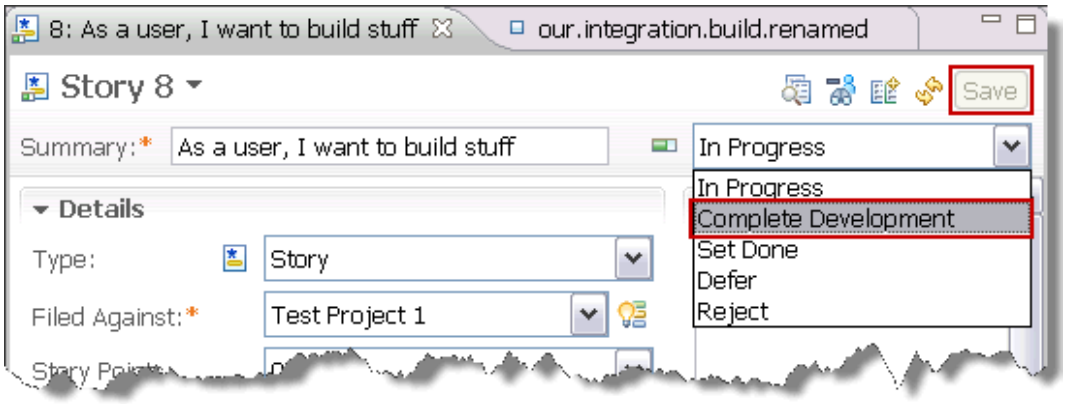


__51. Move the story to the Implemented state again.

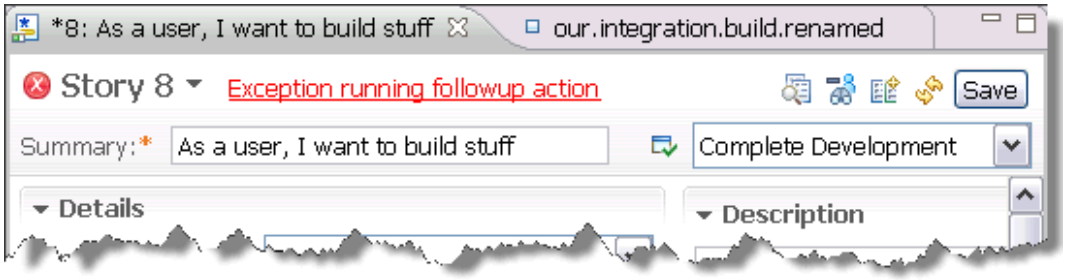
__a. Switch back to the work item editor and select **Reopen** from the state dropdown and then click Save. When the debugger surfaces, just click the resume button ( ). You are not to the interesting bit yet.



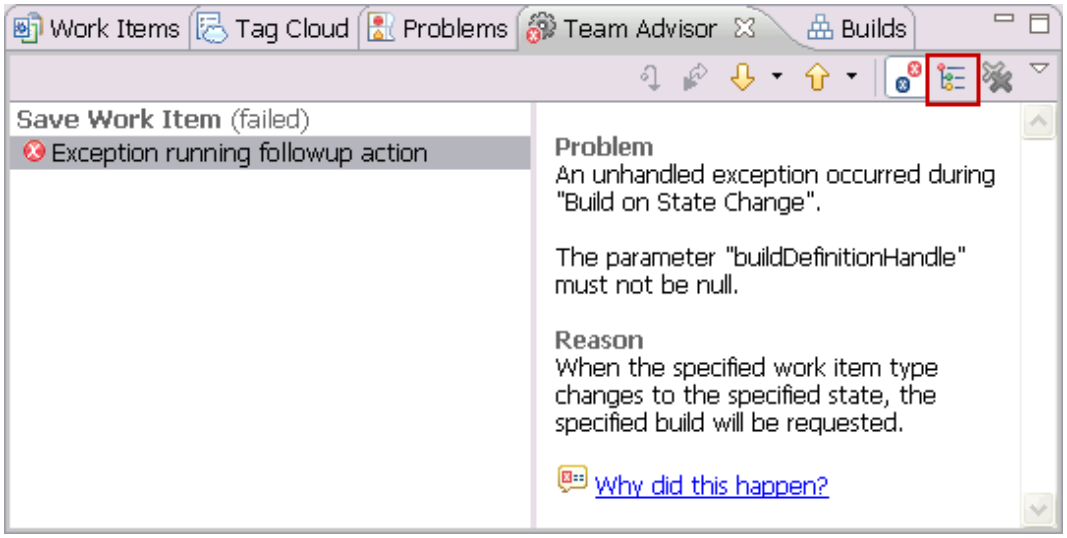
__b. Again in the work item editor, select **Complete Development** from the same dropdown and click **Save** again.



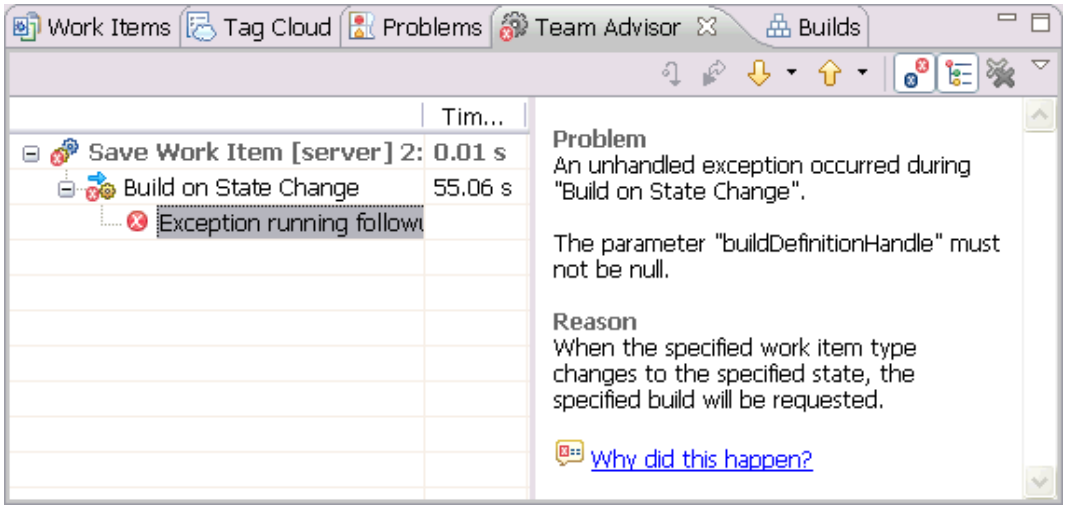
__c. This time, when the debugger surfaces, use the step over button to get to the build method call and then use the step into button. Step through the build method and note the major difference this time. The call to get the build definition returns null and the request of the build throws an exception. 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.



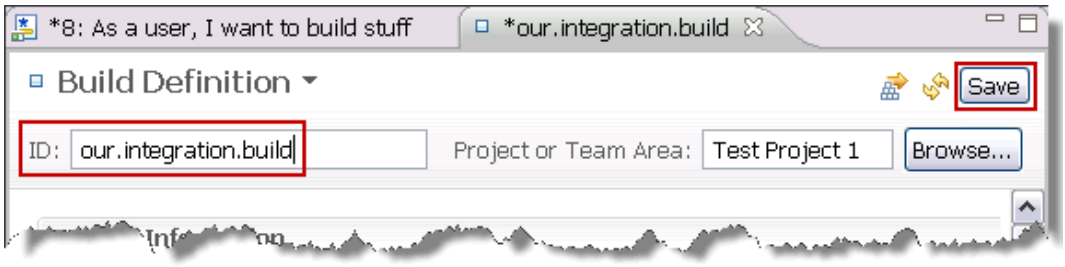
__d. The **Team Advisor** view appears with more information on the error. Click the **Show Detail Tree** button.



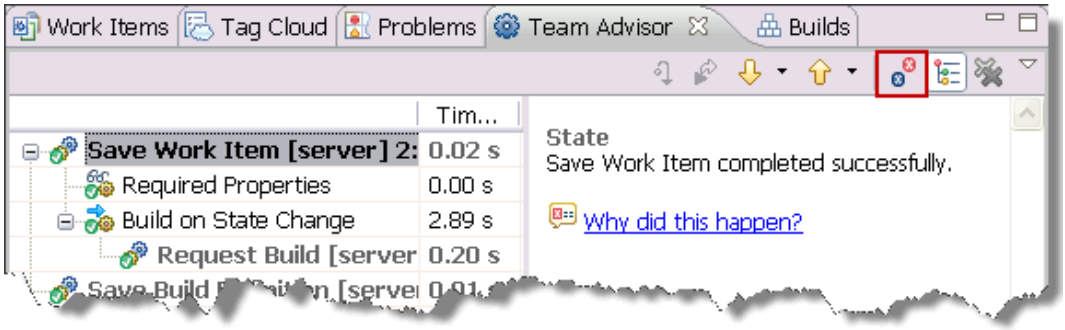
__e. The left side of the view changes to show the structure of the error condition. Click the nodes on the left to see what information is available. It is clear that better information would be helpful. For example, a messages stating that the participant was looking for a particular build definition 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**.



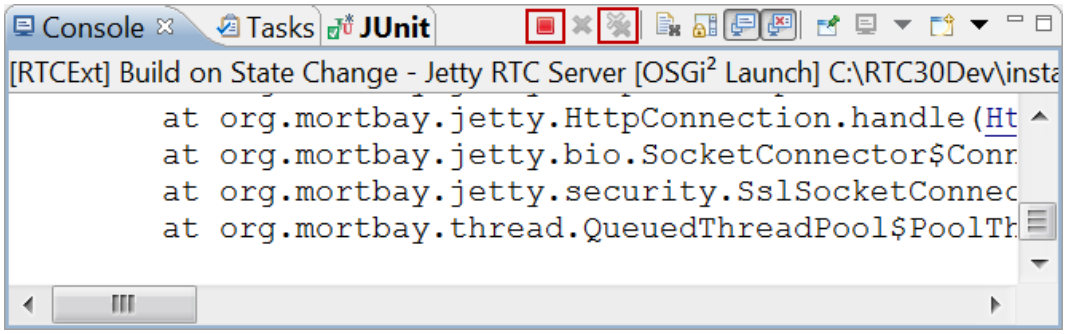
__g. Switch back to the work item editor and click **Save**. When the debugger surfaces, you can step into the build method again or just hit resume. Once you do resume, the work item save should complete okay. Return to the work item editor to confirm this. If you go to the **Team Advisor** view and 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.



__52. Close down the launched client and server.

__a. Close the launched RTC Eclipse client where you were working with the Story and build definition (logged in as `TestJazzAdmin1`).

__b. Back in the original RTC Eclipse client, go to the **Console** view and click the **Remove All Terminated Launches** icon (to remove the console for the client and surface the server console) and then 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:\RTC40Dev\installs\JazzTeamServer\server\server.startup.bat`).

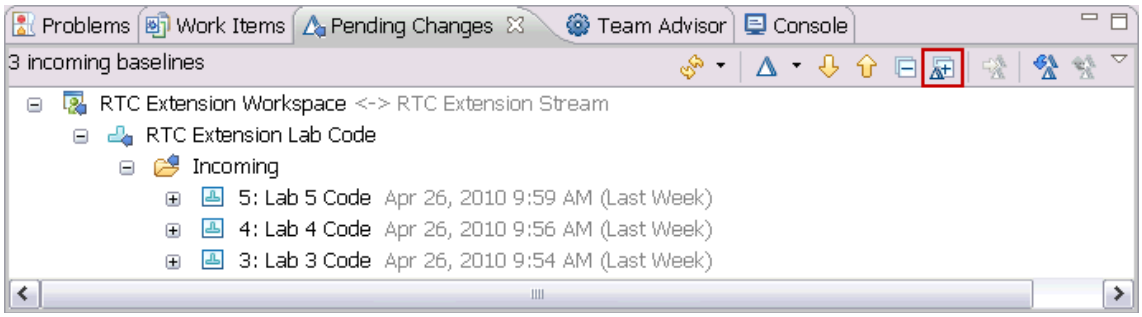
3.1 Understanding Error Handling Code

__1. If your RTC development environment is not open, navigate to `C:\RTC40Dev\installs\TeamConcert\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.

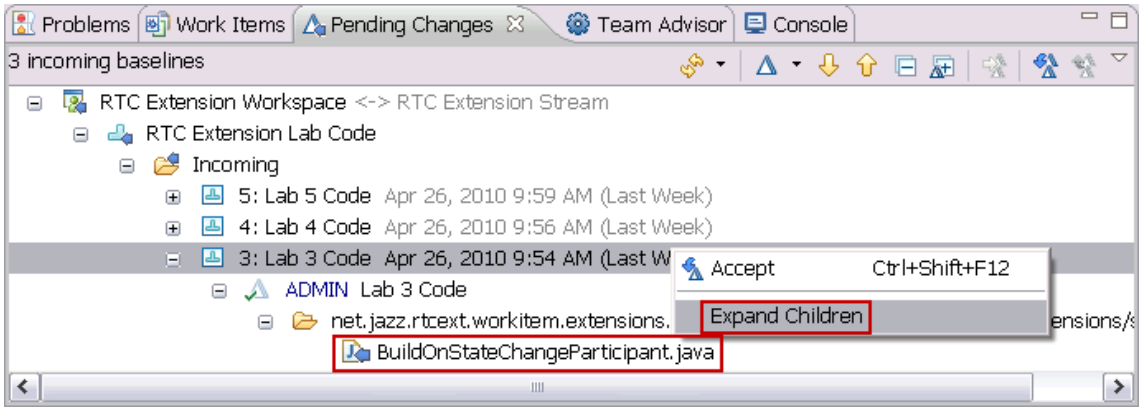
__2. If it is already open, you may need to log back into the server. Recall, that you had to shut the server down earlier and this should be your first interaction with it again since then. Go to the **Team Artifacts** view and check the status of your repository connection.

__3. Browse and load the Lab 3 code.

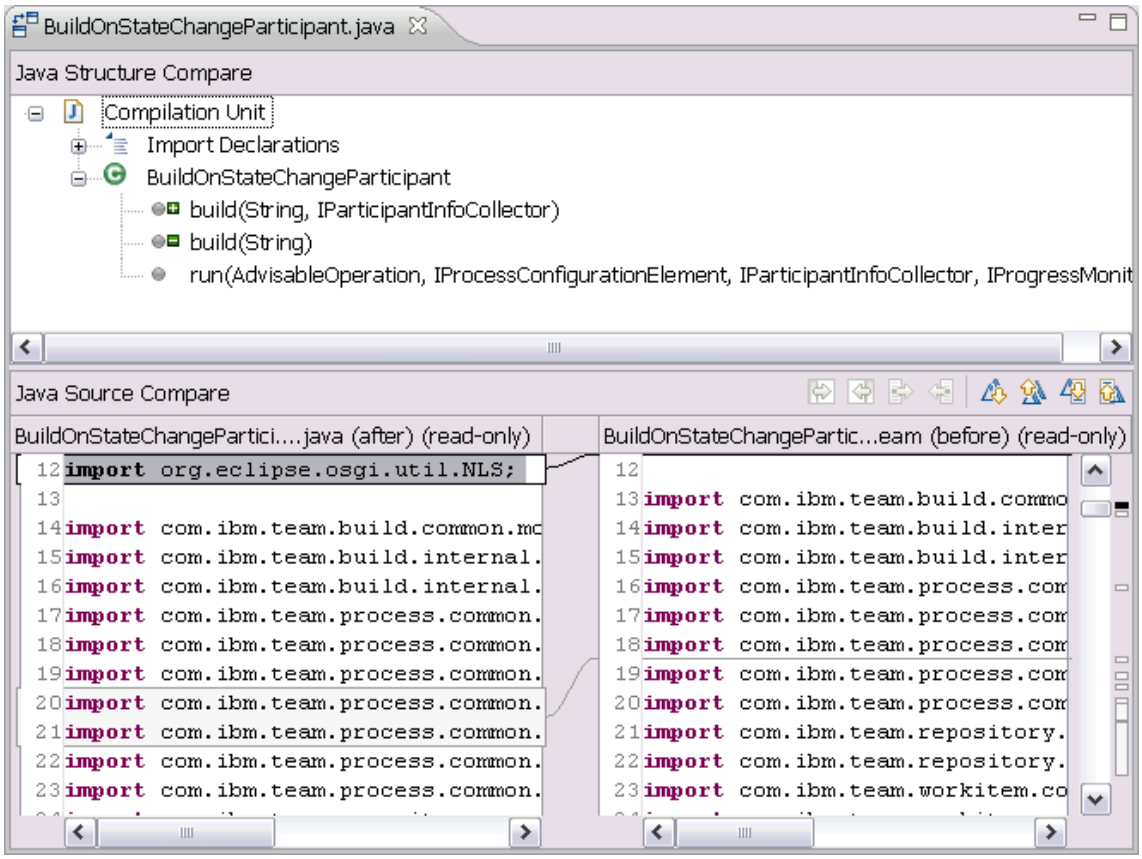
__a. In the **Pending Changes** view, click the **Expand to Change sets** icon. This will show 3 incoming baselines as shown here.



__b. Right click the **Lab 3 Code** baseline 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.

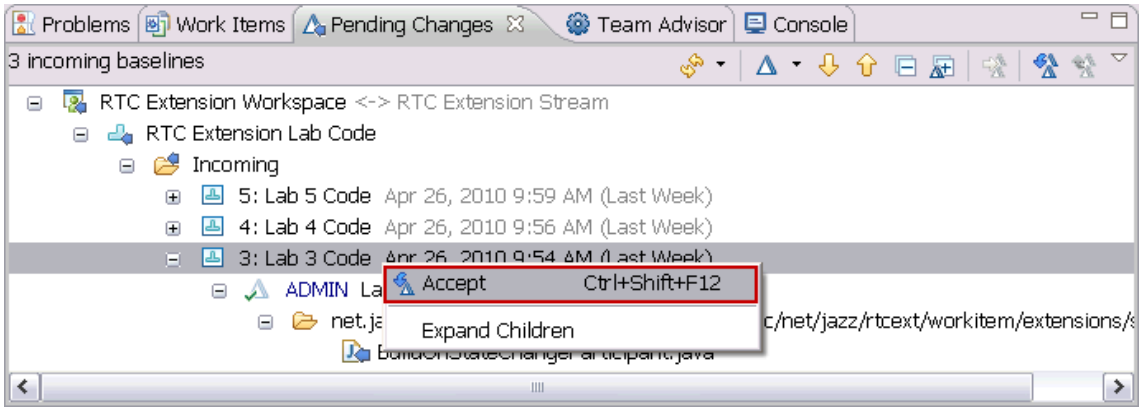
__i. The collector parameter to the run method is now passed through to the build method where it will be used.

__ii. The build method now checks for several error conditions in this order.

- Can the build definition be found?
- 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.



__4. 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 doubled 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 doubled 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.

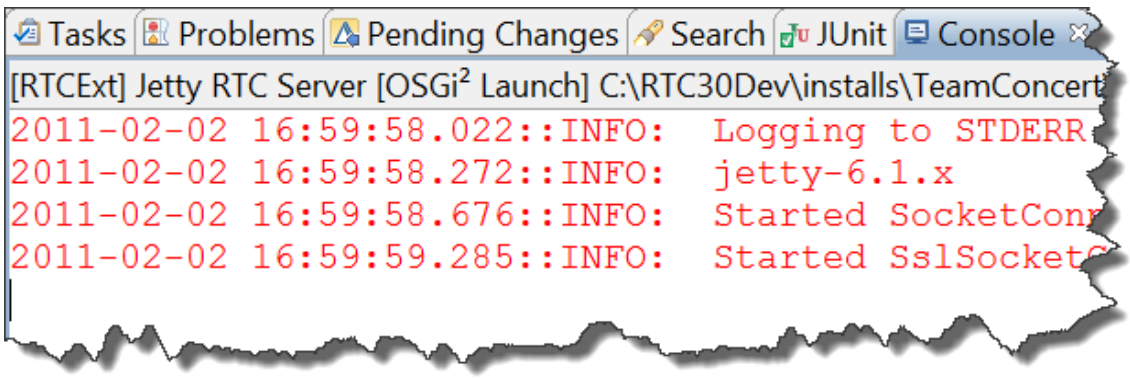
```
String description = NLS
              .bind("A new build request for build definition ''{0}'' was submitted.",
                     buildId);
IReportInfo info = collector.createInfo("Build request successful", description);
collector.addInfo(info);
```

3.2 Launch the Server for Debug Using Jetty

__5. Use the existing launch configuration from lab 2.

__a. From the **Debug** toolbar dropdown ( ) in the toolbar, select **[RTCExt] Build on State Change - Jetty RTC Server**.

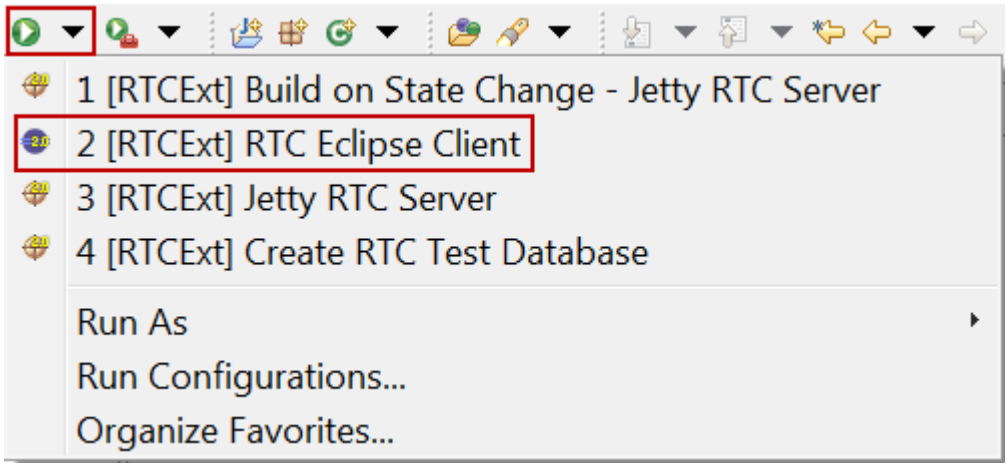
__b. As in lab 1, the **Console** view will show a few log messages indicating that the Jetty server is up and running.



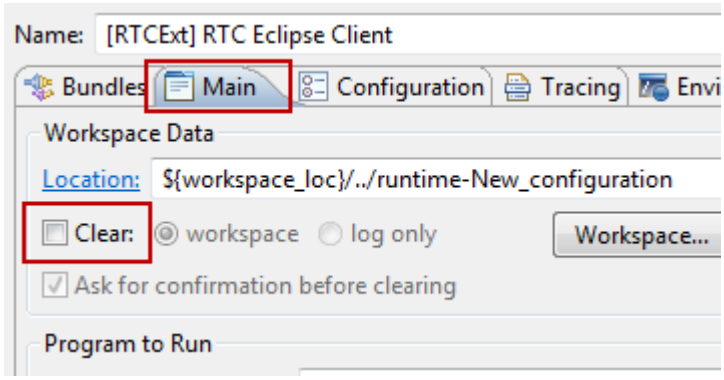
3.3 Launch an RTC Client and Connect to the Server

__6. Launch the RTC Client.

__a. From the dropdown menu of the **Run** toolbar icon, select **[RTCExt] RTC Eclipse 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. If prompted do not clear the runtime workspace. You will probably answer no for this question for the rest of this workshop. You can turn off the prompt by editing the launch configuration.



__c. 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

__7. Find the Story work item used in lab 2 (it is probably number 7) e.g. in the work item history and move it out of the Implemented state (via the **Reopen** action) or create a new story.

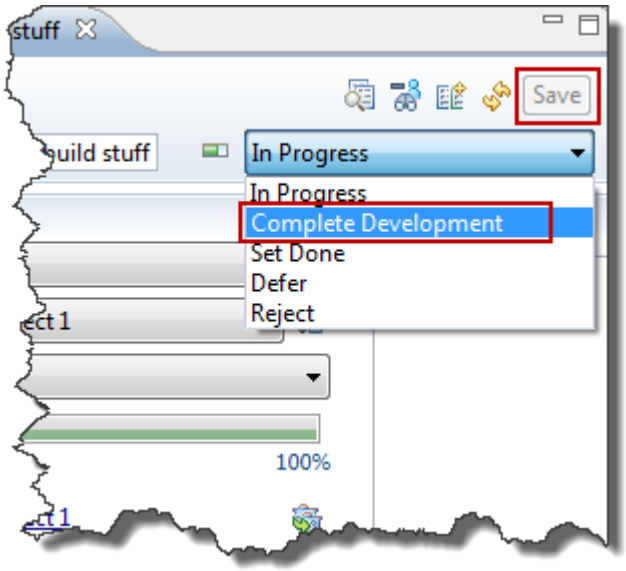
__a. Either of these will cause the breakpoint you set earlier to trigger. If it does not trigger, check if the breakpoint is set to to correct line of code. If necessary remove the old break points and add a valid one. And change the state of the story back. The RTC Eclipse client in which you were 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 ( ).

