IBM Rational Team Concert 7.x Extensibility

Lab Exercises

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/). You can find additional information for how to get started in the post https://rsjazz.wordpress.com/2015/09/30/learning-to-fly-getting-started-with-the-rtc-java-apis/ and various additional API examples in the blog https://rsjazz.wordpress.com/.

Note that these instructions are written specifically for RTC 6.0.3 on Windows®. Please adjust accordingly for different operating systems (primarily the RTC Eclipse client download and the file paths) and RTC versions (downloads). This workshop also works for the version EWM 7.0.x and later. The screen shots are still based on 6.0.3. What you see in the UI's might be different to the screen shots. The workshop works with Linux as well as with macOS considering the typical OS based adjustments.

Along with this lab document(s), you should have received or downloaded the file WorkshopSetup-V7-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 6.0.3 and higher 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 6.0.3 and higher versions of RTC. In case you have trouble, ask in the Jazz.net Forum for help. For RTC versions prior to RTC 6.0.3 download a version of the RTC Extension Workshop on Jazz.net that is compatible with your version.

The solution has been renamed beginning with version 7.0. The solution is now called Engineering Lifecycle Management and Rational Team Concert (RTC) has been renamed to Engineering Workflow Management (EWM).

The file names in downloads also reflect these changes.

This workshop has been tested with ELM 7.0, 7.0.1, 7.0.2 minor issues where found and the workshop was adjusted accordingly.

The workshop needs an external text editor in some sections. Notepad is sometimes too basic to be very effective. Try Notepad++ as a free alternative.

Some of the lab steps take a good while to perform and are resource intensive. Please be patient.

1.1 Download and Install the required files from jazz.net

Important!

Please note, the install description for the RTC server below is for the plain zip files. 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
2. You can install from the Web Installer

If choosing one of these options on Windows change the Shared Resources Directory to be outside the Program Files or Program Files (x86) directories. Don't install the server or the shared folders into these directories. They 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 as an administrator, the default when starting an application is to not run it as an administrator.

Separate RTC versions using different folder names

Use a different folder name for each RTC version, i.e. **RTC600Dev** for RTC 6.0 to be able to maintain different versions. Replace the folder name used in the workshop instructions with your version if following this approach.

Download into a special folder

Download the files into a special download folder so you have easy access. For example download into C:\RTC603Dev\downloads

---

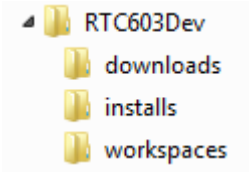
Follow the instructions for extracting the RTC SDK

Learn more about extracting the SDK .zip and the included gifar file. As mentioned here, one zip extraction tool that works is 7Zip.

For Linux and macOS download the Linux versions of the Jazz product install and supporting files.

__1. Set up a directory structure to contain your extensions development and test environment. To be able to do this for different versions of RTC, experience shows that it is a good idea to code the version into the root folder.

__a. Developers 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.



__b. The RTC603Dev root folder will contain all your work for this workshop.

__i. The `installs` folder will be the target of the product installations you are about to perform.

__ii. The `workspaces` folder will contain your Eclipse workspace(s) and other related folders.

__iii. The `downloads` folder is a good destination for the files you download in the next section.

Version dependent root folder Name

For versions such as 5.0, 6.0, 7.0 choose a folder name such as **RTC500Dev**, or **RTC600Dev** this way the folders show up correctly ordered by version. When using RTC60Dev, the lexicographic order is mixed up for the versions with two and three digits.