__8. Move the Story to the Implemented state.

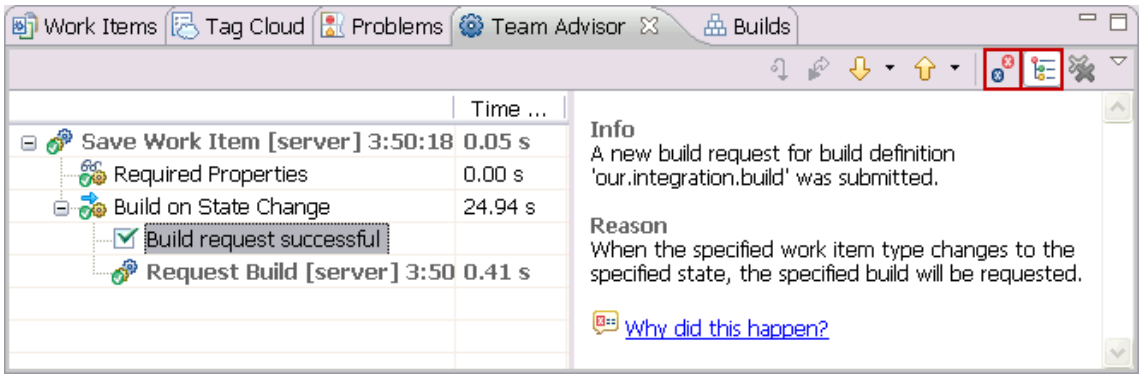
__a. At the upper right portion of the work item editor, select **Set Implemented** or **Complete Development** (depends on which workflow state the story is currently in) and then click **Save**.



__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 ( ).

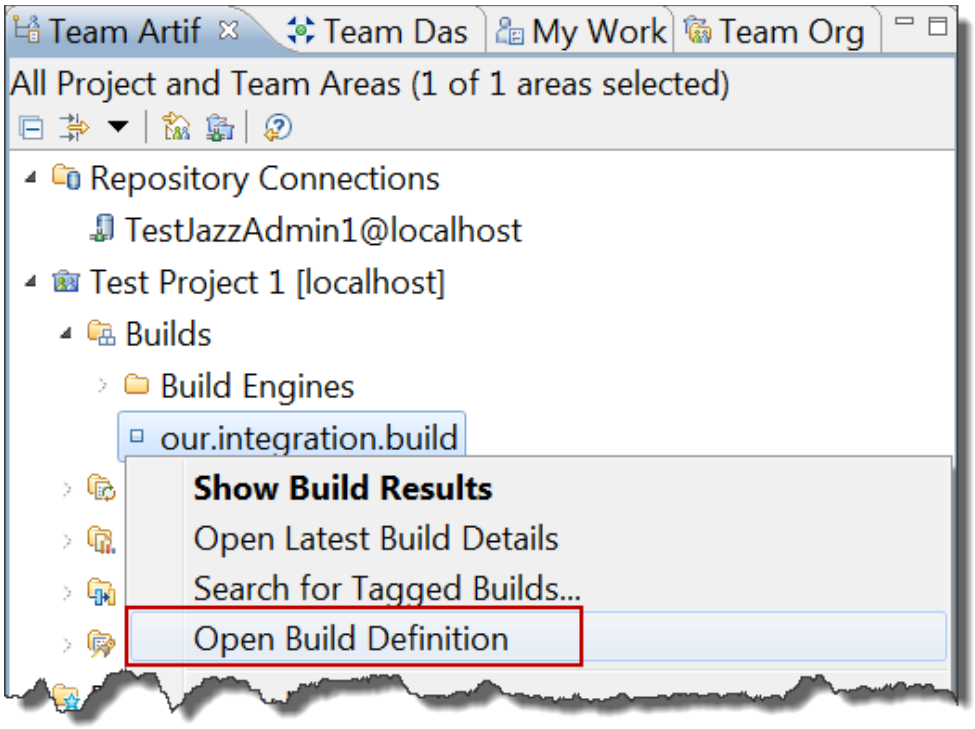
__d. If you go to the **Team Advisor** view and check to make sure the **Show Failures Only** filter is off and **Show Detail Tree** is on (see highlight below), you can browse the results of this successful operation. Also, if you double click **our.integration.build** in the **Team Artifacts** view, the **Builds** view will show a new pending build request.



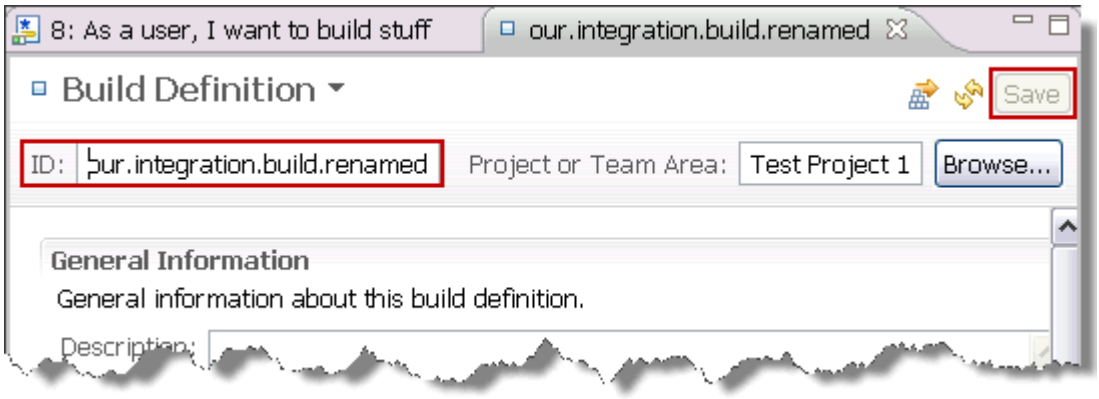
3.5 Rename Build Definition and Try Again

__9. Rename the build definition.

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

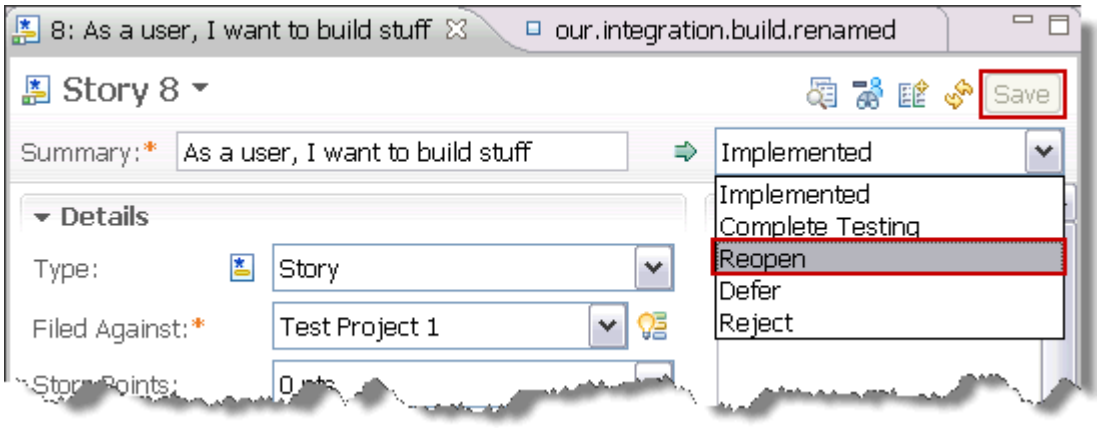


__b. In the build definition editor change the **ID** to `our.integration.build.renamed` and then click **Save**. Do not close the editor as you will want to rename it back soon.

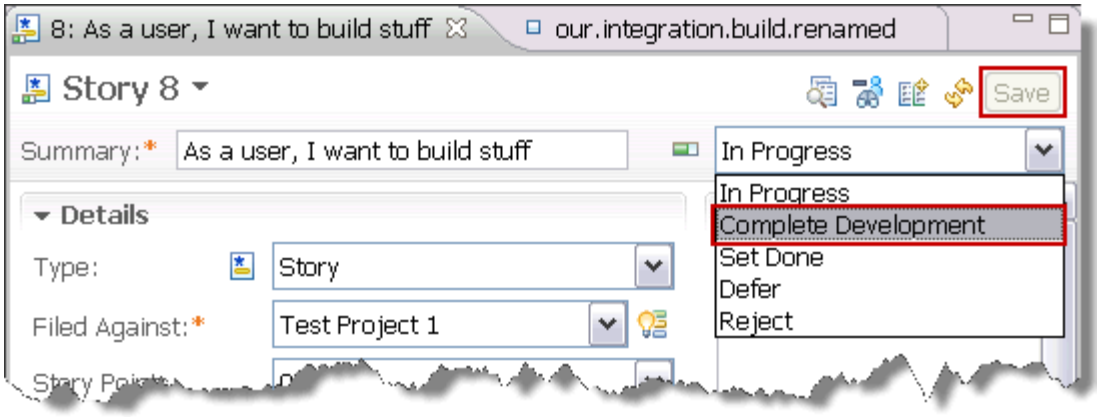


__10. Move the story to the Implemented state again.

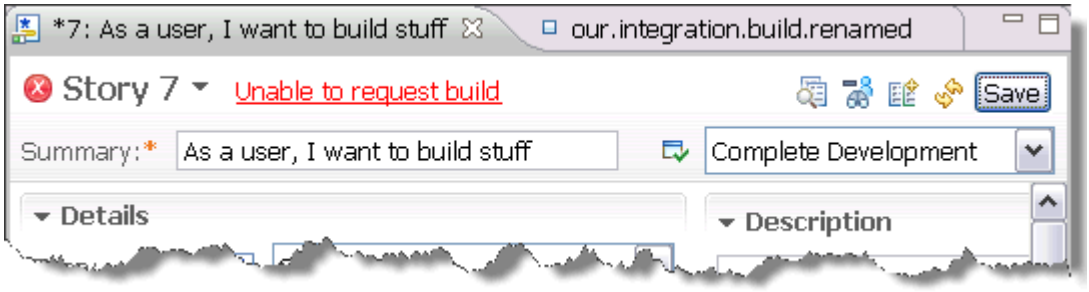
__a. Switch back to the work item editor and select **Reopen** from the state dropdown and then click **Save**. When the debugger surfaces, just click the resume button ( ). You are not to the interesting bit yet.



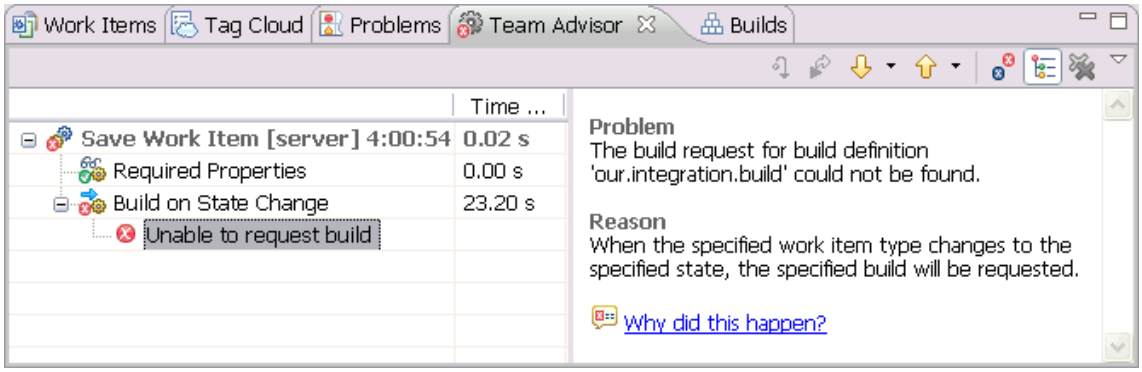
__b. Again in the work item editor, select **Complete Development** from the same dropdown and click **Save** again.



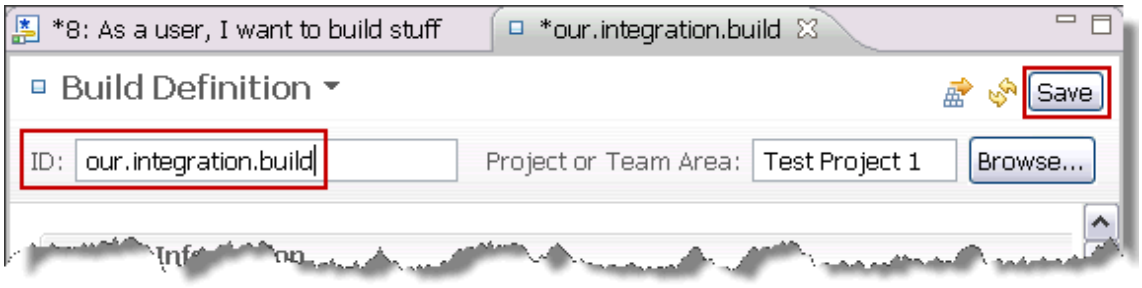
__c. This time, when the debugger surfaces, use the step over button to get to the build method call and then use the step into button. Step through the build method and note the major difference this time. The call to get the build definition returns null, but this time a null pointer exception is not 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**.

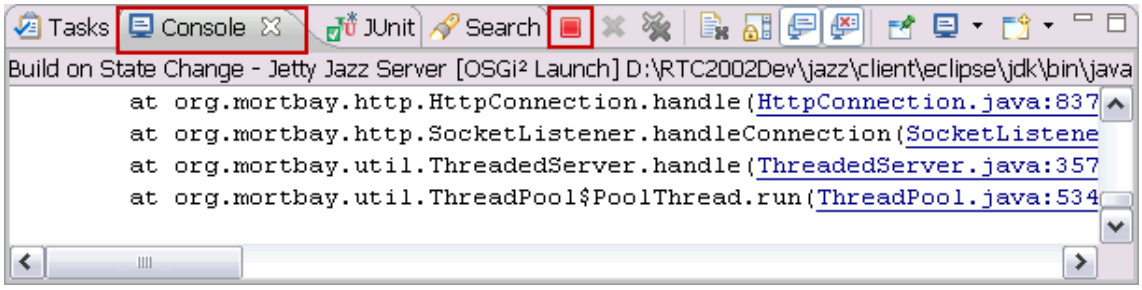


__f. Switch back to the work item editor and click **Save**. When the debugger surfaces, you can step into the build method again or just hit resume. Once you do resume, the work item save should complete okay. Return to the work item editor to confirm this. If you go to the **Team Advisor** view and 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.

__11. Close down the launched client and server.

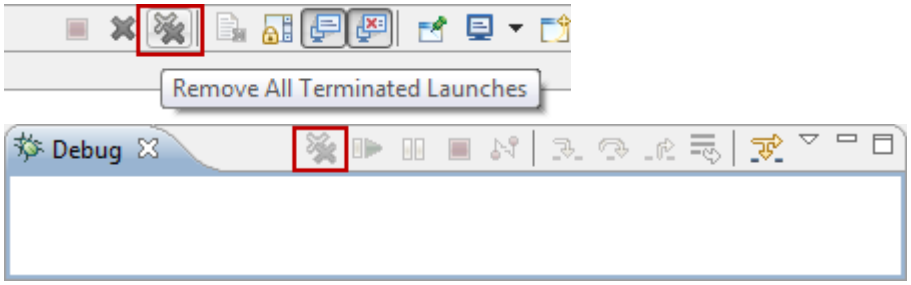
__a. Close the RTC Eclipse client where you were working with the Story and build definition.

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



__c. If there are many launches it is easy to miss to stop one. This can cause conflicts when starting a new launch.

To make sure to get not confused about which launch is still running, use the button **Remove All Terminated Launches** in the **Console** or the **Debug** window and check the **Debug** window is empty.



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!

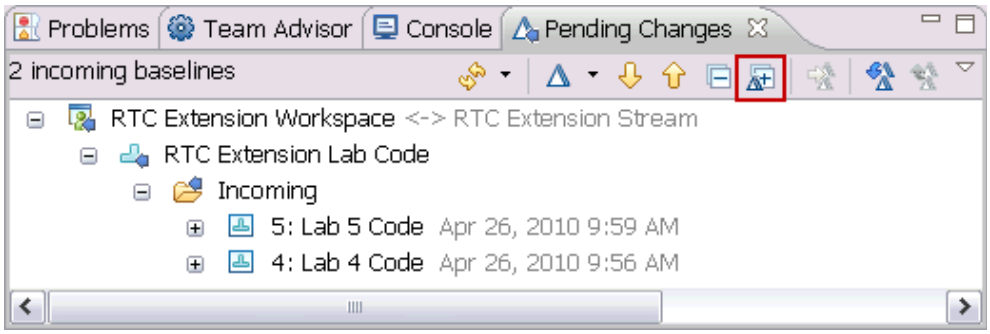
If your RTC server is not running, start it now (`C:\RTC40Dev\installs\JazzTeamServer\server\server.startup.bat`).

4.1 Understanding Parameterization

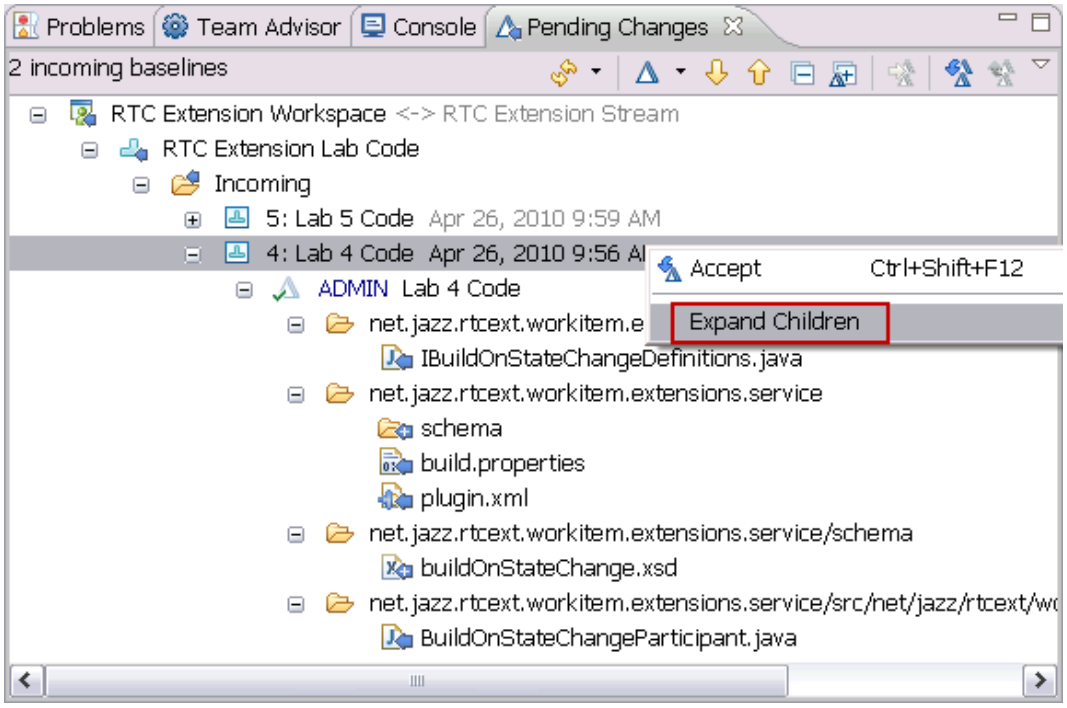
__1. If your RTC development environment is not open, navigate to `C:\RTC40Dev\installs\TeamConcert\jazz\client\eclipse` in the Windows explorer and double click **eclipse.exe**. If prompted to select an Eclipse workspace, select the same one you used in lab two. If the **Plug-in Development** perspective 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.

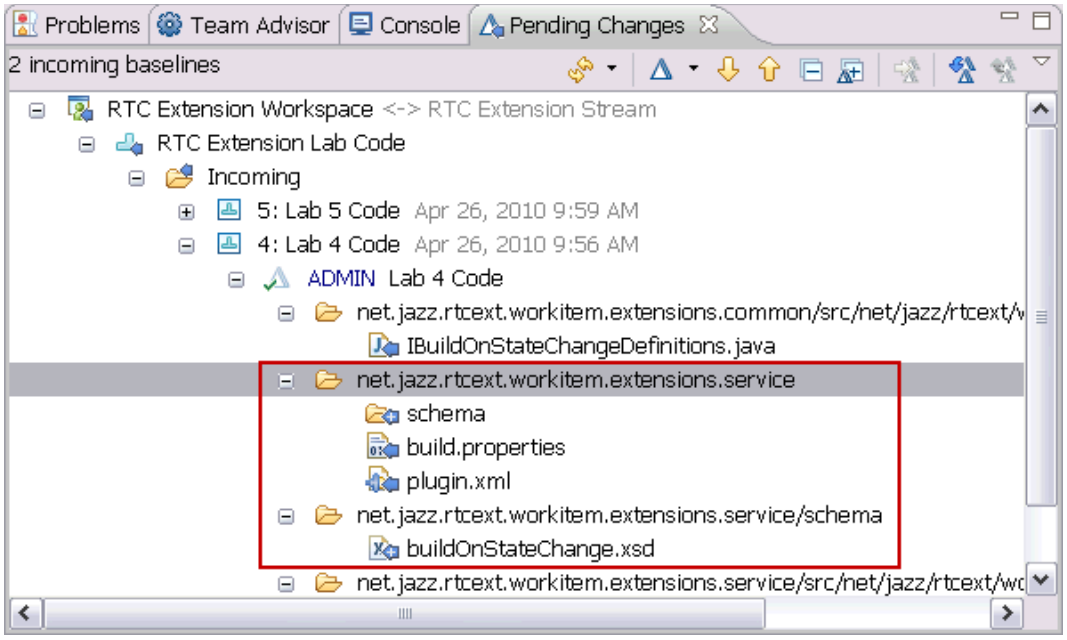


__b. Right click the **Lab 4 Code** baseline under the **RTC Extension Workspace** node, and then click the **Expand Children** action. This will reveal all the changes made for lab 4. As you can see there are quite a few more changes in this lab.

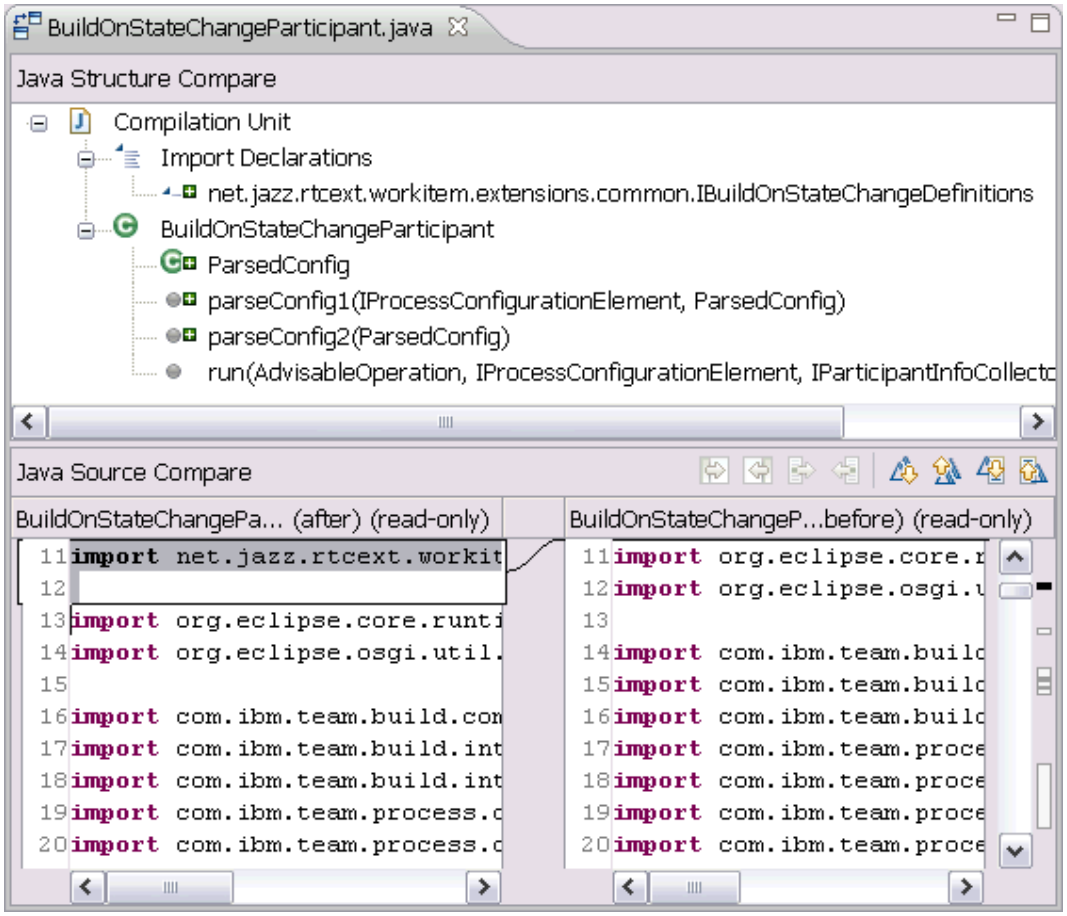


__c. Double click the first changed file, **IBuildOnStateChangeDefinitions.java** to open a comparison editor. You may want to double click the tab of the opened editor to maximize it. A set of constants have been added to this file. Most of them define elements of the XML schema that will be used to 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.



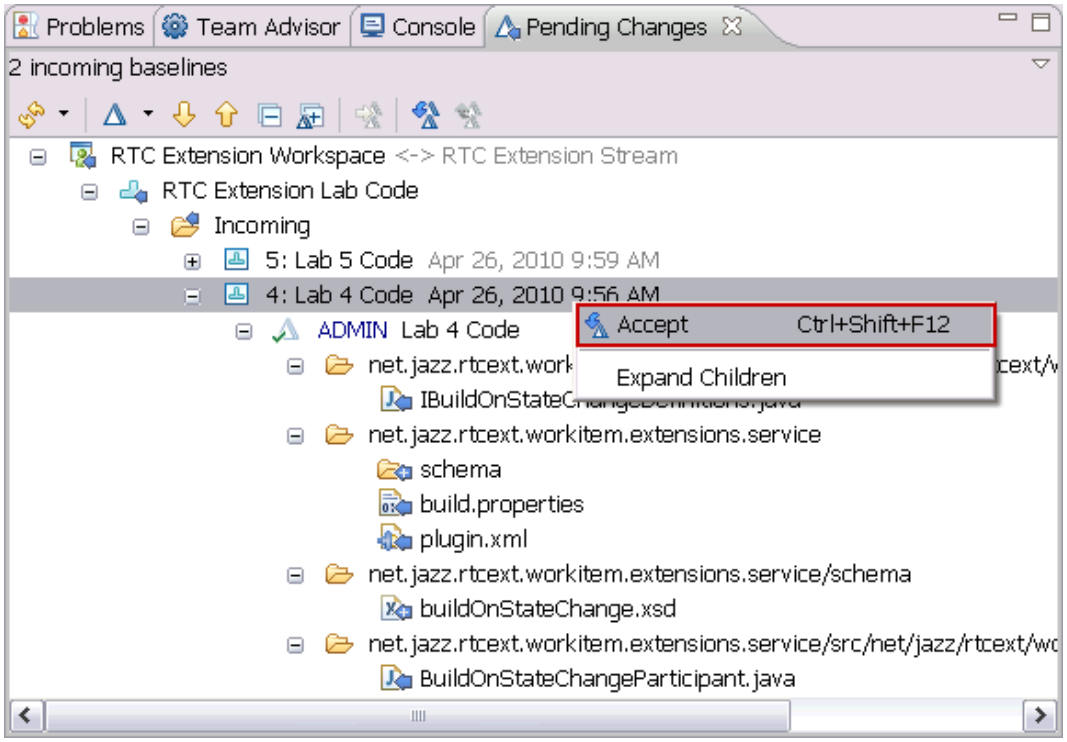
__f. Browse the changes and you will notice these key changes. The additional behavior will be discussed in detail after the code is loaded.

__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. These 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.rtcext.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";
```

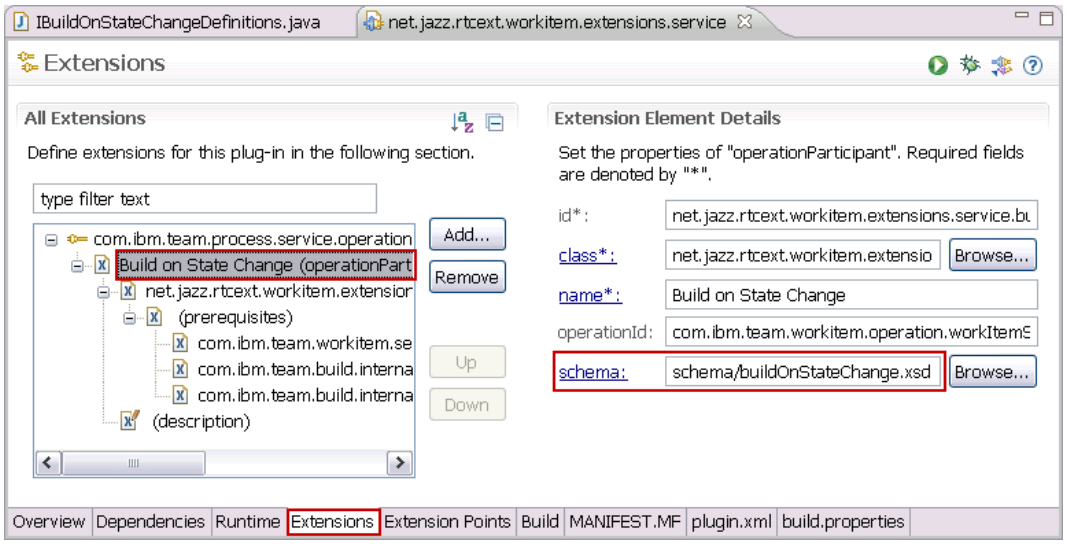
__iii. The second comment and set of constants defines what how the target build is configured.

```
// <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.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.

__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.fWorkItemTypeId)) {
```

__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 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);
                    }
            }
        }
}
```

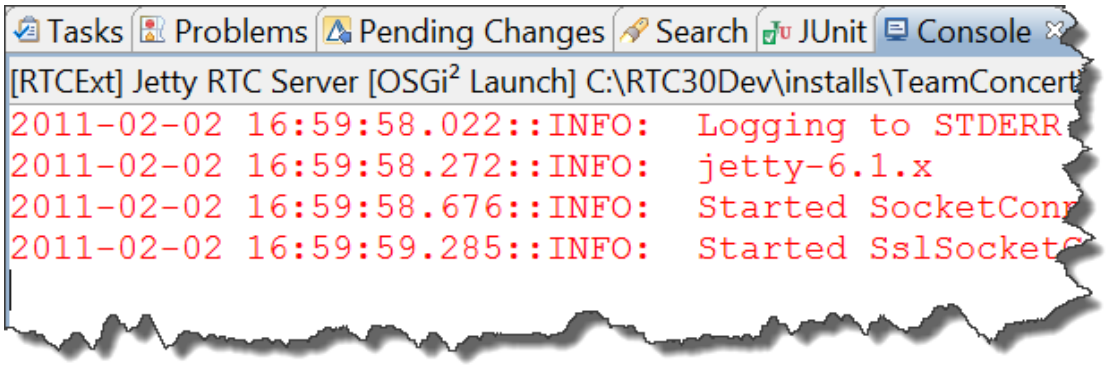
__k. You can now close all your open editors and proceed to the next section to configure and again step through the configured follow-up action.

4.2 Launch the Server for Debug Using Jetty

__5. Use the existing launch configuration from lab 2.

__a. From the **Debug** toolbar dropdown ( ) in the toolbar, select **[RTCExt] Build on State Change - Jetty RTC Server**.

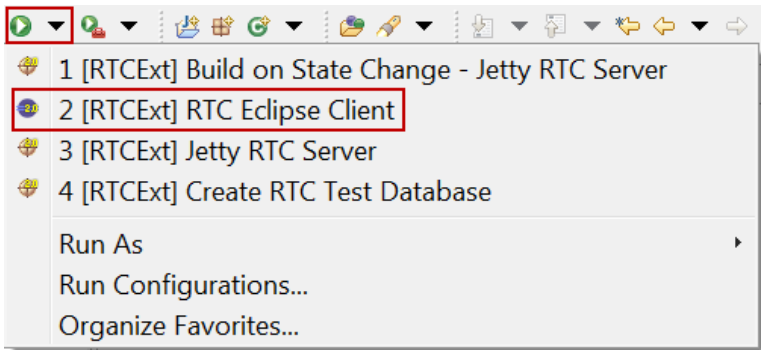
__b. As before, the **Console** view will show a few log messages indicating that the Jetty server is up and running.



4.3 Launch an RTC Client and Configure the Participant

__6. Launch the RTC Client.

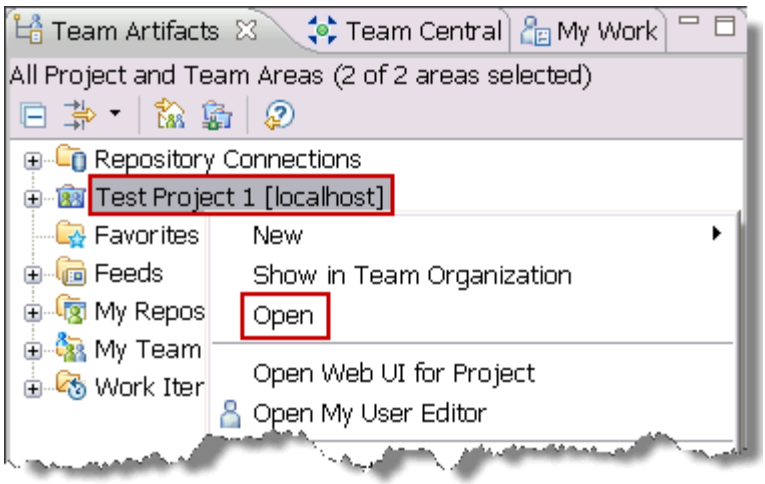
__a. From the dropdown menu of the **Run** toolbar icon, select **[RTCExt] RTC Eclipse Client**. Note that you are just running the client and not debugging. The same launch configuration can be used for both. You will debug a client in a future lab. If prompted do not clear the runtime workspace. You will probably answer no for this question for the rest of this workshop. You can turn off the prompt by editing the launch configuration.



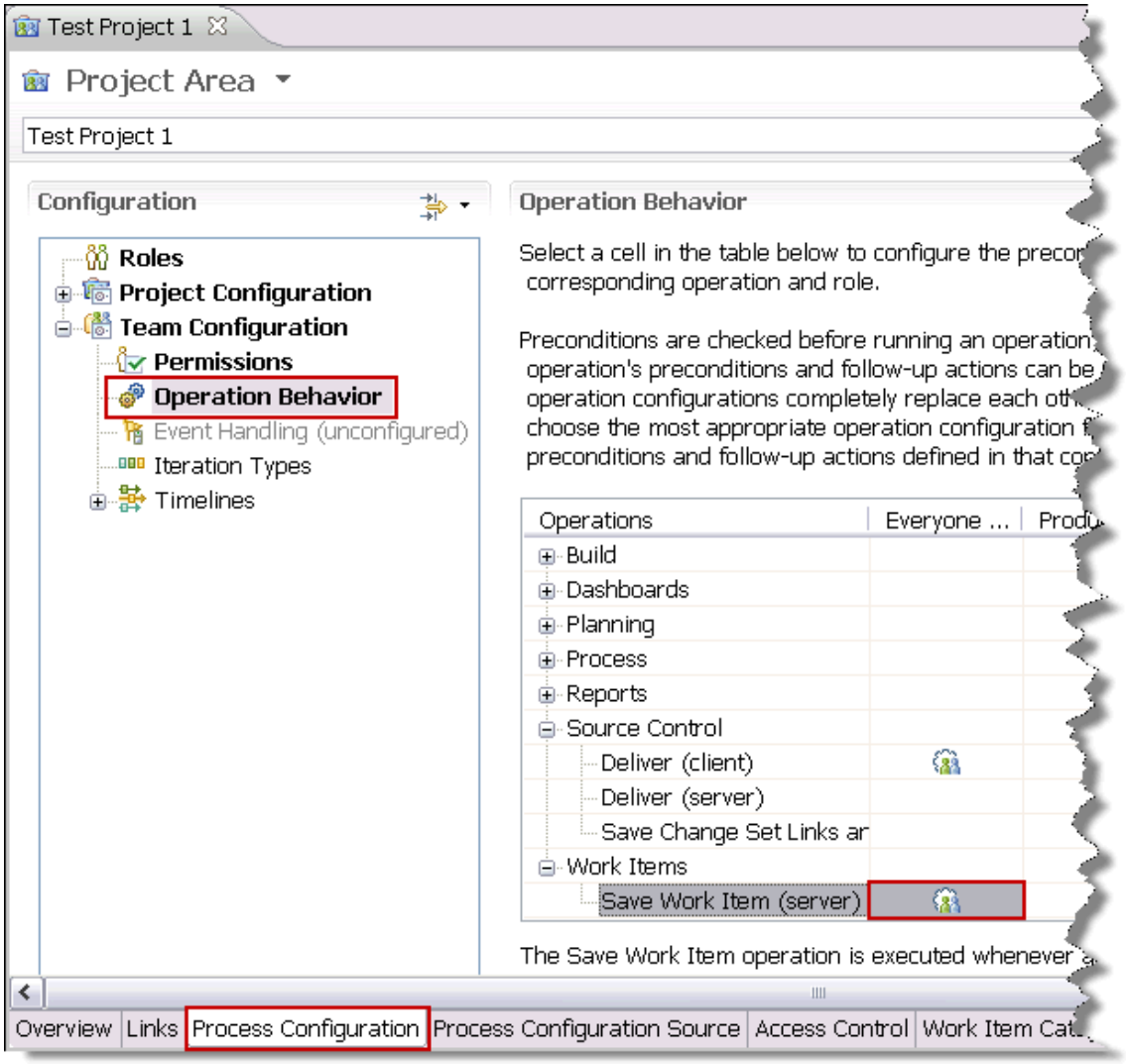
__b. The RTC Eclipse client will start up and will connect automatically to the Jetty server you just launched via the repository connection you created in lab 2. The project area will still be connected; however, you do have some more work to do this time. The participant is still added as a follow-up 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.

__7. 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.

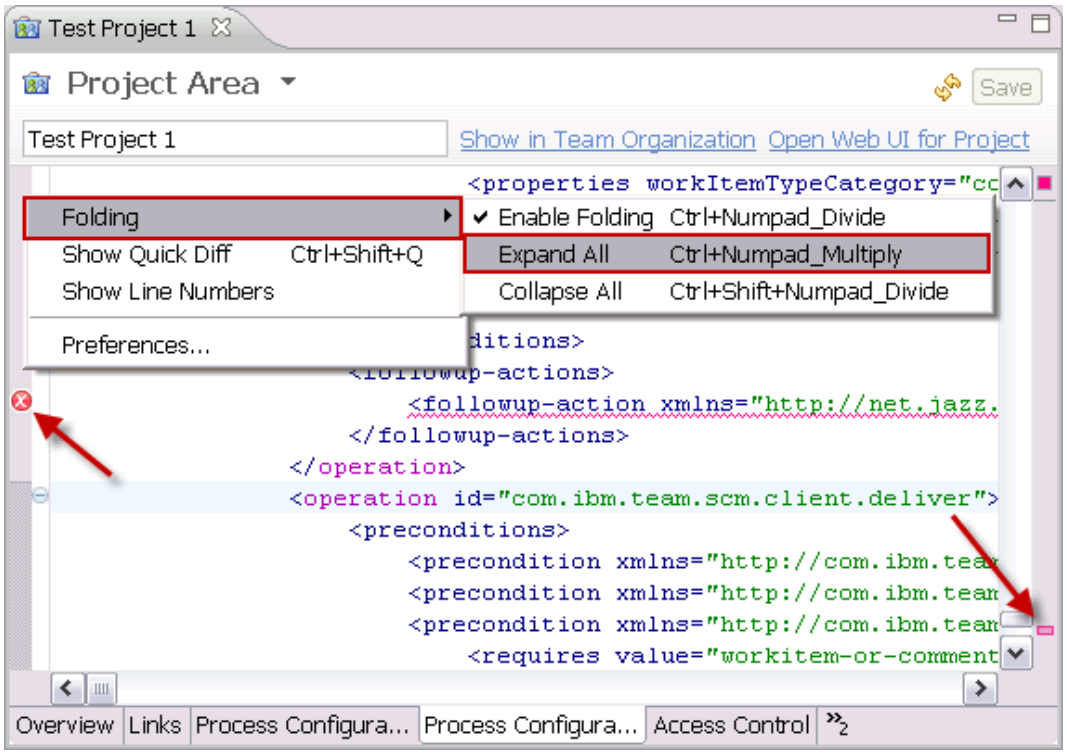


__c. Scroll down to find the **Follow-up actions** section on the right, remove the **Build on State Change** participant that is already in the list and then add it back in again. This may seem unusual, but there is a good reason for it. If you looked at the XML for the participant before and after 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.

__8. 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.rtcext.workitem.extensions.service/server/buildOnStateChange":
 trigger, "http://net.jazz.rtcext.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.apt.workItemType.story"/>
              <trigger-state id="com.ibm.team.apt.story.tested"/>
        </trigger>
        <build>
              <build-definition id="our.integration.build"/>
        </build>
</followup-action>
```

__d. 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.

The trigger-state id should be: `com.ibm.team.apt.story.tested`

The build-definition id should be: `our.integration.build`

__e. 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

__9. Find the Story work item used in lab 2 and 3 (it is probably number 7) and then move it out of the Implemented state (via the **Reopen** action) or create a new story.

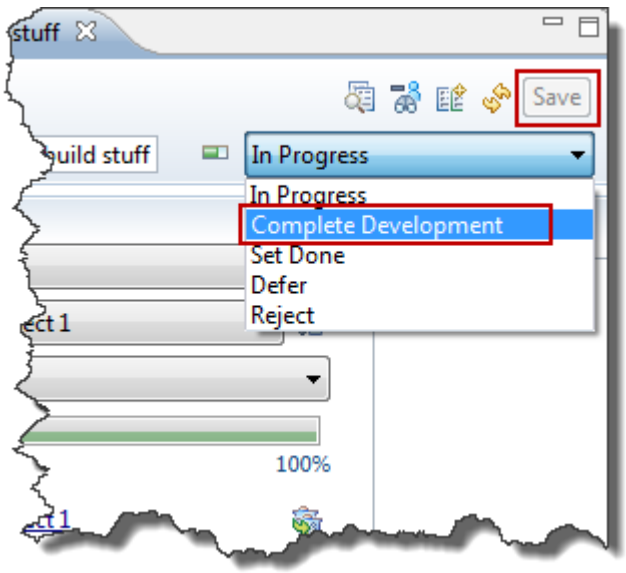
__a. Either of these will cause the breakpoint you set earlier to trigger. If not, re-check the breakpoint is at a valid source code line. The RTC Eclipse client in which you were studying the code will now surface (if asked about switching to the debug perspective, click **Yes**). If it does not surface, 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 ( ).

__10. Move the Story to the Implemented state.

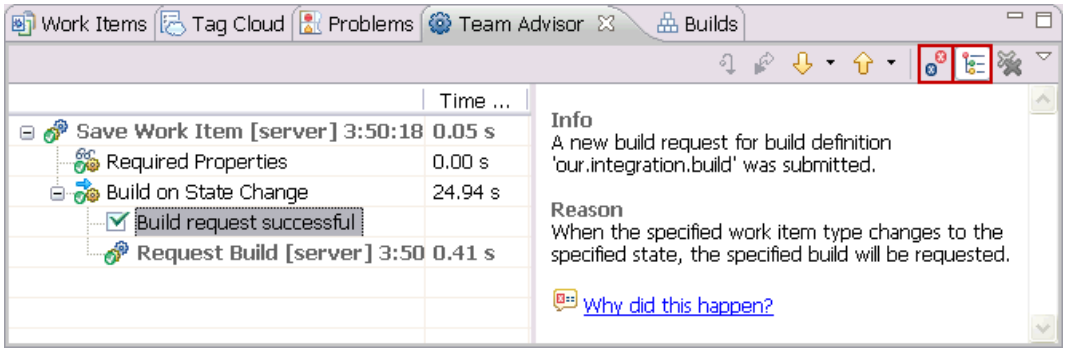
__a. At the upper right portion of the work item editor, select **Set Implemented** or **Complete Development** (depends on which workflow state the story is currently in) and then click **Save**.



__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 double click **our.integration.build** in the **Team Artifacts** view, the **Builds** view will show a new pending build request.



4.5 Change the Build Id in the Configuration and Try Again

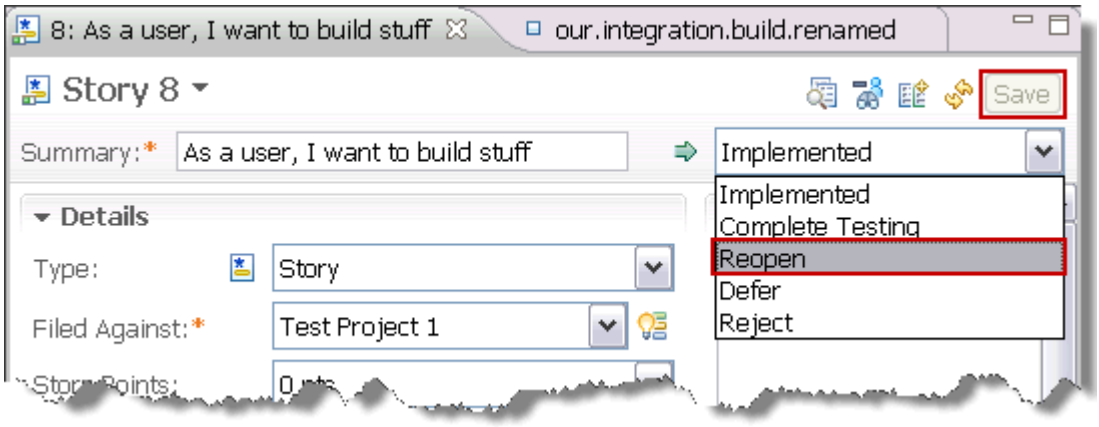
__11. Return to the Test Project 1 project area editor and change the build id.

__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.

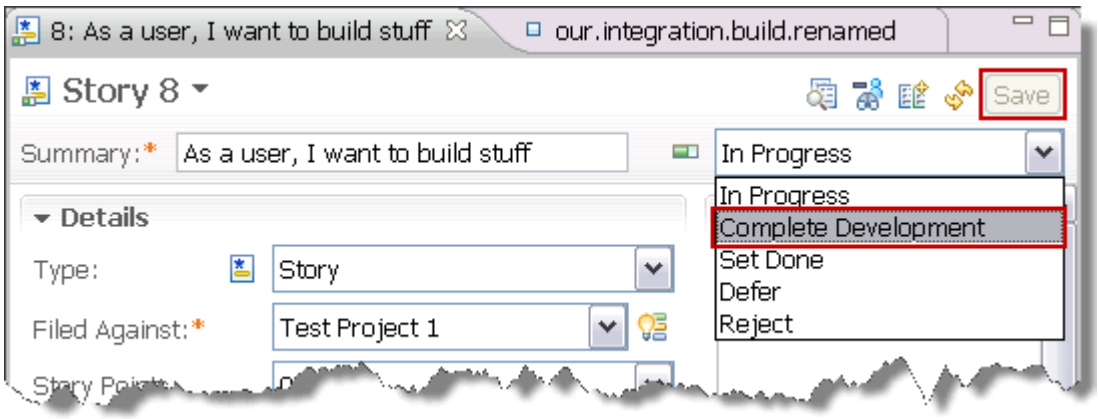
```
<followup-action
  xmlns="http://net.jazz.rtcext.workitem.extensions.service/server/buildOnStateChange"
  description="When the specified work item type changes to the specified state, the
   specified build will be requested."
  id="net.jazz.rtcext.workitem.extensions.service.buildOnStateChange"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

__12. Move the story to the Implemented state again.

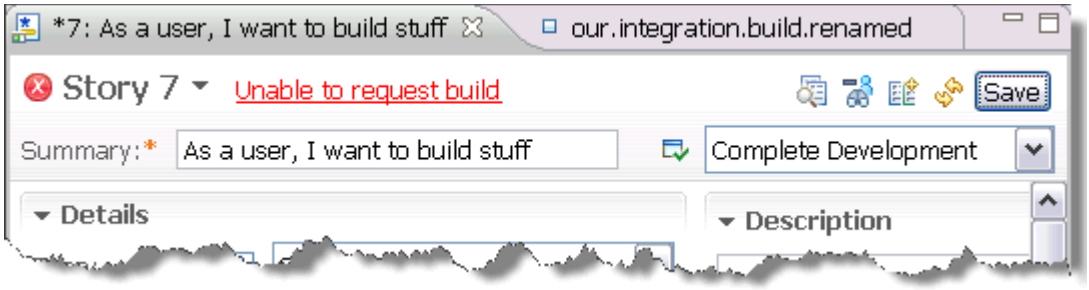
__a. Switch back to the work item editor and select **Reopen** from the state dropdown and then click **Save**. When the debugger surfaces, just click the resume button ( ). You are not to the interesting bit yet.