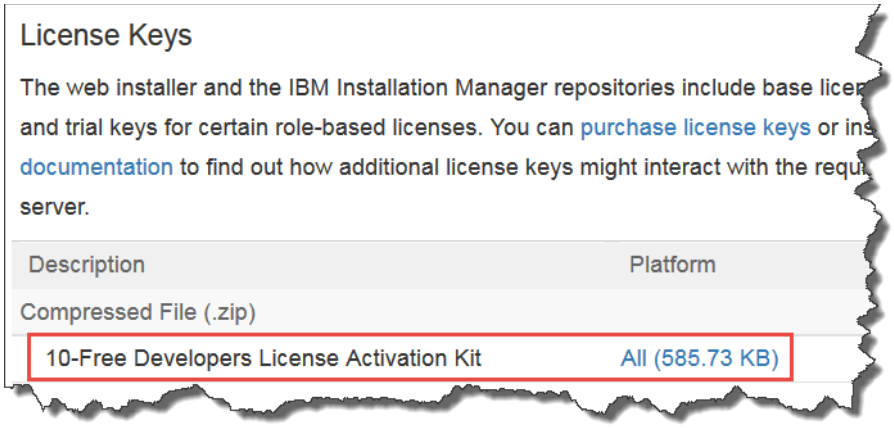
__c. Create a folder `C:\RTC603Dev` and create the folder `C:\RTC603Dev\downloads`.

__2. Download the product installation and license files. You will need a jazz.net id for all the download operations. Please download into a separate folder `C:\RTC603Dev\downloads` or make sure to know the file names and locations for the downloads to copy them there later.

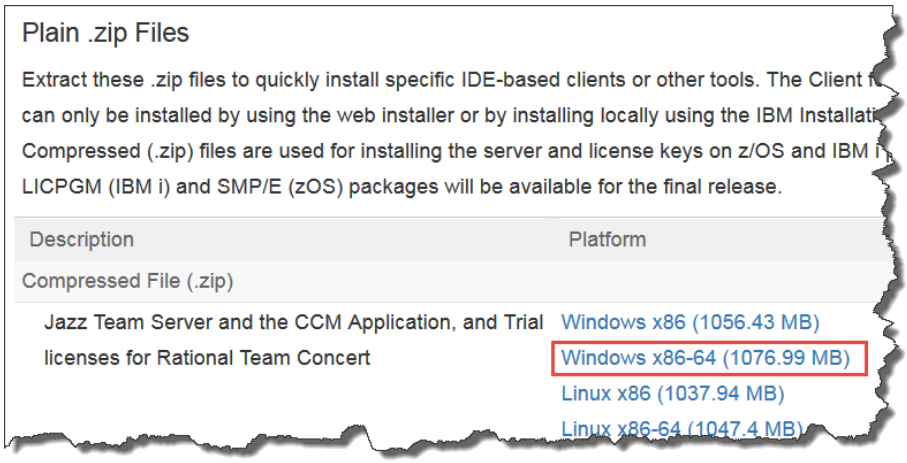
__a. Go to the RTC all downloads page for your version of RTC. As example for 6.0.3 at https://jazz.net/downloads/rational-team-concert/releases/6.0.3?p=allDownloads.

For EWM version 7.0.2 the link looks as follows: https://jazz.net/downloads/workflow-management/releases/7.0.2?p=allDownloads.

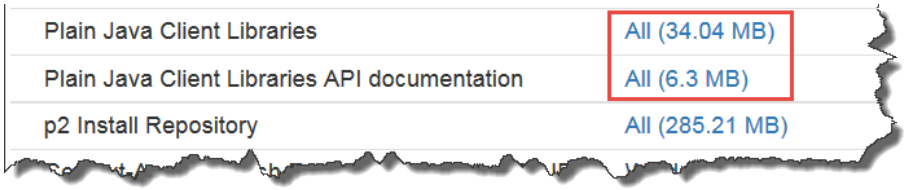
__b. Scroll down to the **License Keys** section and download the highlighted file with the 10-Free Developers License Activation Kit.