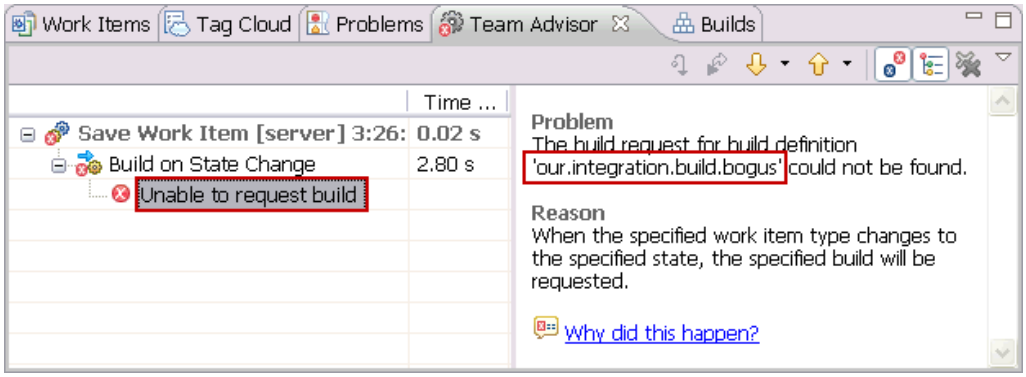
__b. Again in the work item editor, select **Complete Development** from the same dropdown and click **Save** again.



__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.



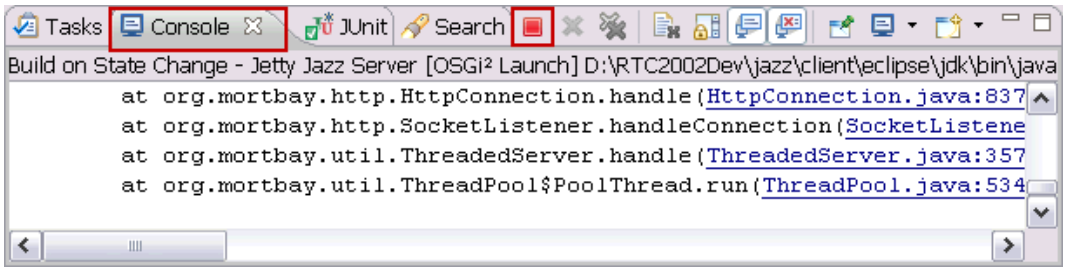
__e. Switch back to the project area editor and change the **ID** back to `our.integration.build` and click **Save**.

__f. Switch back to the work item editor and click **Save**. When the debugger surfaces, just click resume and 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.

__13. 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's **Console** view, click the **Terminate** icon, remove all terminated launches and check no launch is still running.



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).

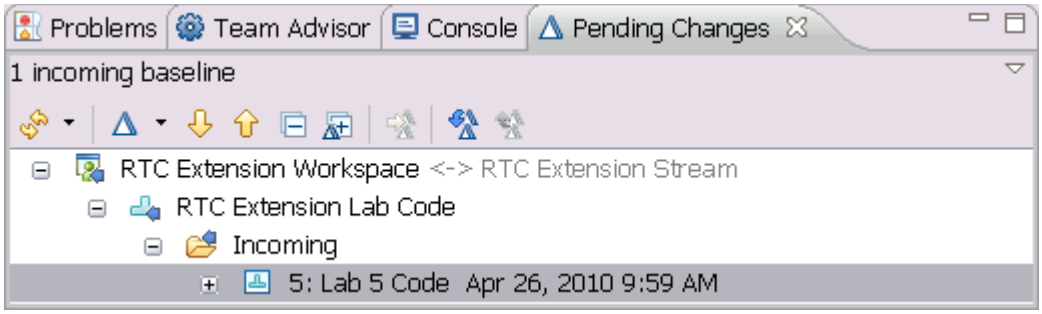
If your RTC server is not running, start it now (`C:\RTC40Dev\installs\JazzTeamServer\server\server.startup.bat`).

5.1 Understanding the Aspect Editor

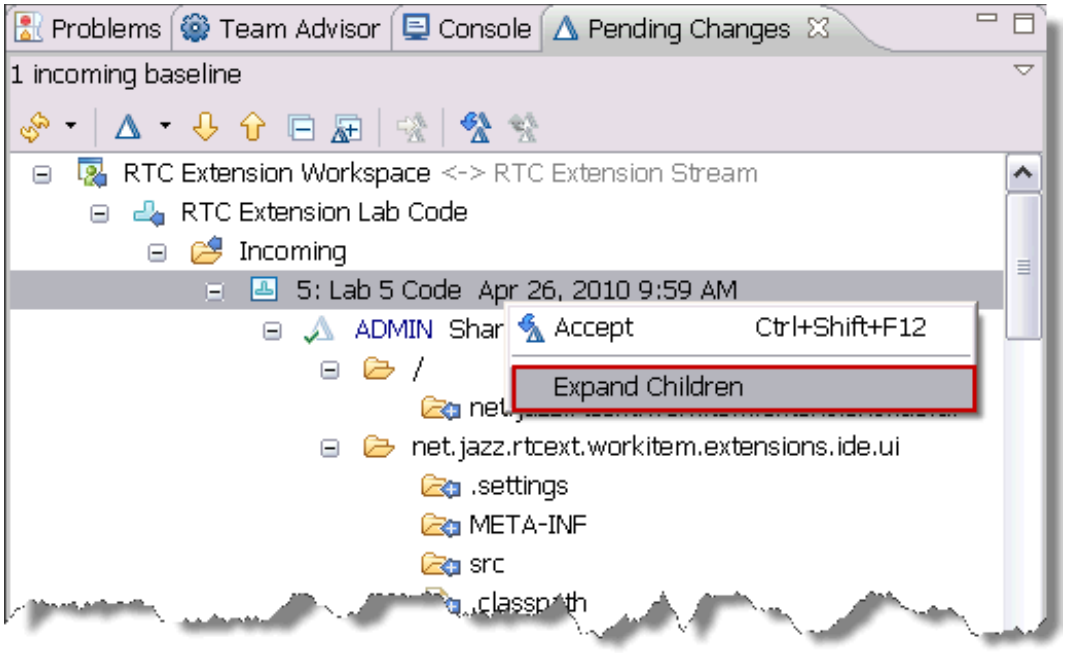
__1. If your RTC development environment is not open, navigate to `C:\RTC40Dev\installs\TeamConcert\jazz\client\eclipse` in the Windows explorer and double click **eclipse.exe**. If prompted to select an Eclipse workspace, select the same one you used in lab two. If the **Plug-in Development** perspective 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.

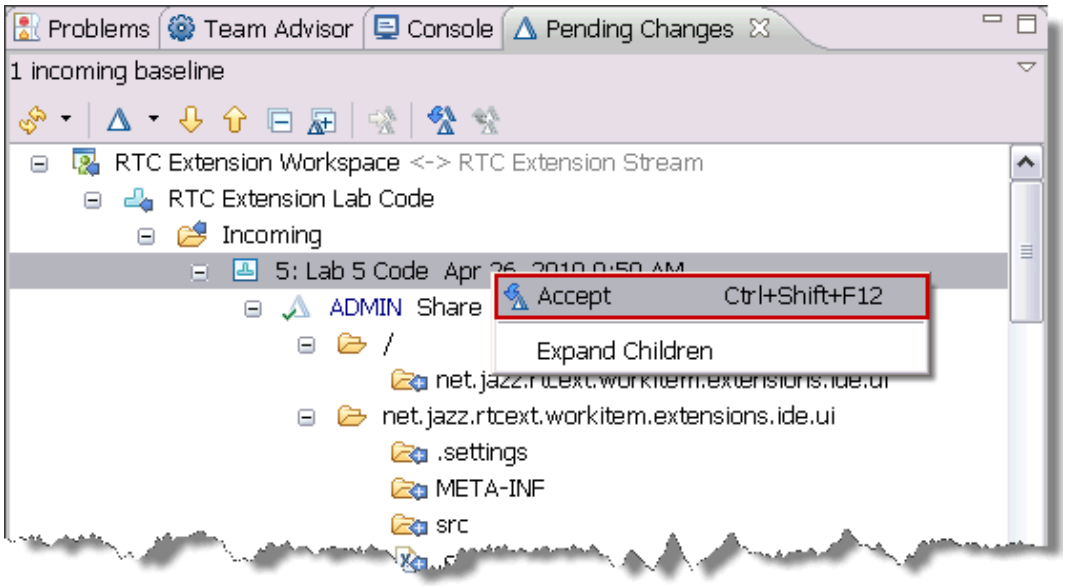
__a. In the **Pending Changes** view, click the **Expand to Change sets** icon. This will show 1 incoming baseline as shown here.



__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. You will next load the code and then go through it in detail.



__c. 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.

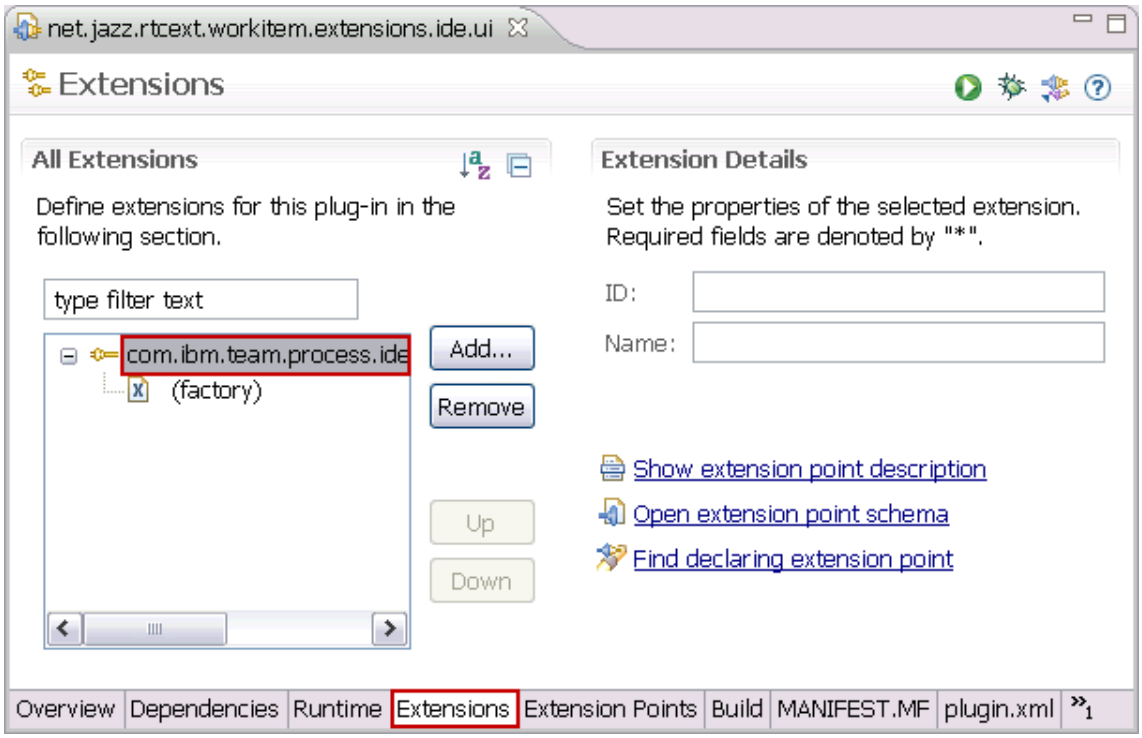


__d. If you see the following message select **Yes**

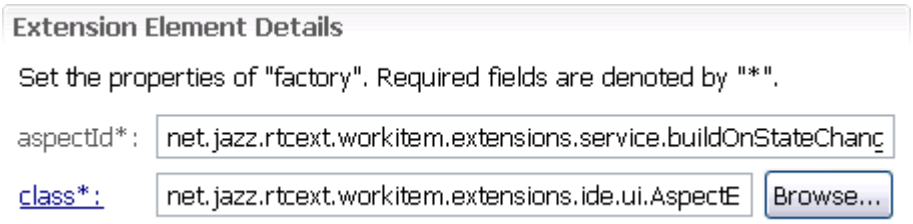
__3. Understanding the aspect editor plug-in.

__a. In the **Package Explorer** view, expand the tree for the new user interface project (net.jazz.rtcext.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 text in parenthesis on each line is the name of the xml element for that line. The raw xml can be seen on the **plugin.xml** tab of the editor.



__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(IBuildOnStateChangeDefinitions.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 workItemTypeId
 *            the work item type id to set
 * @return true if the value changed, false if it did not
 */
public boolean setWorkItemTypeId(String workItemTypeId) {
        boolean changed = false;
        String normalizedId = normalize(workItemTypeId);
        if (!fWorkItemTypeId.equals(normalizedId)) {
             fWorkItemTypeId = 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 descendants 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.

- `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.
- `saveState(IMemento memento)` which passes through to the `saveTo(IMemento memento)` method on the model class you just studied.
- `dispose()` which does nothing.
- `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) {
```

__e. 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.

__i. 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.

__ii. The second group is for initialization of the user interface. They include initUI and initStates.

- 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 combox and then uses the model to select the proper element of each combobox.
- 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.

__iii. The third group is used from the combobox selection changed listeners (and a couple other locations) to validate user selections.

- 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.

- The setValidationMessage method is use by validateSelections to actually manipulate the UI elements that are used to inform the user of validation issues.

__iv. 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.

__v. 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.

__vi. The sixth group includes getWorkItemTypes, getStatesForTypeCategory and their related nested types: WorkItemType and WorkItemState.

- 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).

- The getStatesForTypeCategory method returns an array of WorkItemState instances that are valid for the passed workflow id.

- 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.

__vii. The seventh and final group includes the getBuildDefinitions method and the BuildDef nested type.

- 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.

- 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.

__f. Next there is the issue of setting breakpoints for your upcoming debug session(s). Recommended locations include the beginning of the createContol 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.

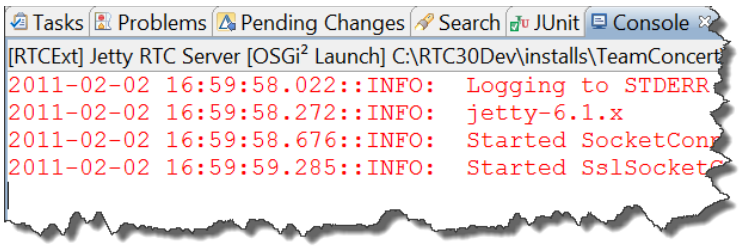
__g. You can now close all your open editors and proceed to the next section to try out your new aspect editor.

5.2 Launch the Server for Debug Using Jetty

__5. Use the existing launch configuration from the prior labs.

__a. From the **Debug** toolbar dropdown ( ) in the toolbar, select **[RTCExt] Build on State Change - Jetty RTC Server**.

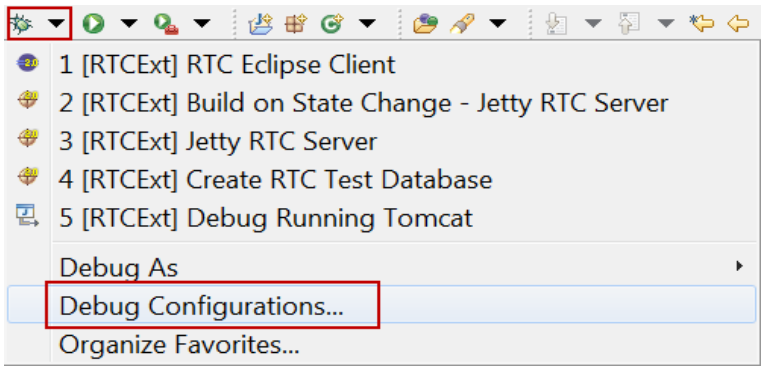
__b. As before, the **Console** view will show a few log messages indicating that the Jetty server is up and running.



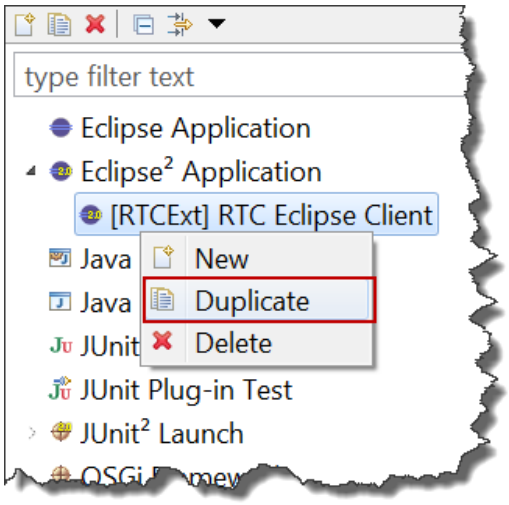
5.3 Launch an RTC Client and Configure the Participant

__6. Create a new launch configuration for the RTC Client plus your aspect editor.

__a. From the **Debug** toolbar dropdown, select **Debug Configurations…**

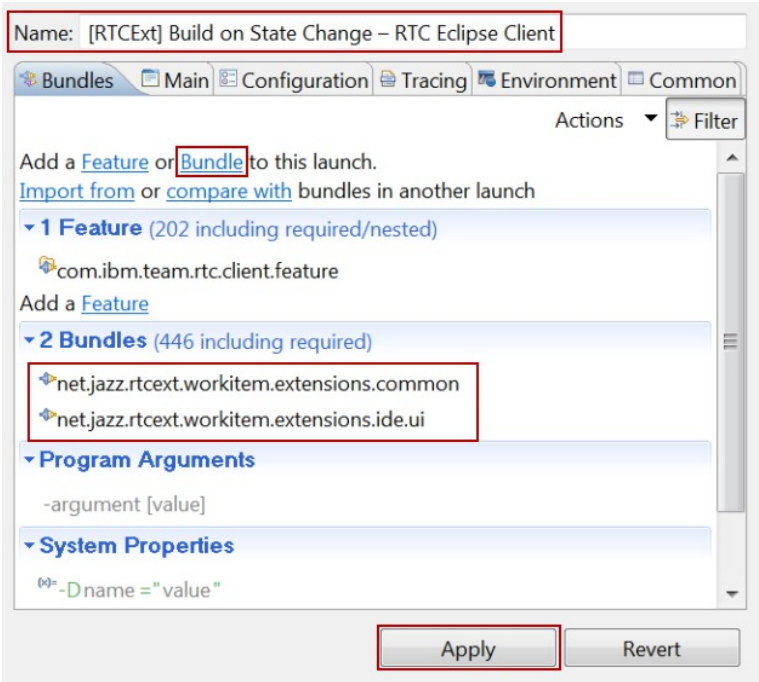


__b. In the **Debug Configurations** dialog, expand the **Eclipse² Application** tree and right click the **[RTCExt] RTC Eclipse Client** configuration and then from the popup menu, select **Duplicate**. Note that you are not changing the existing launch but creating a copy of it. You should keep the 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 `[RTCExt] Build on State Change - RTC Eclipse Client`.

__d. Add the common and ui bundles to the configuration. Click on the **Bundle** link and in the **Add Bundle** dialog, type `rtcext` 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.

__7. Launch the RTC client.

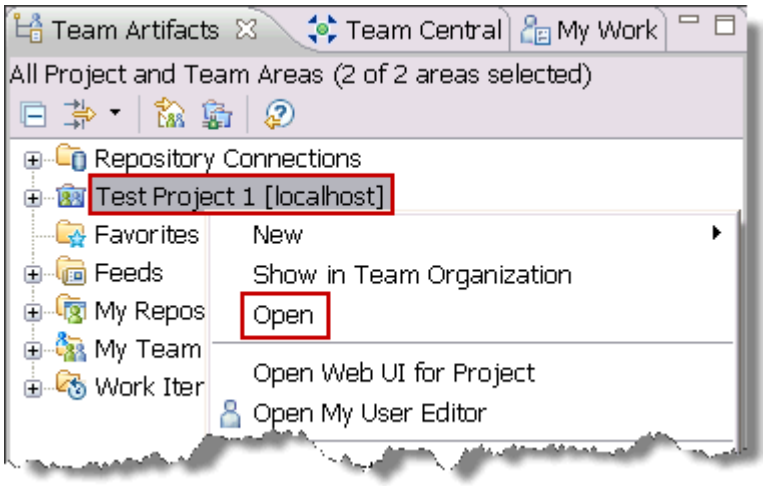
__a. Click **Debug** at the bottom of the **Debug Configurations** dialog. If prompted do not clear the runtime workspace. You will probably answer no for this question for the rest of this workshop. You can turn off the prompt by editing the launch configuration.

__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. The project area will still be connected and the participant is fully configured from lab 4.

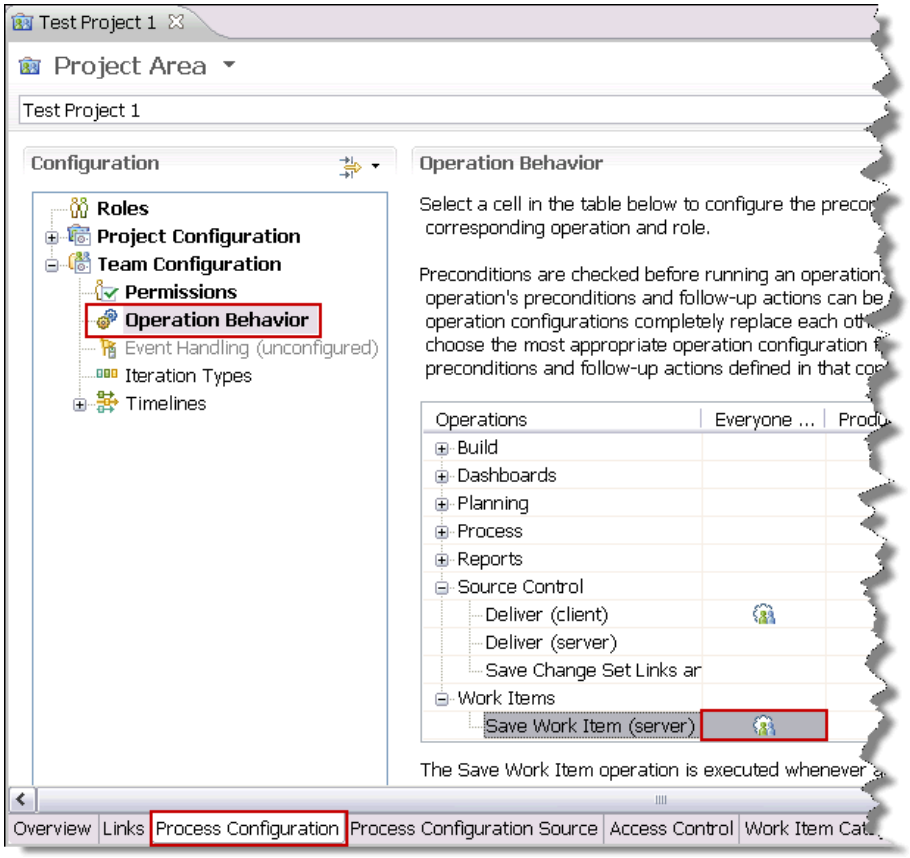
__c. The next time you want to debug this server configuration, you will be able to click a shortcut to it on the dropdown of the **Debug** toolbar icon. You will not need to open the **Debug Configurations** dialog.

__8. 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.

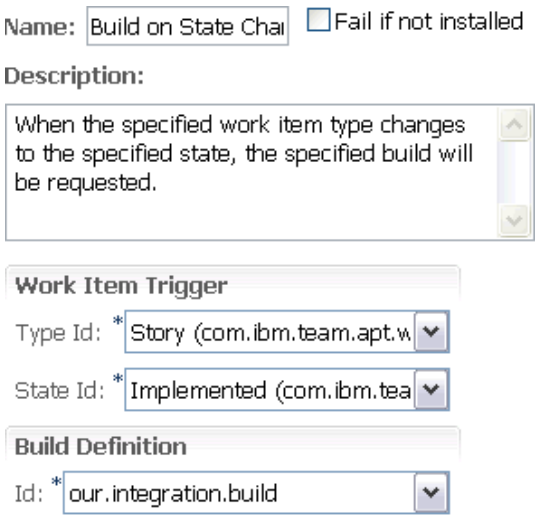
__i. If you set it, your breakpoint in the restoreState method will trigger. Step into and through the two methods called from here.

__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 comboxes 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.



__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

__9. 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.

__10. Find the Story work item used in labs 2 through 4 and then move it out of the Implemented state (via the **Reopen** action) or create a new story.

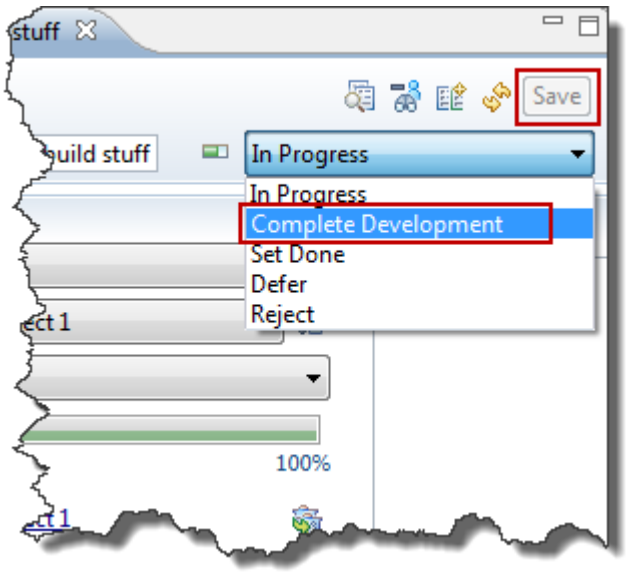