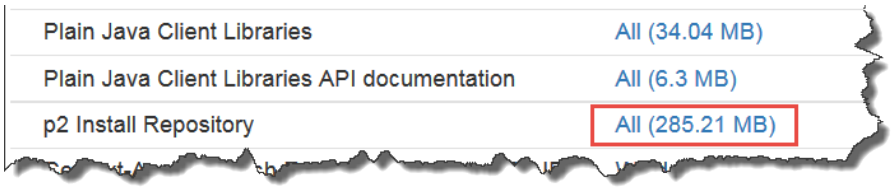
__c. Scroll down to the **Plain .zip files** section and download the highlighted file with the plain zip version of the Jazz Team Server and the CCM Application, and Trial licenses for Rational Team Concert. There are other options for download and install of the RTC server you can use. This workbook will however use this method.



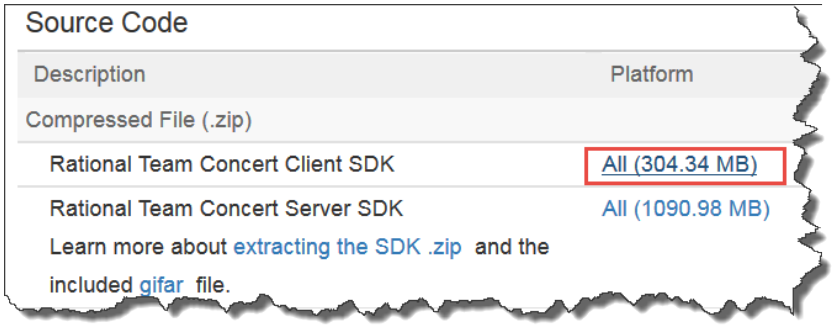
__d. In the same section download the highlighted zip files for the Plain Java Client Libraries.



__e. Download the highlighted zip files for the p2 Install Repository for the RTC Eclipse client in the same section.

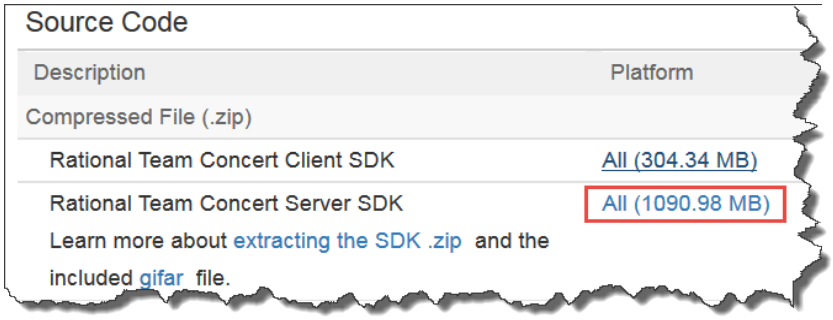


__f. Scroll down to the **Source Code** section and download the highlighted file for the Rational Team Concert Client SDK or since version 7 IBM Engineering Workflow Management Client SDK.



Follow the instructions to extract the RTC SDK provided for the download “Learn more about extracting the SDK .zip and the included gifar file” and get the zip tool mentioned installed. The zip tool shipped with windows will not be able to extract the SDK. The tool mentioned in the link can be used for all extraction steps in this workshop.

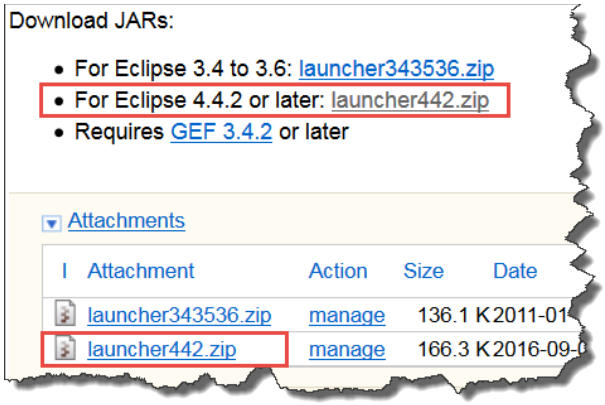
__g. In the **Source Code** section download the highlighted file for the Rational Team Concert Server SDK or since version 7 IBM Engineering Workflow Management Server SDK.