__a. Either of these will cause the breakpoint you set earlier to trigger (unless you cleared it). The RTC Eclipse client in which you were studying the code will now surface (if asked about switching to the debug perspective, click **Yes**). If it does not surface, you probably minimized it earlier. 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 ( ).

__11. Move the Story to the Implemented state (or your different type to the trigger state).

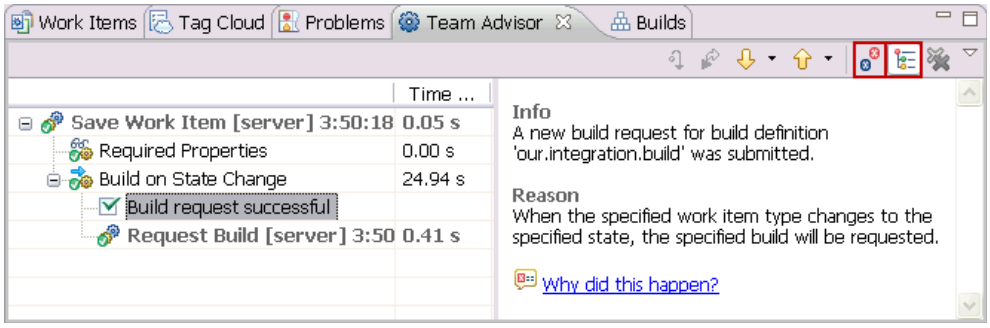
__a. At the upper right portion of the work item editor, select **Set Implemented** or **Complete Development** (depends on which workflow state the story is currently in) and then click **Save**.



__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 ( ).

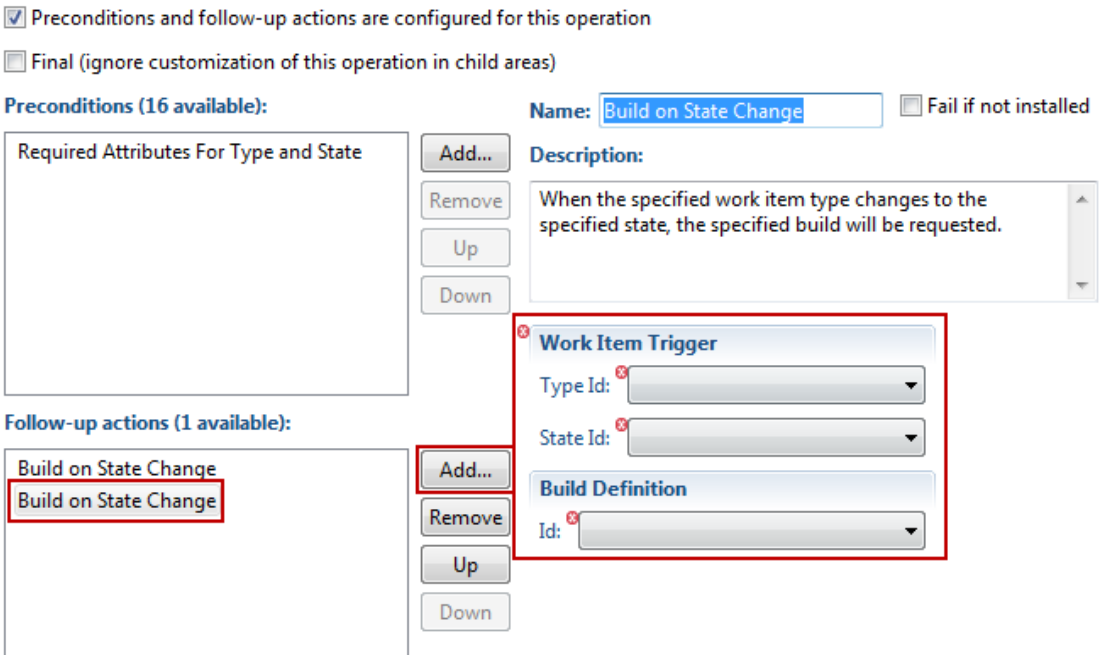
__d. If you go to the **Team Advisor** view and check to make sure the **Show Failures Only** filter is off and **Show Detail Tree** is on (see highlight below), you can browse the results of this successful operation. Also, if you double click **our.integration.build** in the **Team Artifacts** view, the **Builds** view will show a new pending build request.



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

__12. 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 actions** 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.



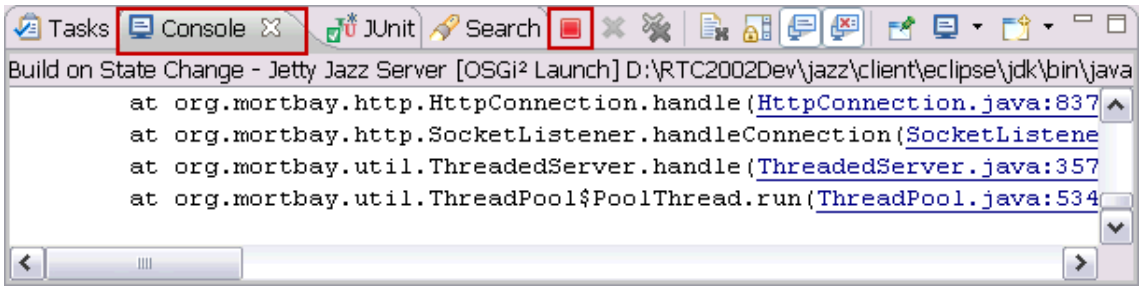
__b. Select a work item and state that are different from the ones configured for the first instance. Select the one and only build definition. If you wish, you can create a new build definition. If you do create a new build definition, you will not see it until a new instance of the aspect editor is created. A new instance is created each time you select a participant in the **Follow-up actions** list.

__13. 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.

__14. 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:\RTC40Dev\installs\TeamConcert\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

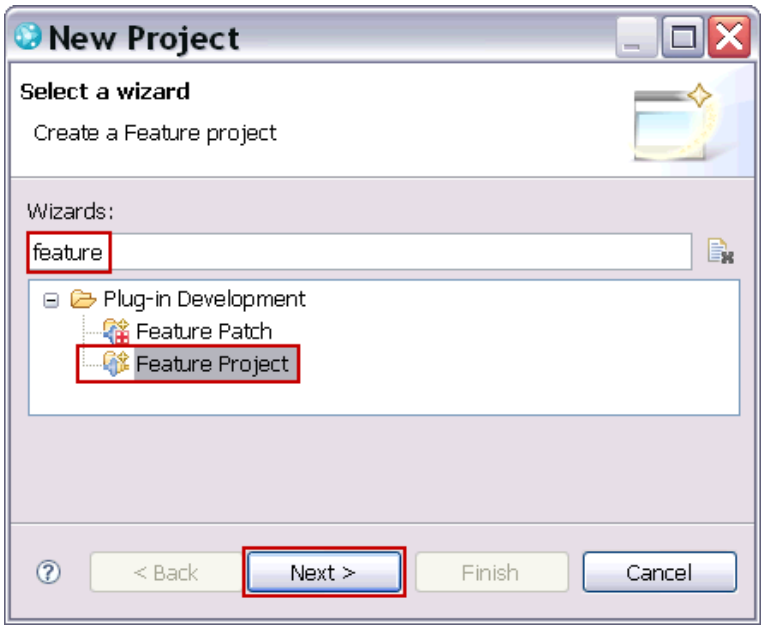
If your RTC server is not running, start it now (`C:\RTC40Dev\installs\JazzTeamServer\server\server.startup.bat`).

6.1 Creating a Server Side Feature

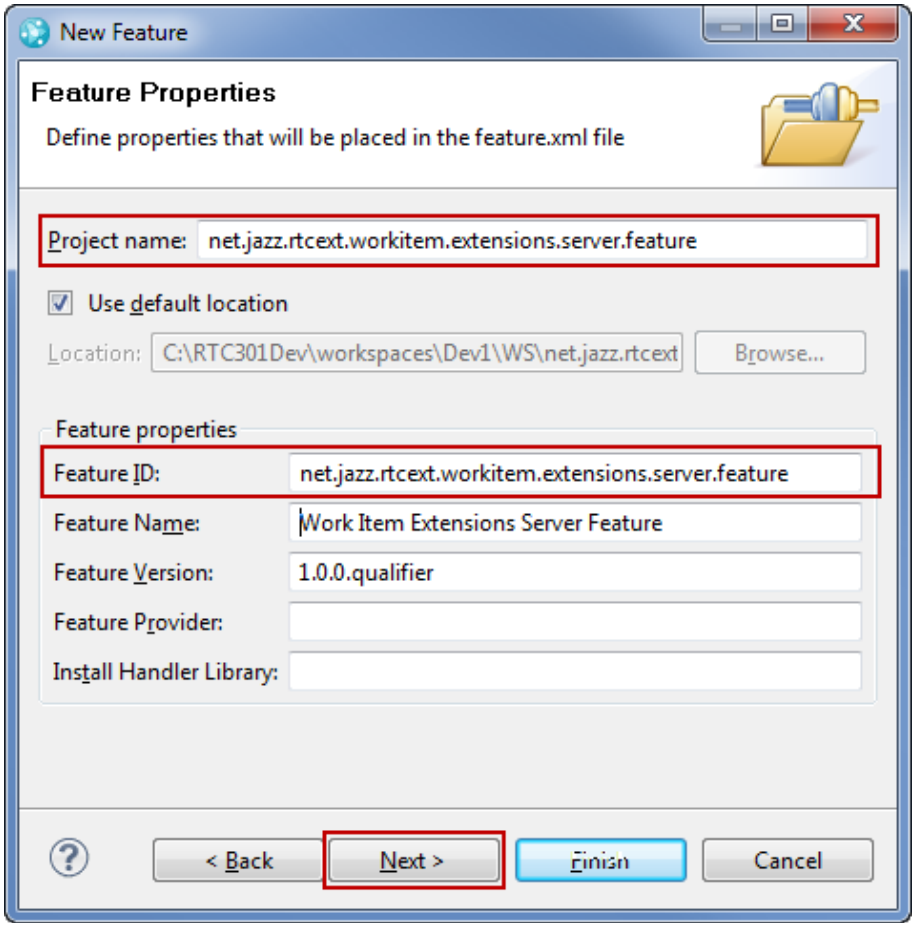
__15. If your RTC development environment is not open, navigate to `C:\RTC40Dev\installs\TeamConcert\jazz\client\eclipse` in the Windows explorer and double click **eclipse.exe**. If prompted to select an Eclipse workspace, select the same one you used in lab two. If the **Plug-in Development** perspective is not open, open it now by selecting **Window > Open Perspective > Other… > Plug-in Development** from the menu bar.

__16. Create the server side feature.

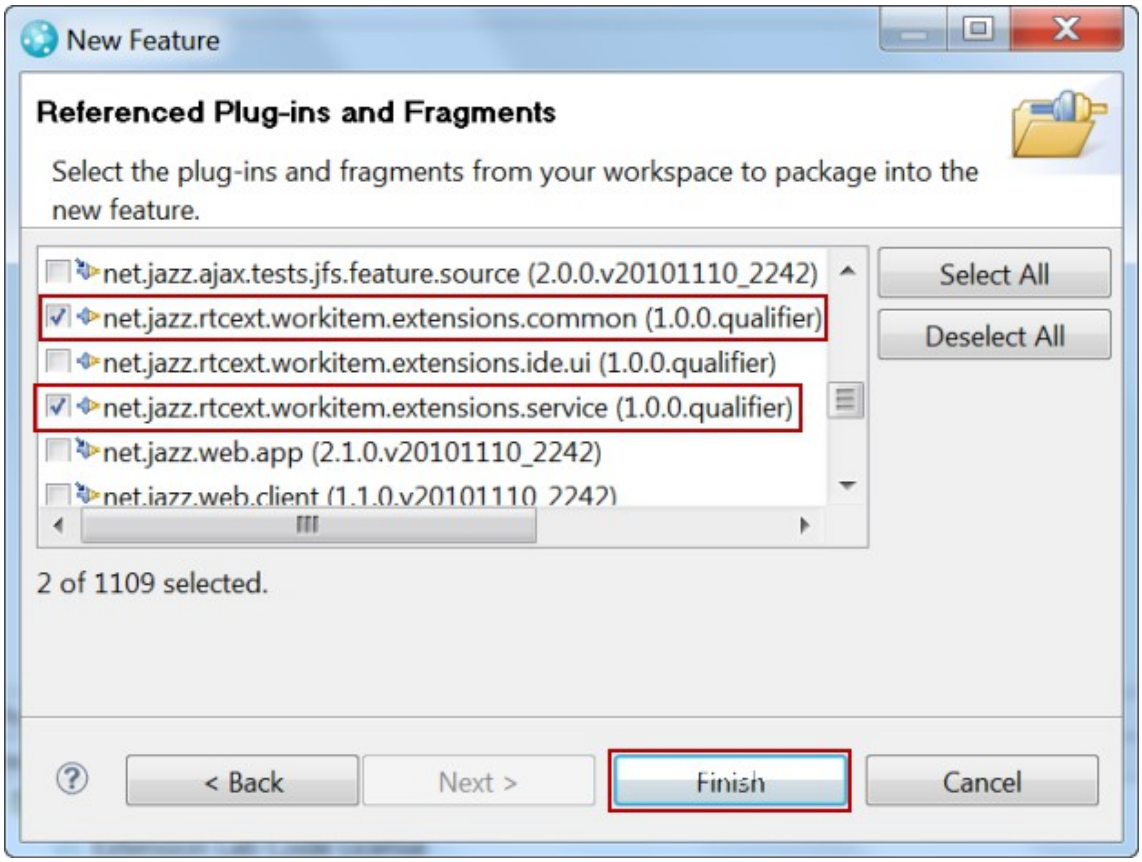
__a. From the menu bar, select **File > New > Project…** then in the **New Project** wizard, type `feature` in the filter field, select **Feature Project** from the list and then click **Next**.



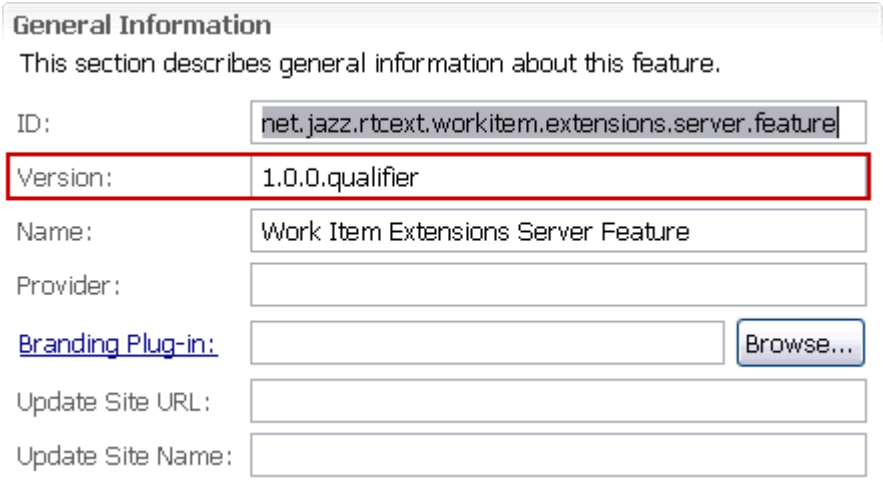
__b. On the second page of the wizard type `net.jazz.rtcext.workitem.extensions.server.feature` into the **Project name** field. As you type, the **Feature ID** is set to a reasonable value but the **Feature Name** should be reset to: `Work Item Extensions Server Feature`. You can set the **Feature Provider**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



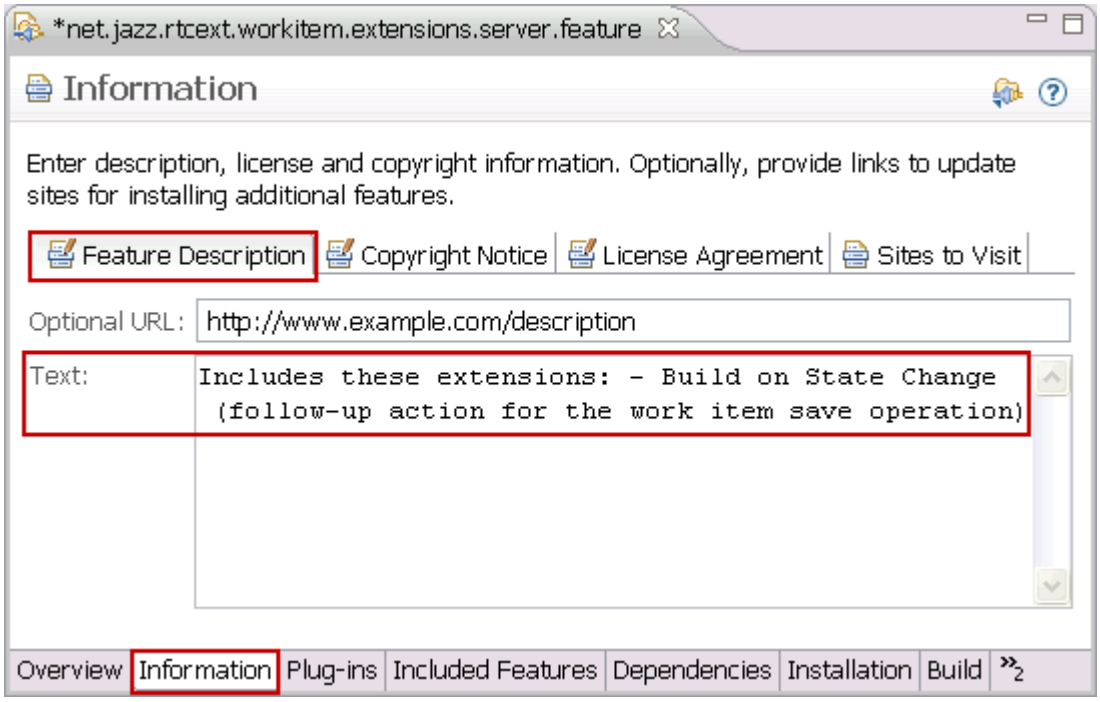
__c. On the final page of the wizard select the common and service plug-ins and then click **Finish**.



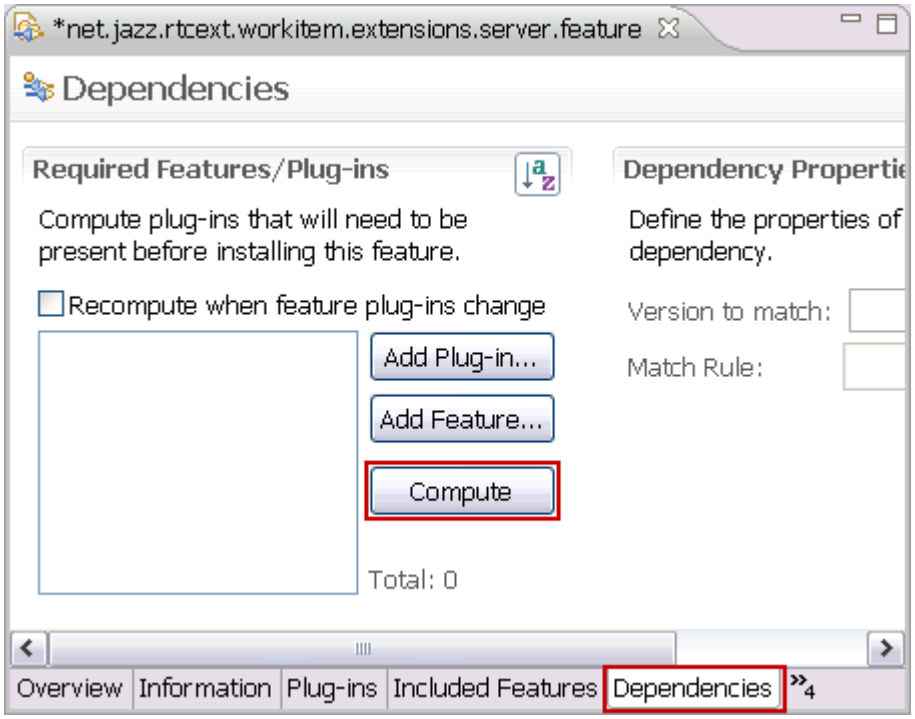
__d. Your new feature project appears in the **Package Explorer** view and an editor opens on the feature.xml file. On the **Overview** tab, make sure the **Version** is set to `1.0.0.qualifier`. This is the same Eclipse best practice you used for the plug-ins.



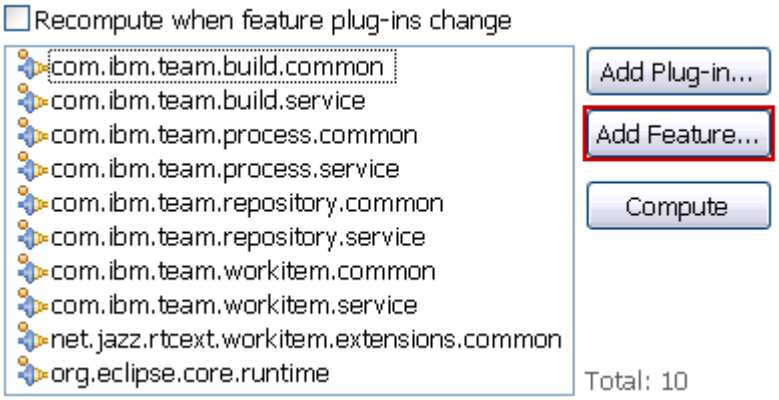
__e. Still in the editor, switch to the **Information** tab, select the **Feature Description** sub-tab and enter a **Text** description as shown here. If you wish you can look at other information that can be added, such as a copyright and license information.



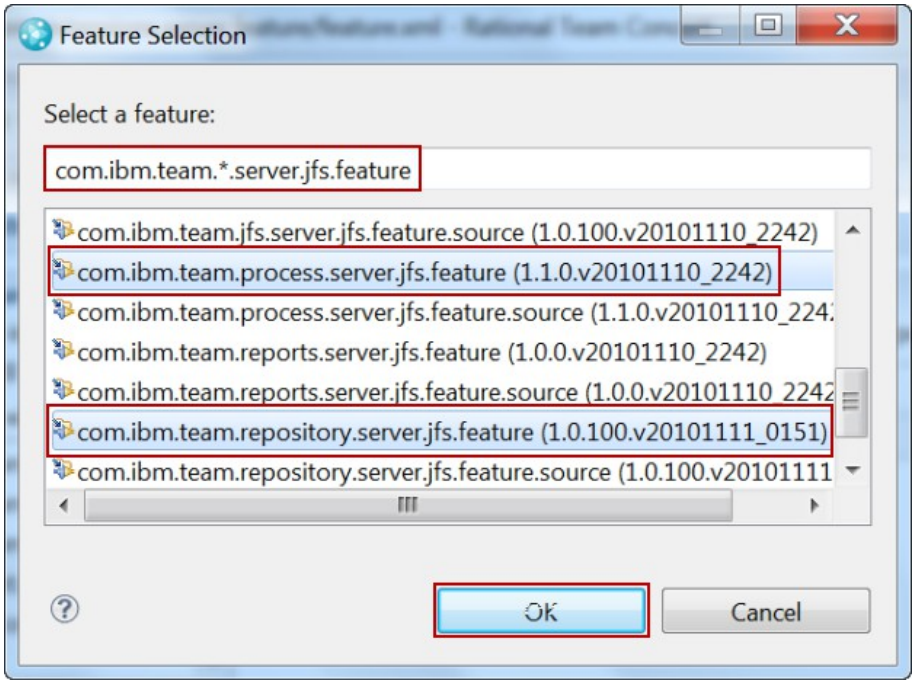
__f. Switch to the **Dependencies** tab and click **Compute**.



__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 two consistent naming patterns: **com.ibm.team.*component*.server.jfs.feature** and **com.ibm.team.*component*.server.rtc.feature** Click **Add Feature…**

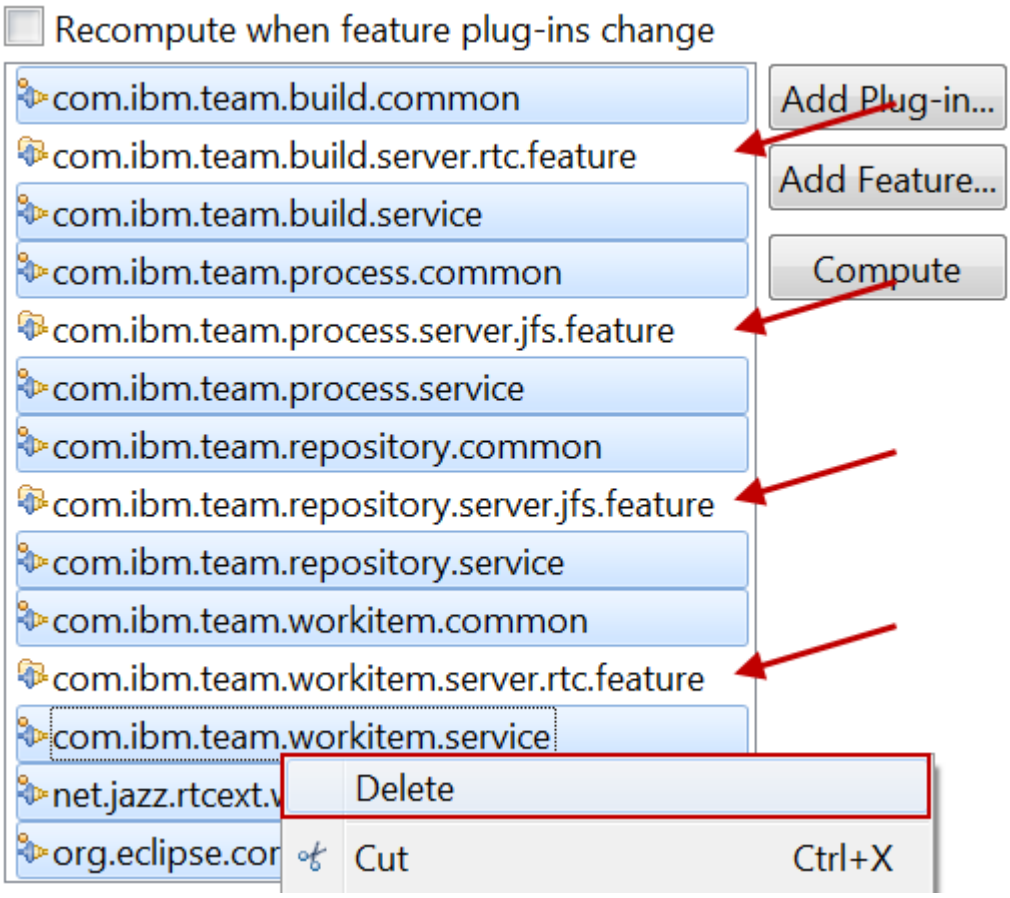


__h. In the **Feature Selection** dialog type `com.ibm.team.*.server.jfs.feature` into the filter field, select the two features shown here (**com.ibm.team.process.server.jfs.feature** and **com.ibm.team.repository.server.jfs.feature**) and then click **OK**.

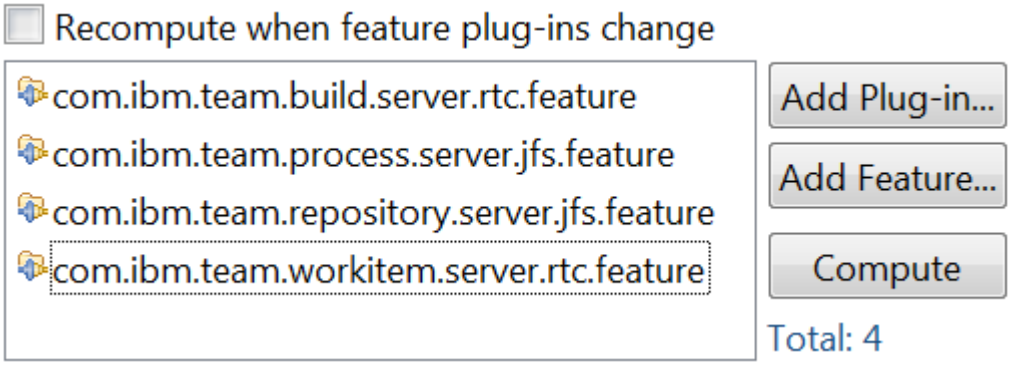


__i. Click **Add Feature...** again, but this time in the **Feature Selection** dialog type `com.ibm.team.*.server.rtc.feature` into the filter field, select these two features (**com.ibm.team.build.server.rtc.feature** and **com.ibm.team.workitem.server.rtc.feature**) and then click **OK**.

__j. 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.



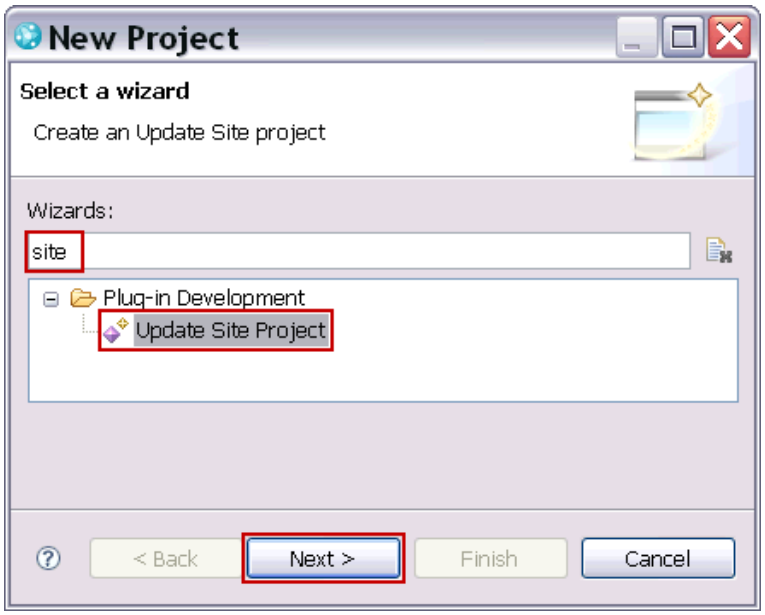
__k. The list will now look like this. Type **Ctrl+S** to save the feature.xml file. You can now close the editor.



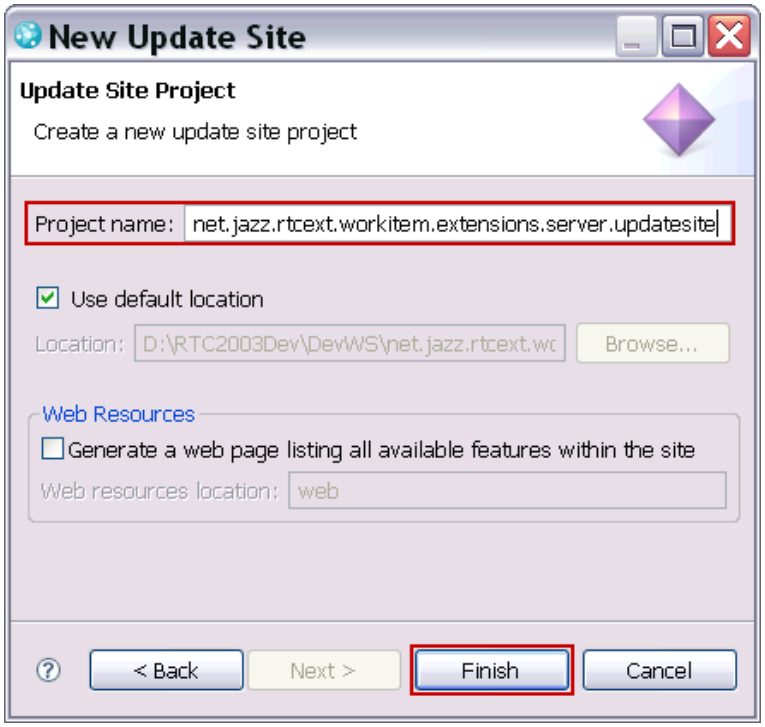
6.2 Create the Server Update Site

__17. 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**.

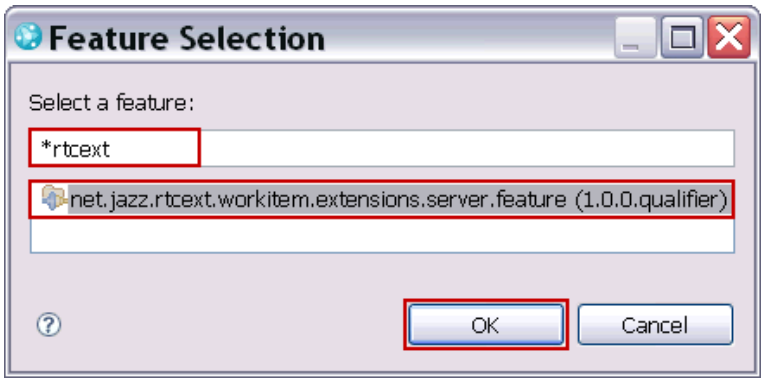


__b. On the second page of the wizard type `net.jazz.rtcext.workitem.extensions.server.updatesite` into the **Project name** field. Click **Finish**.



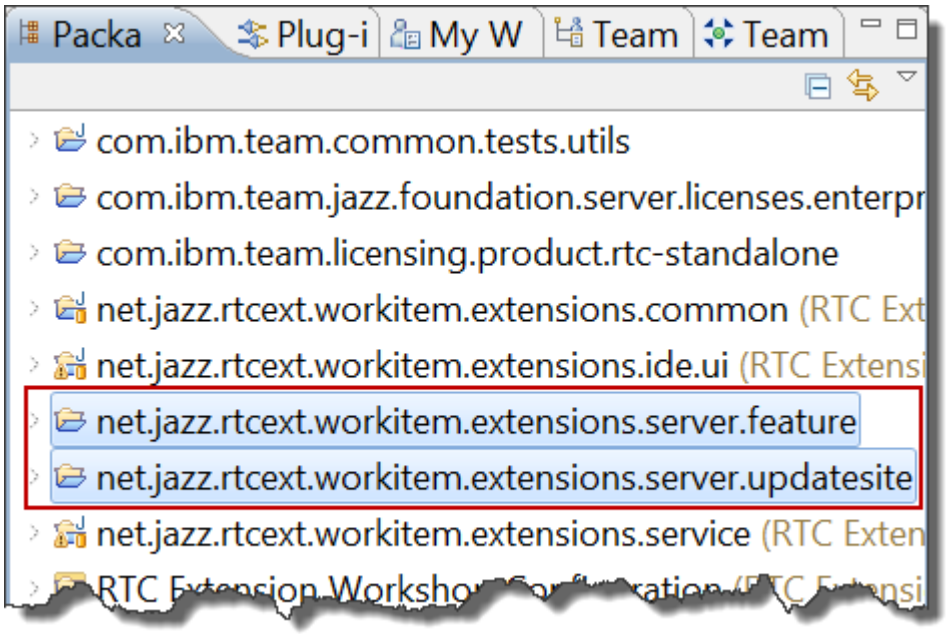
__c. Your new update site project appears in the **Package Explorer** view and an editor opens on the site.xml file. In the editor, remain on **Site Map** tab and click **Add Feature**.

__d. In the **Feature Selection** dialog, type `*rtcext` 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.

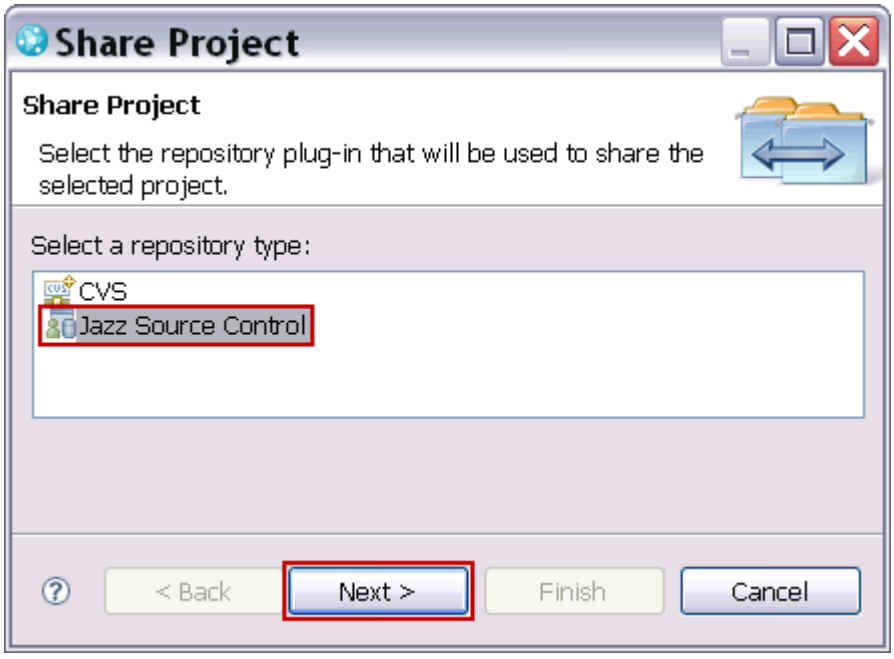


__18. Share the new projects to your repository workspace.

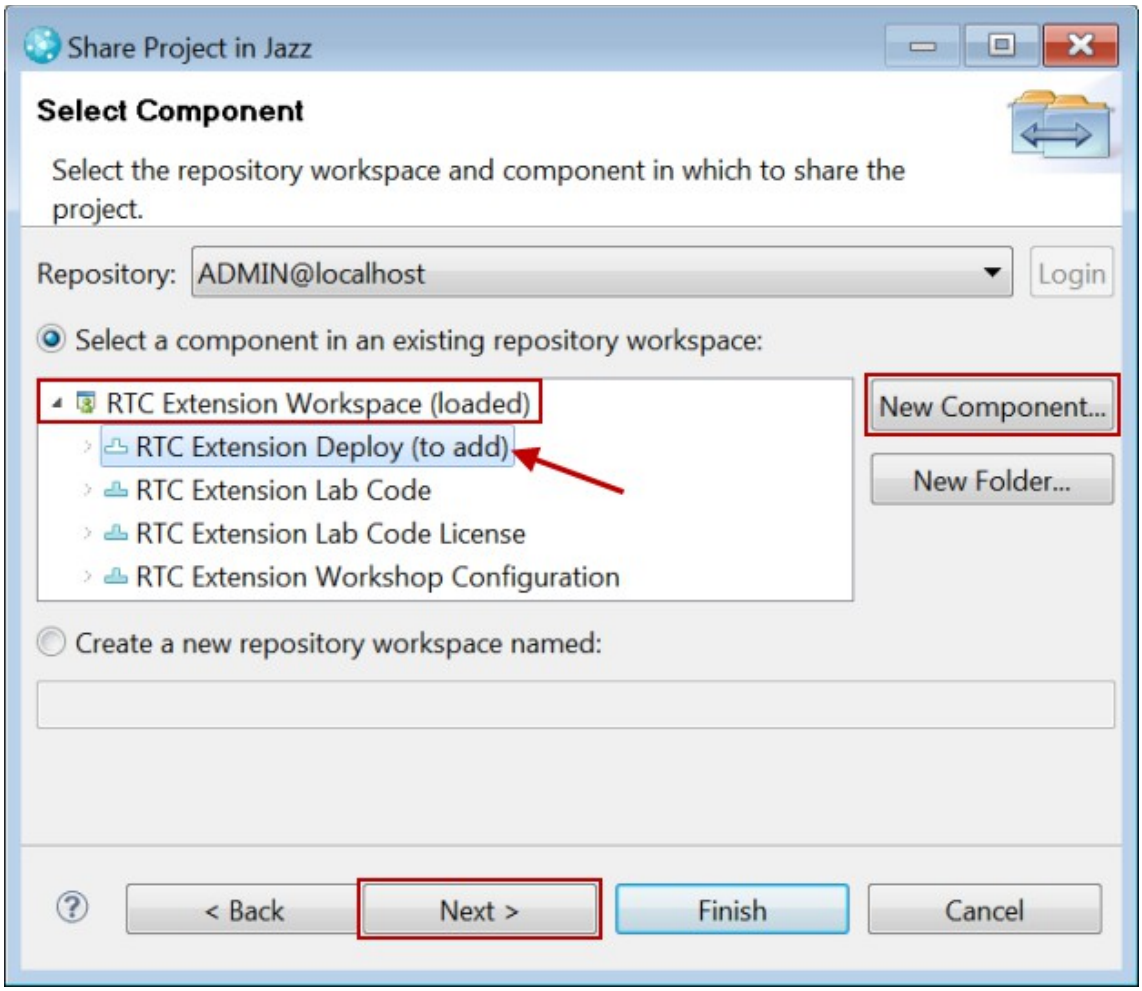
__a. In the **Package Explorer** view, select the feature and update site projects as shown here. Then, right click one of them and from the menu, select **Team > Share Project…**



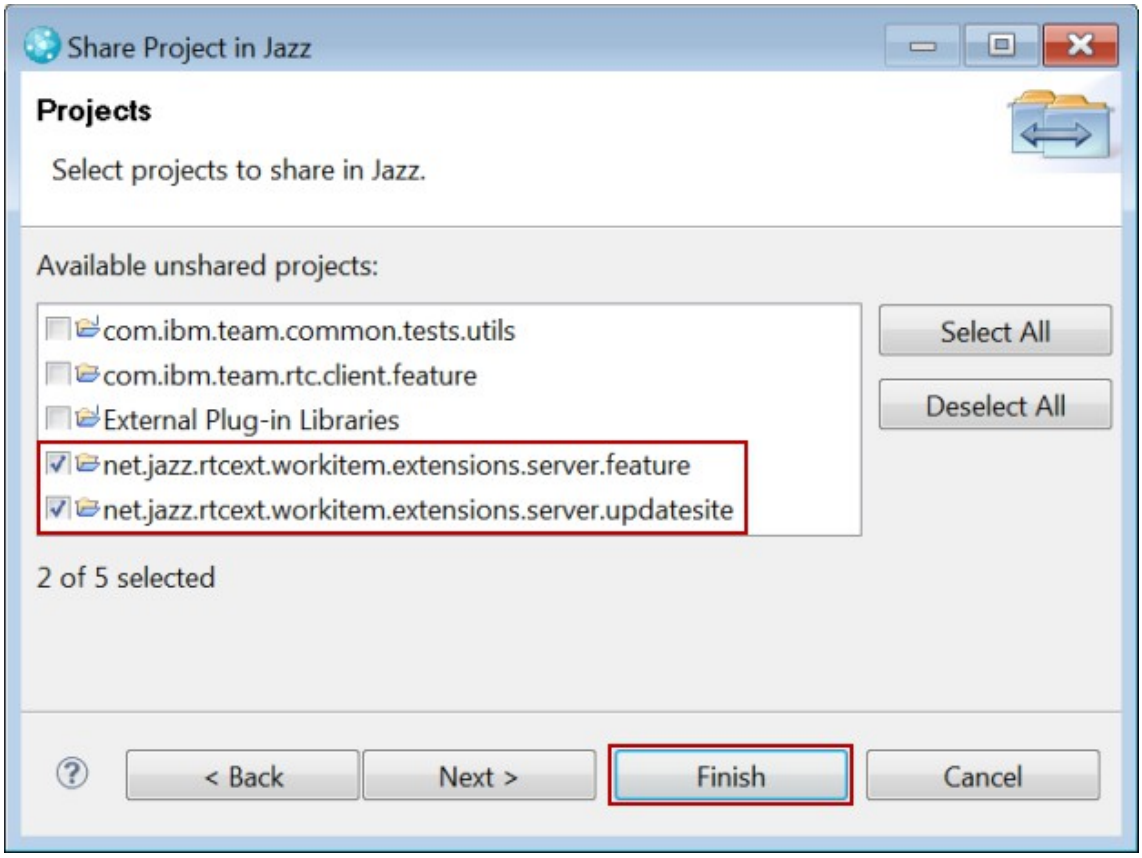
__b. In the **Share Project** wizard, select **Jazz Source Control** then click **Next**.



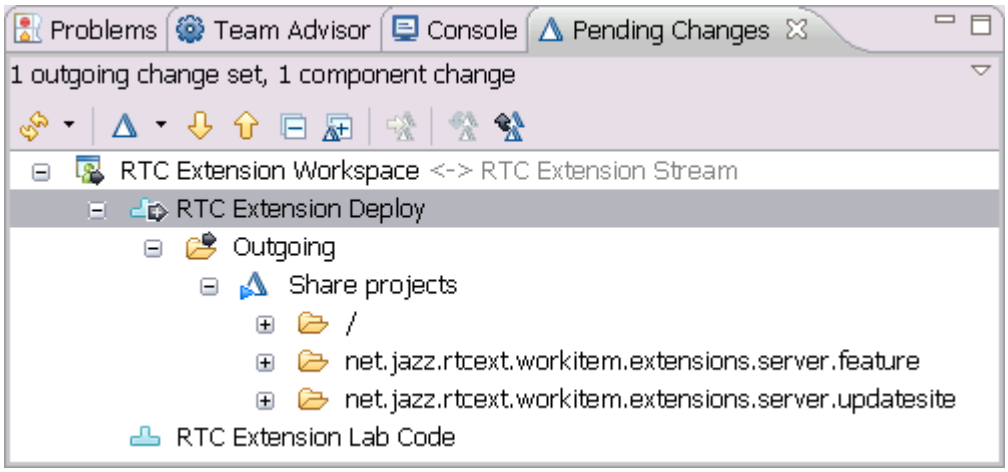
__c. On the second page of the wizard, select the **RTC Extension Workspace** (as highlighted with a red box) and click **New Component**. In the **New Component** dialog, enter `RTC Extension Deploy` as the component name and click **OK**. Finally, back to the wizard, make sure the new component is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

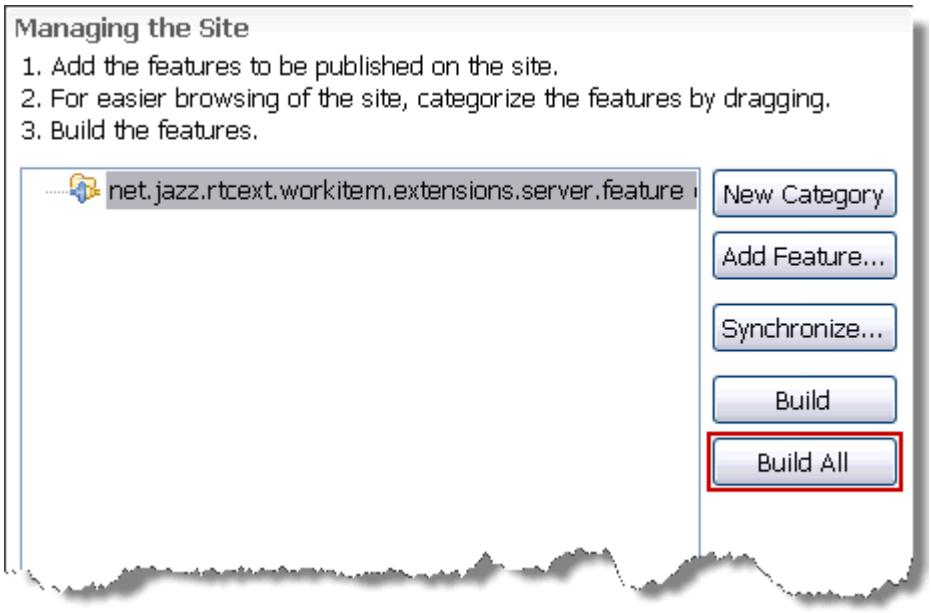


__e. The **Pending Changes** view will show your outgoing component addition with its newly shared projects. You will deliver them later.

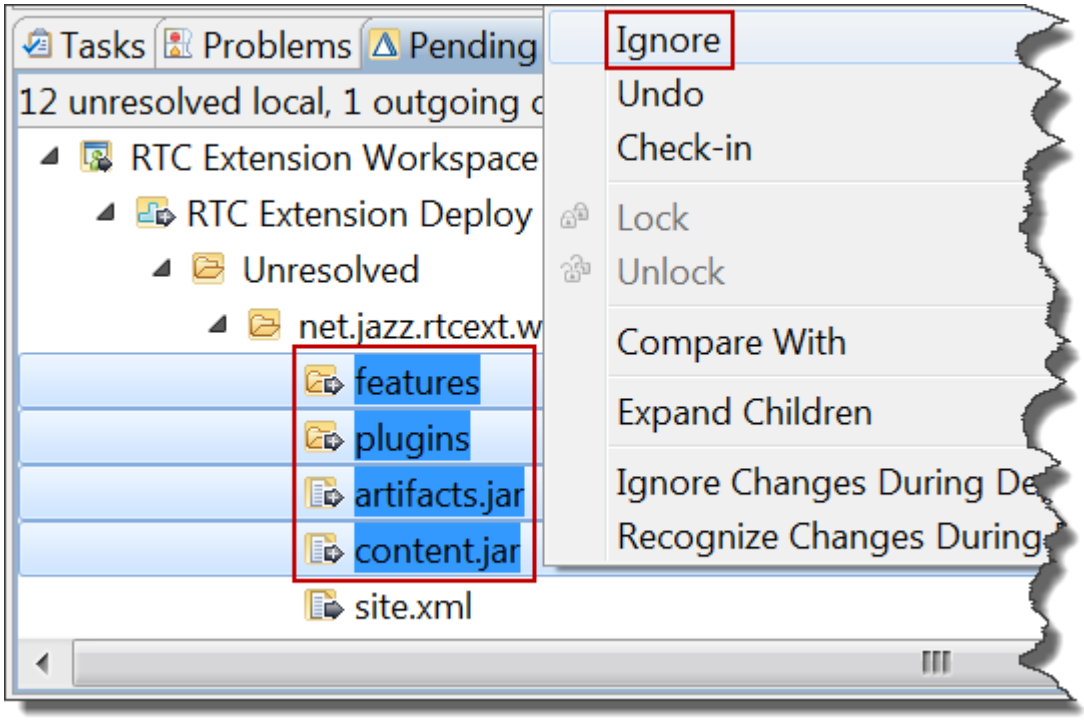


__19. Build the update site.

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

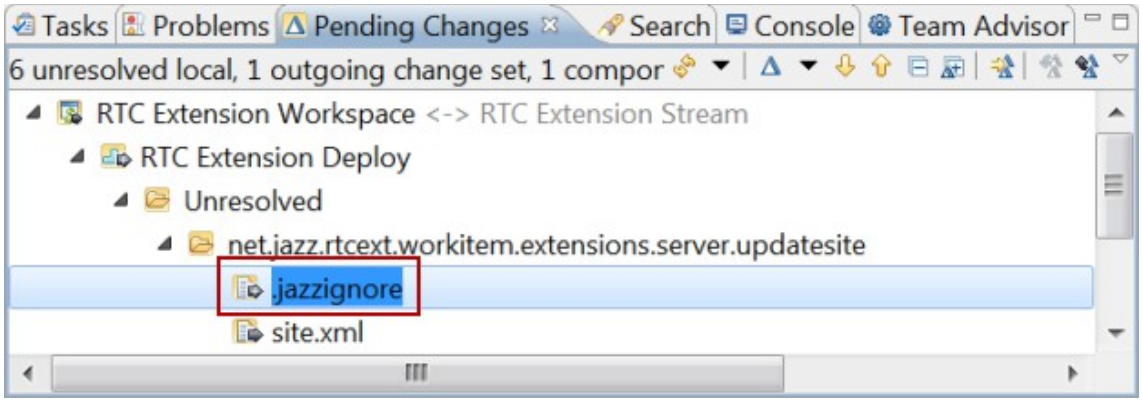


__b. The **Package Explorer** and **Pending Changes** views will show several new files in your update site project. In the Pending Changes view they will show up as **Unresolved**. Select the four entries in the root of the update site as shown here (note that site.xml is not selected). Then, right 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. Also, you do not want to check-in the change to the site.xml file but you do not want to ignore the file either. The version that was shared before and is already in the repository workspace is the one you want. The site.xml file is both a build input and output when an update site is built. You want to build input version under source control, not the build output version.

__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 ("Share projects" is the name of the change set created when you shared the two projects into the RTC Extension Deploy component). Note that site.xml is not selected.



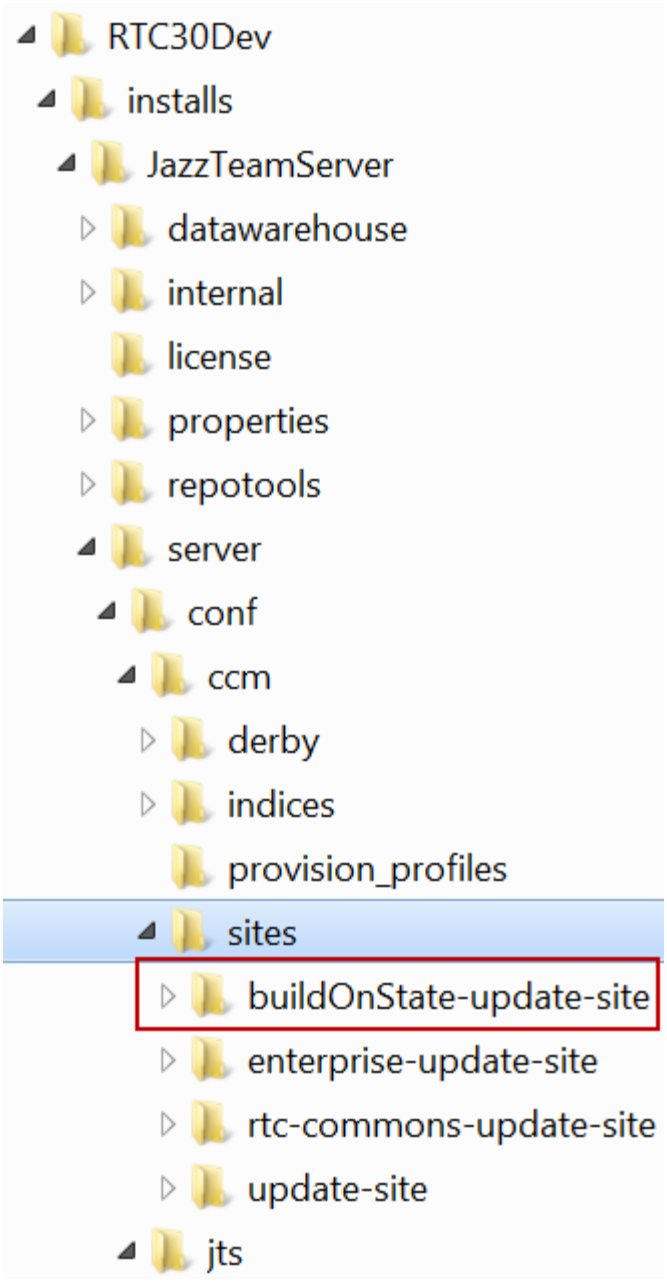
__d. If you now dig into the site.xml file and into the jars in the features and plugins folders inside the update site project, you will notice that all the update site build has converted all the "qualifier" segments of the version numbers to date and time stamps. This will make it easier to update 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

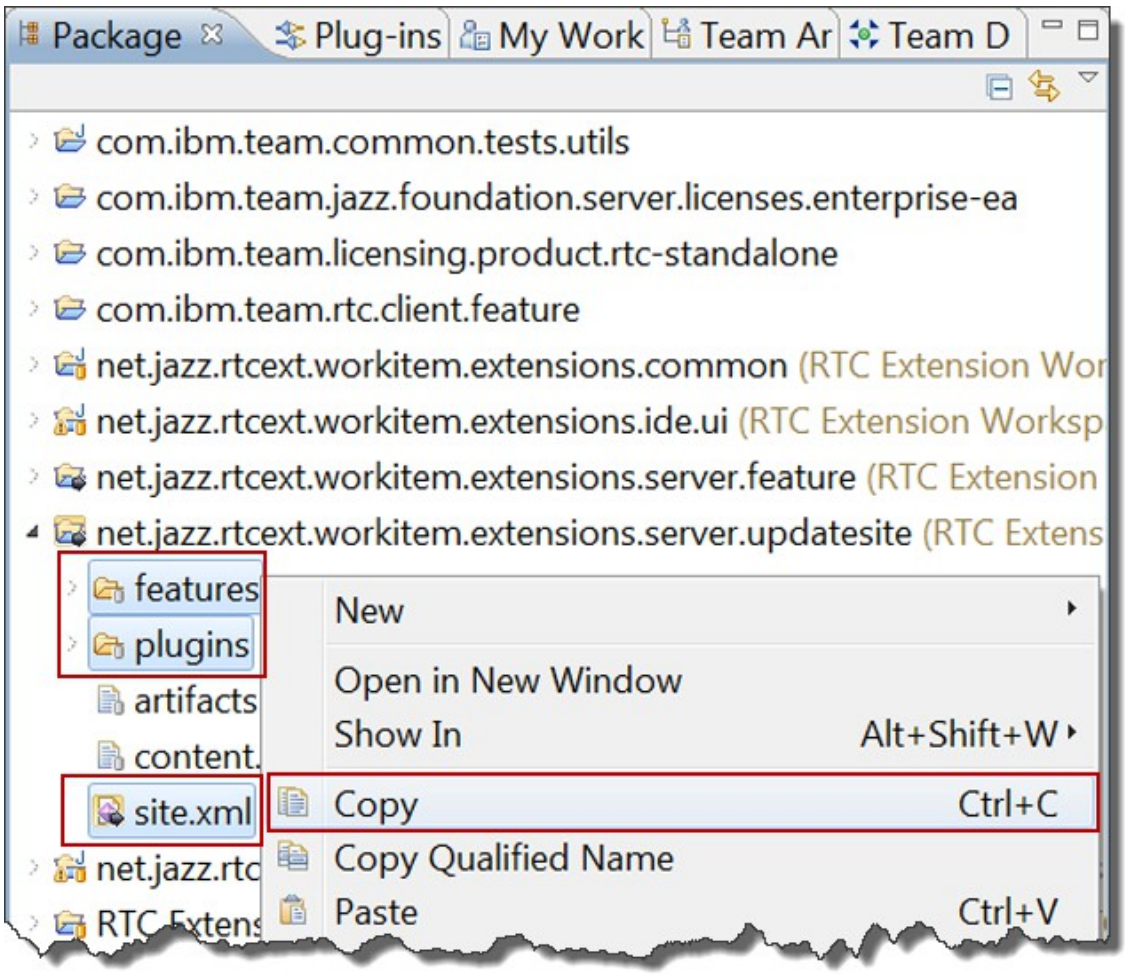
__20. Shutdown the RTC server and copy the update site into place.

__a. In the Windows Explorer, navigate to `C:\RTC40Dev\installs\JazzTeamServer\server` and run the **server.shutdown.bat** file.

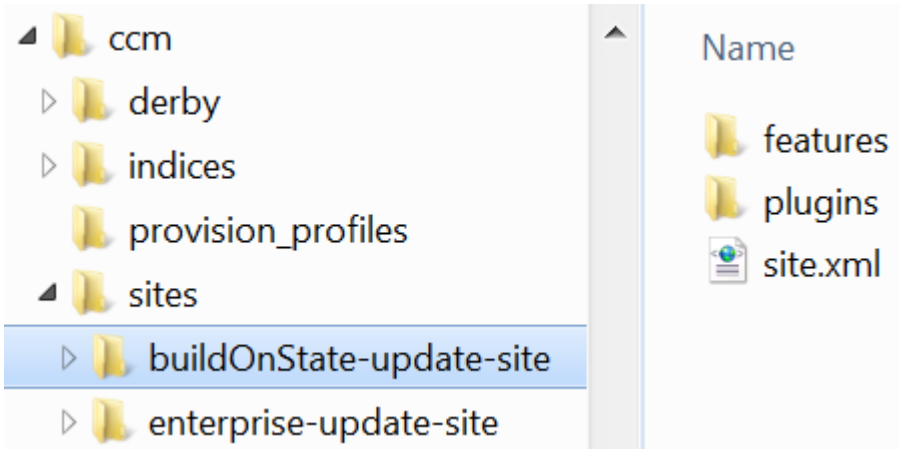
__b. In the Windows Explorer, navigate to `C:\RTC40Dev\installs\JazzTeamServer\server\conf\ccm\sites` and create a new folder to contain the extension. For this lab, call it `buildOnState-update-site` as shown here. Be sure you are in the ccm application configuration and not the jts application configuration.



__c. 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.



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



__21. Create the provisioning ini file.

__a. In the Windows Explorer, navigate to `C:\RTC40Dev\installs\JazzTeamServer\server\conf\ccm\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:

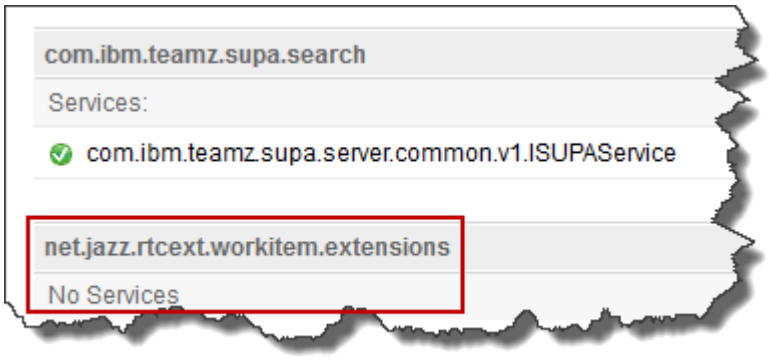
```
url=file:ccm/sites/buildOnState-update-site
featureid=net.jazz.rtcext.workitem.extensions.server.feature
```

__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.

__22. Start the RTC server.

__a. In the Windows Explorer, navigate to `C:\RTC40Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file.

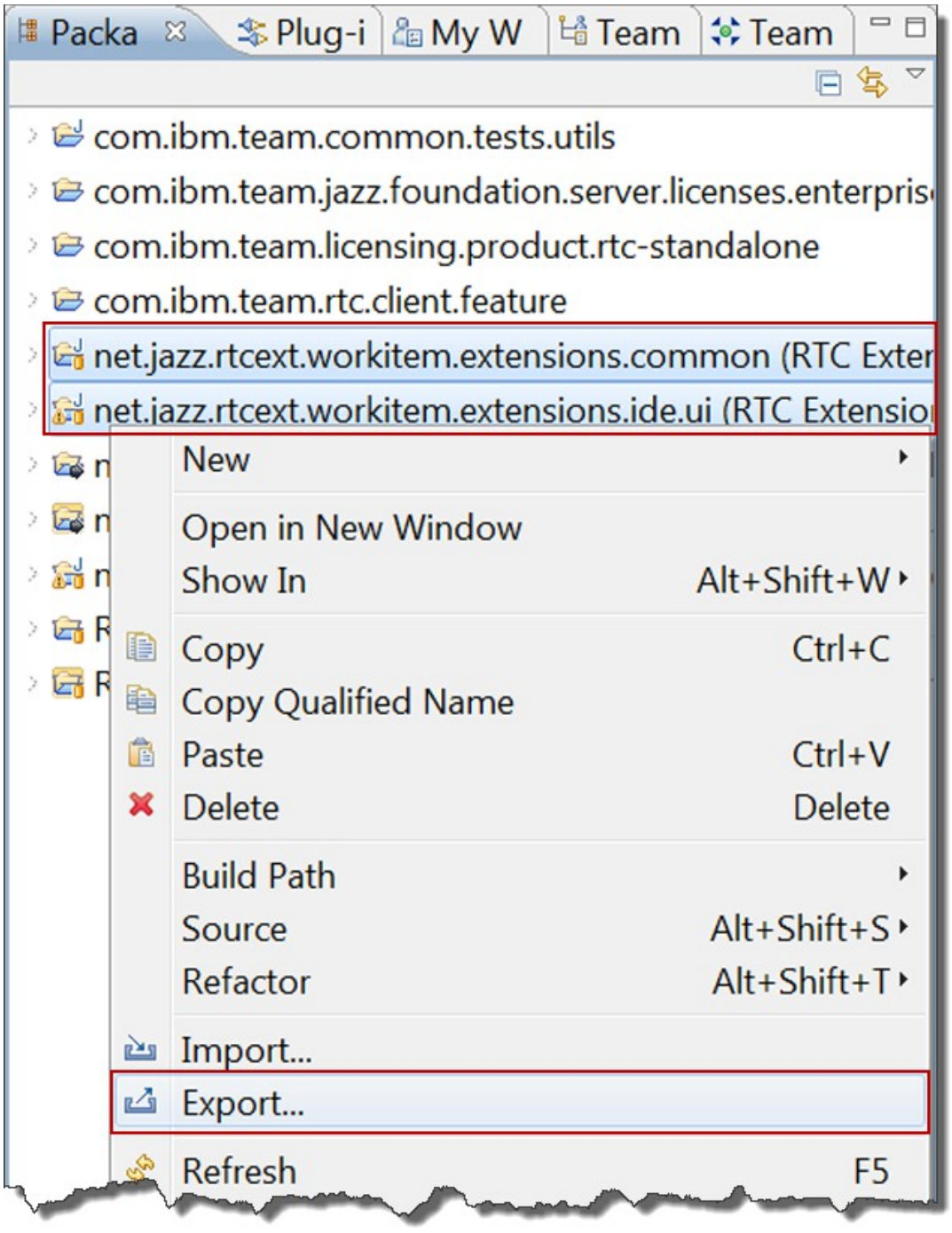
__b. If you open your browser to this URL https://localhost:9443/ccm/admin#action=com.ibm.team.repository.admin.componentStatus , login as `myadmin / myadmin`. Use the browser text search to search for **rtcext** or go to the end of the page. You will see the **net.jazz.rtcext.workitem.extensions** component is running. It does not show any services since it just contains the operation participant.



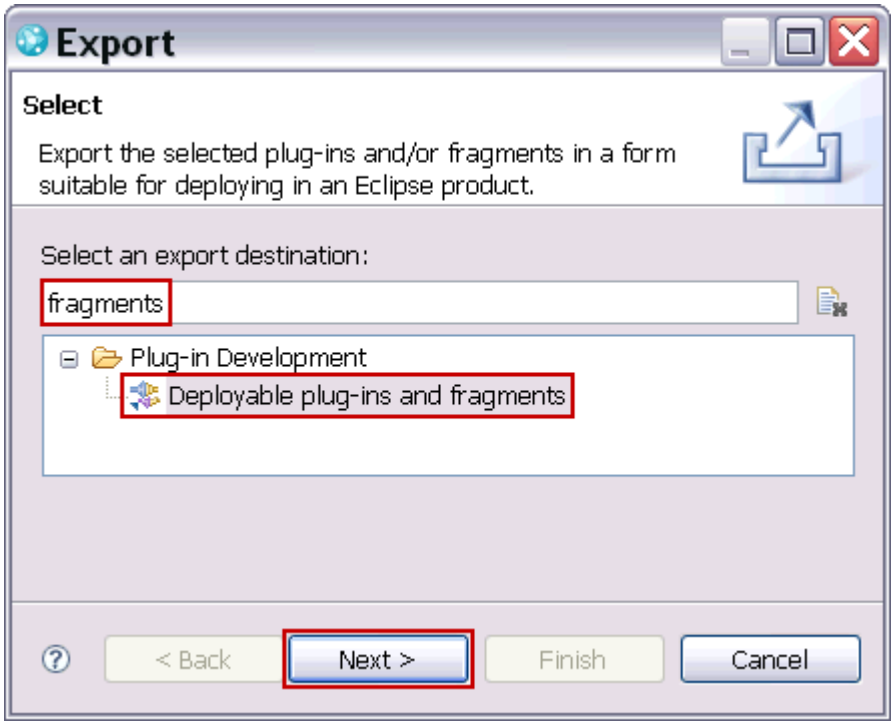
6.4 Deploy the Client Plug-ins

__23. Export deployable plug-ins to the drop-ins folder.

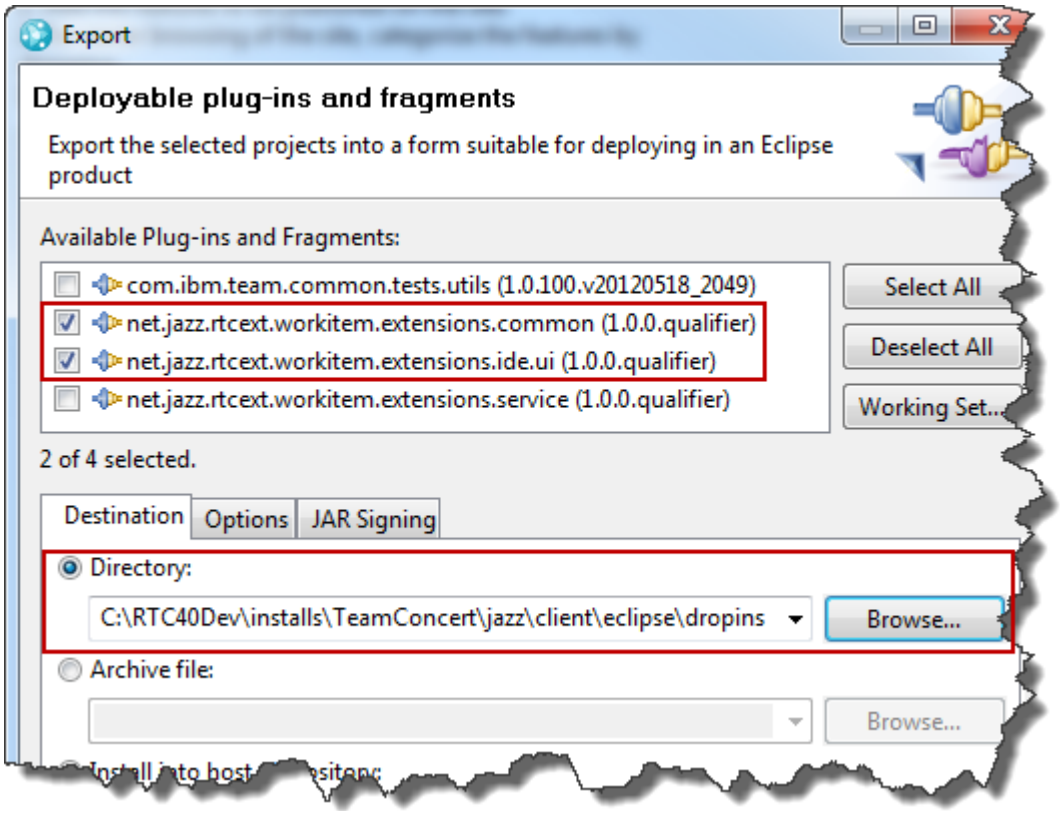
__a. In the **Package Explorer** view, select the common and ui plug-ins as shown here and then right click one of them and select **Export…**



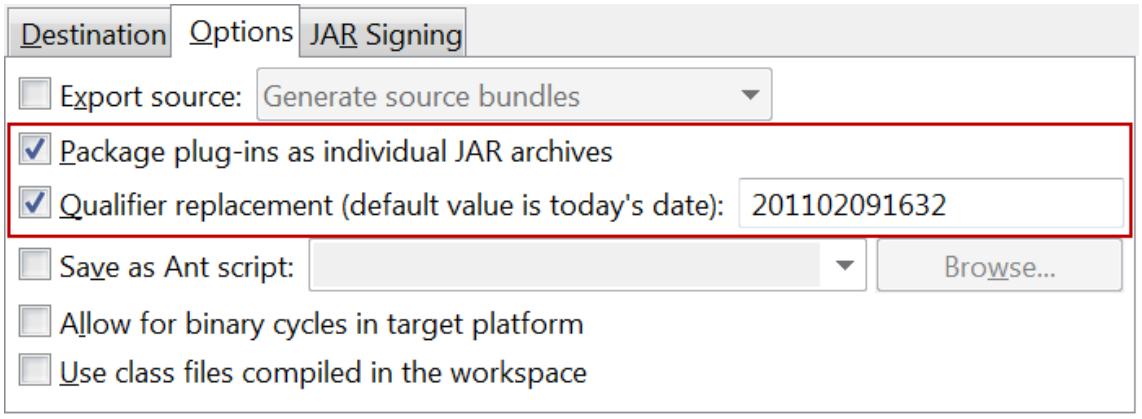
__b. In the **Export** wizard, type fragments in the filter, select **Deployable plug-ins and fragments** from the list and then click **Next**.



__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**.



__24. Restart the RTC Eclipse client.

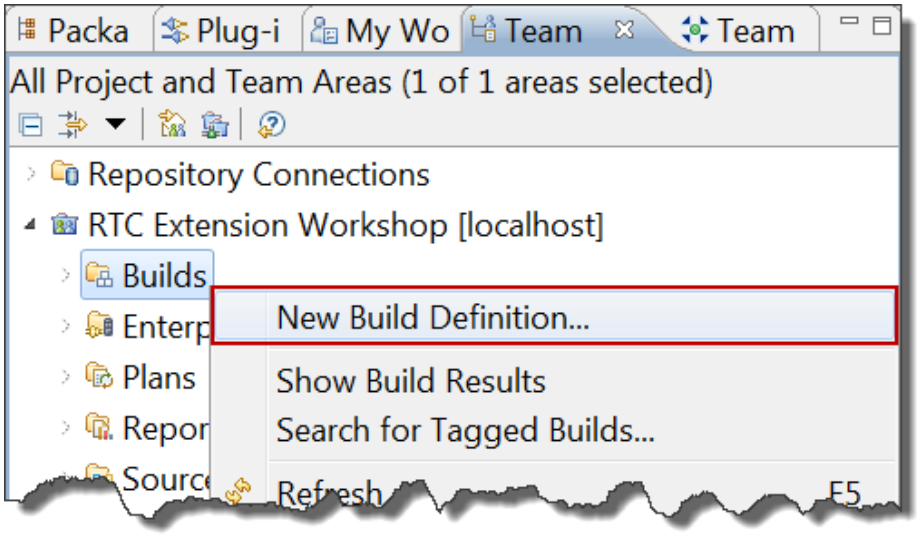
__a. Close your RTC Eclipse client.

__b. In Windows Explorer, navigate to `C:\RTC40Dev\installs\TeamConcert\jazz\client\eclipse` and double click **eclipse.exe**.

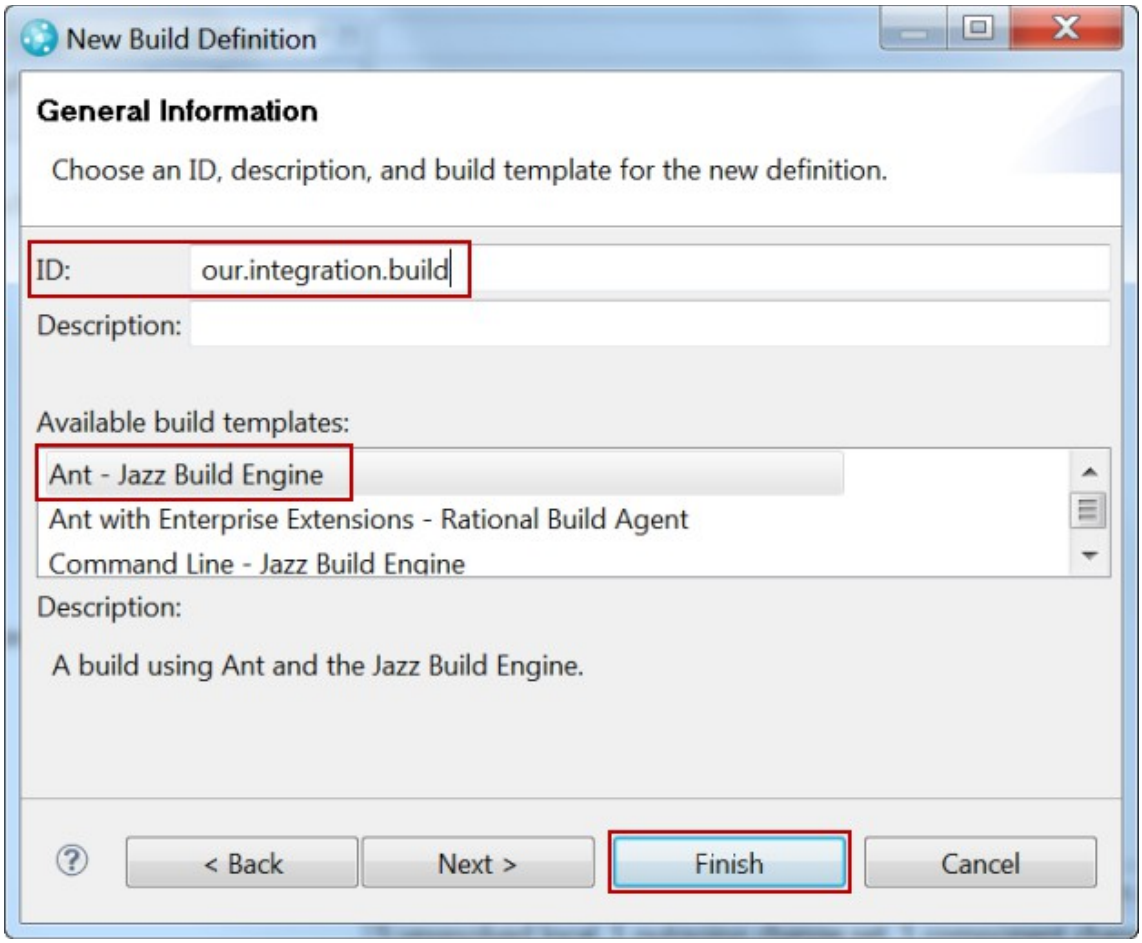
6.5 Test the Deployed Participant

__25. 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.

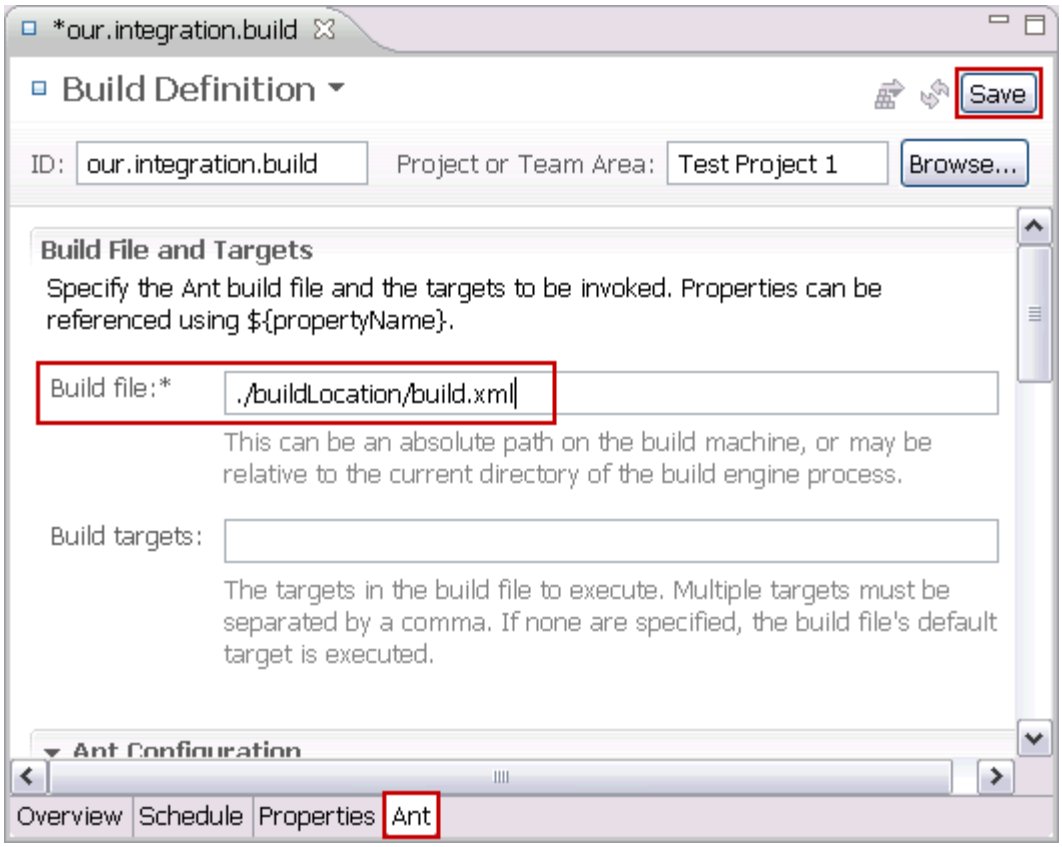
__a. In the **Team Artifacts** view, expand the **RTC Extension Workshop** node, right click **Builds** and then click **New Build Definition…**



__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**.

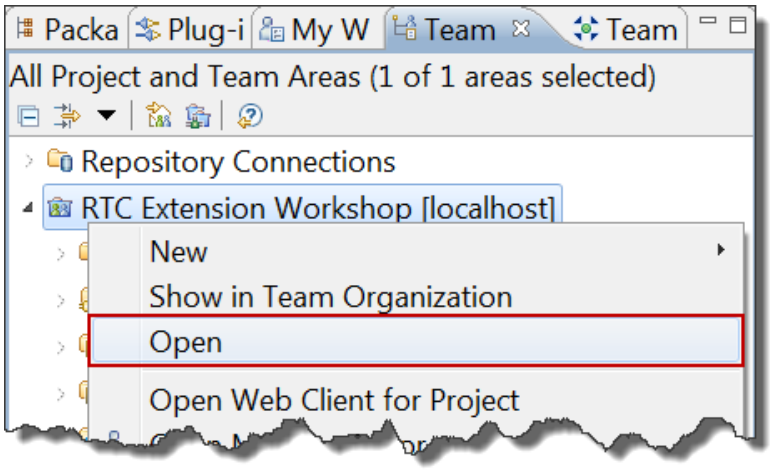


__c. In the build definition editor that opens, switch to the **Ant** tab, and enter a path for the **Build** file and then click **Save**. You may now close the editor. Note that the build file does not exist and any path will work for the current purpose. If you wish, you can use the path shown, `./buildLocation/build.xml`. 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.

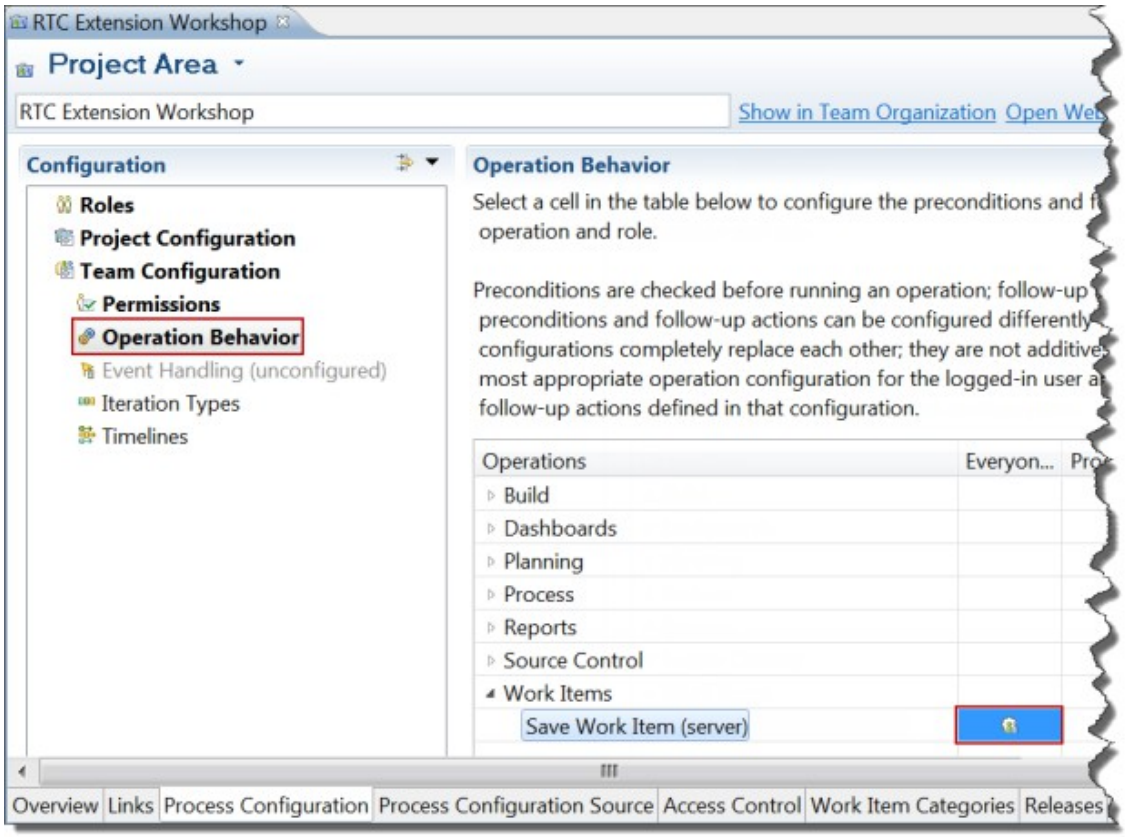


__26. Add the follow-up action to the project area.

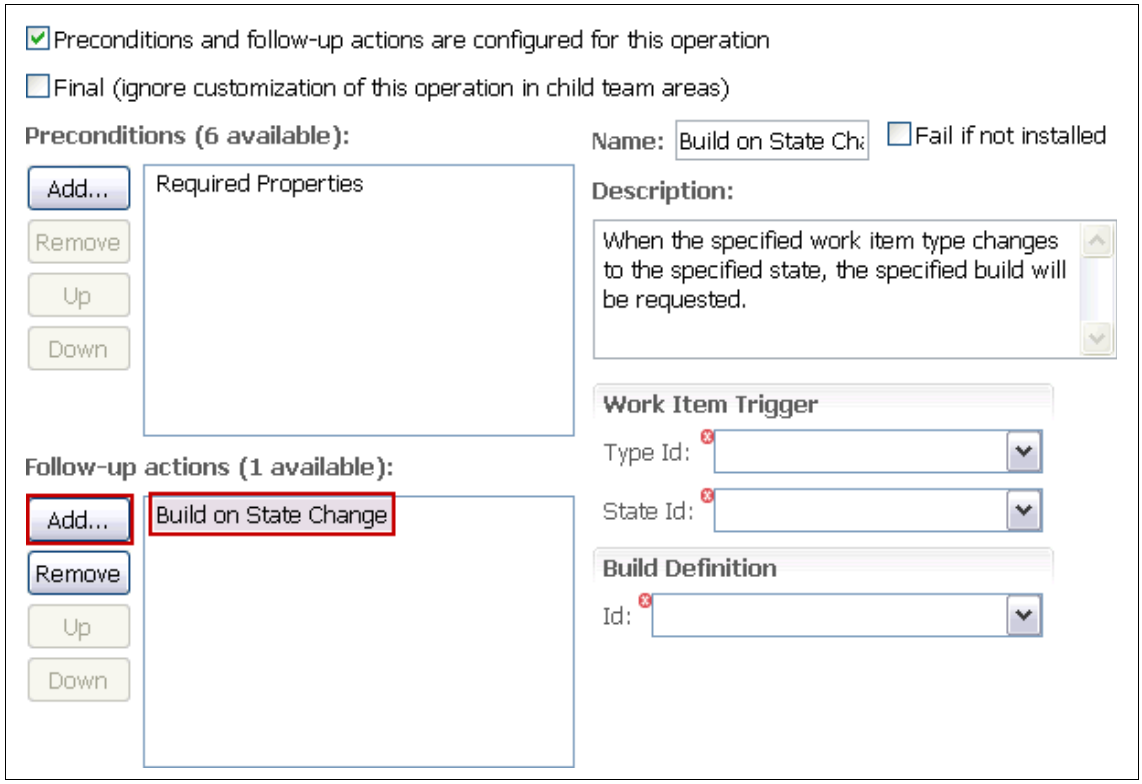
__a. In the **Team Artifacts** view, right click the **RTC Extension Workshop** project area and select **Open** from the menu.



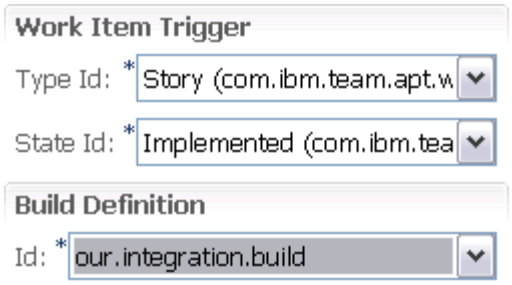
__b. In the project area editor, switch to the **Process Configuration** tab and then on the left, expand the **Team Configuration** tree then select **Operation Behavior**. Then, on the right, scroll down to the **Work Items > Save Work Item (server)** operation and select the **Everyone (default)**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



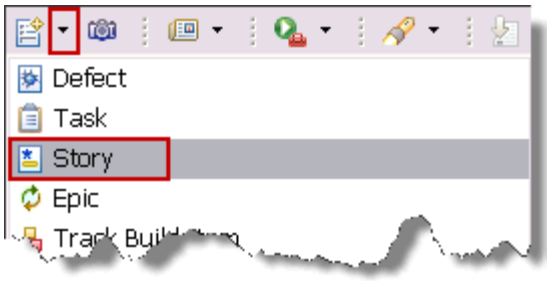
__d. Fill in the **Work Item Trigger** as shown here. You may, of course, choose different values for the work item type and state, but then you will need to adjust the following steps accordingly.



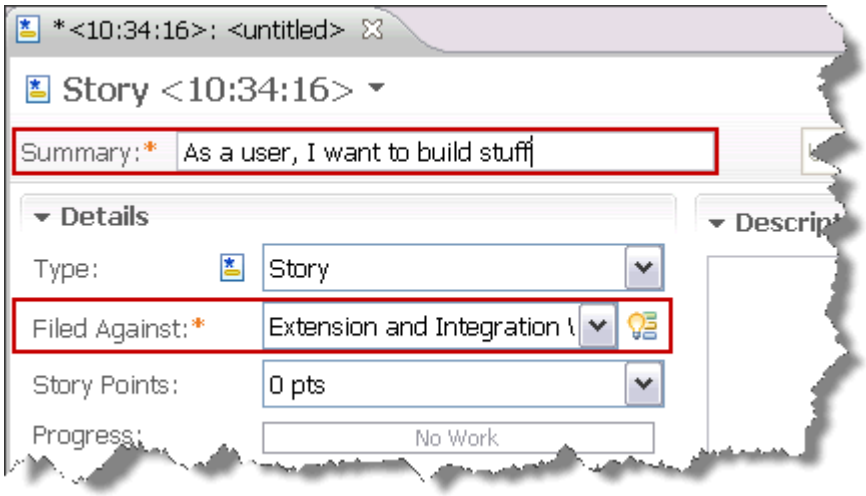
__e. Click **Save** at the top right of the editor. You may now close the project area editor and any other editors that may still be open.

__27. Create a Story and move it to the target state.

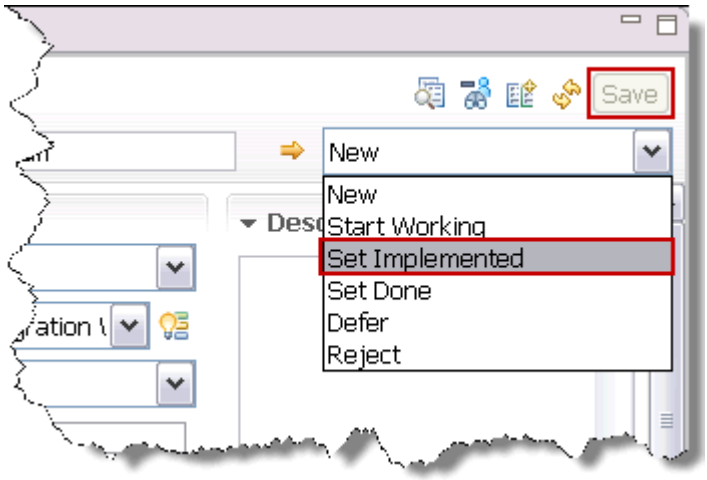
__a. Click the dropdown menu arrow next to the **New Work Item** toolbar icon and then click **Story**.



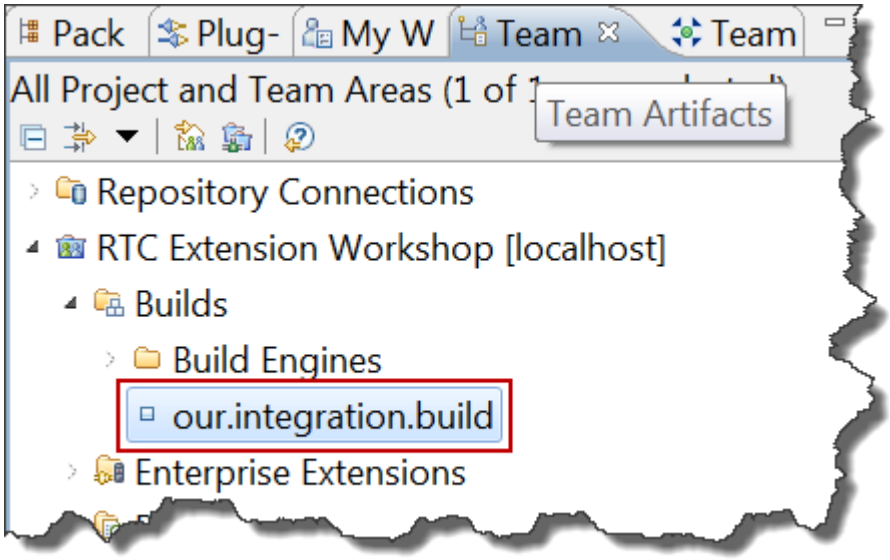
__b. In the new work item editor that opens, set the two required fields and shown here and then click **Save** in the upper right corner.



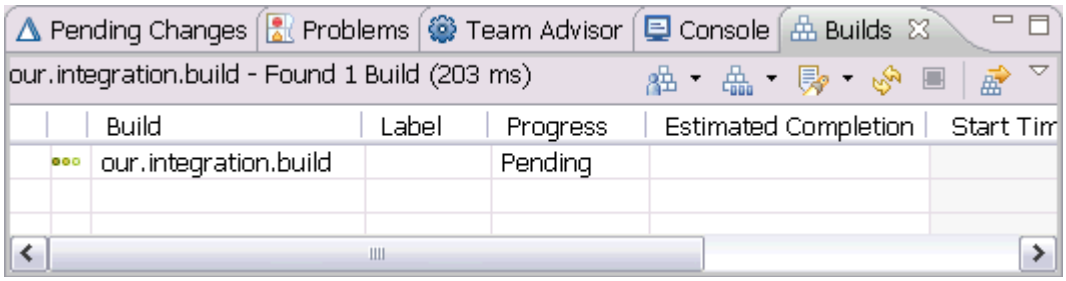
__c. At the upper right portion of the work item editor, select **Set Implemented** and then click **Save**.



__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

__28. 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.

__29. 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 ccm application you have been using in this lab, the URL would be this (https://localhost:9443/ccm/admin?internal#action=com.ibm.team.repository.admin.serverReset). 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 many of the 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.

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

NOTES

Please Recycle

Rational® software