__3. Download the feature based launches Eclipse capability extension.

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

Scroll down to the end of the page. Download the latest available version of the feature based launcher for Eclipse 4.4.2 or later. At the time of writing the version is launcher442.zip.



__4. Download the Eclipse client from http://www.eclipse.org/. With RTC 6.0.3 and higher you can use Eclipse 4.4.2 or higher.

32bit or 64bit?

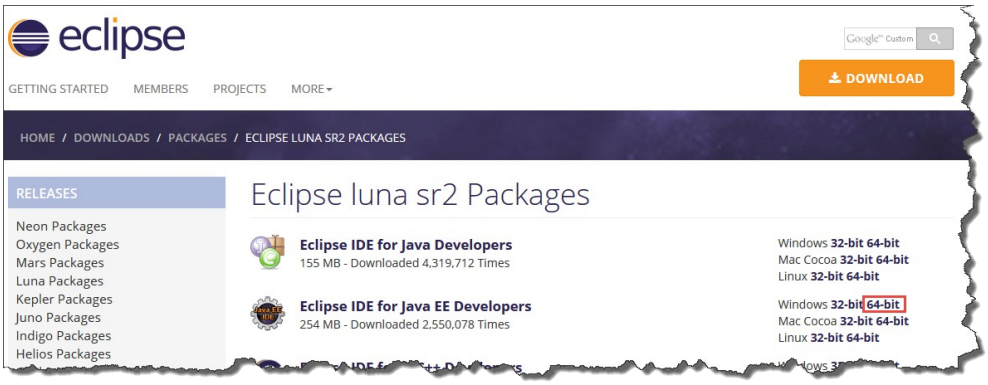
You should use a 64 bit version of Java and Eclipse. These instruction where created using a 64 bit version.

The downloads for the Jazz Team Server and the CCM Application contains a valid IBM Java JRE. The Client for Eclipse IDE contains a valid IBM Java JDK.

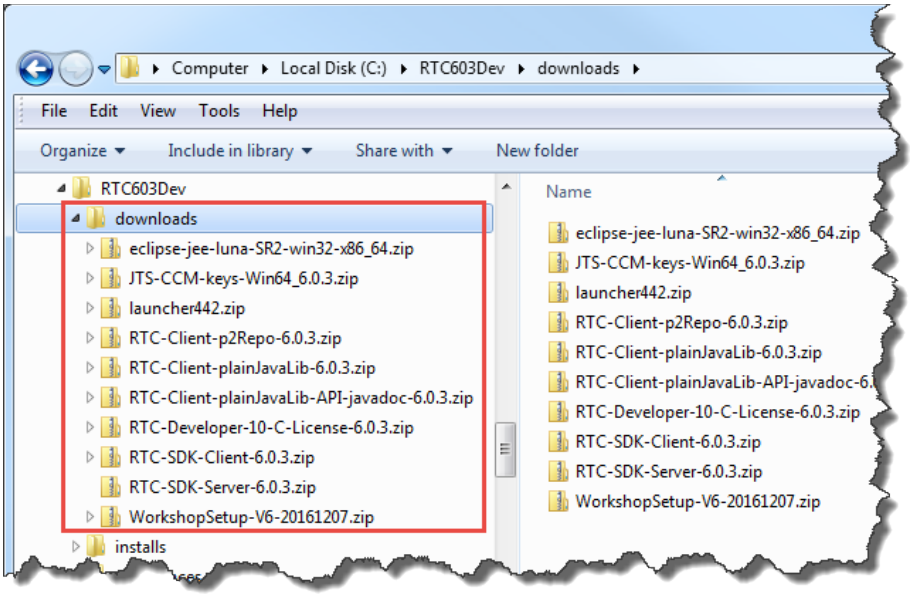
For Linux and macOS download the 64 bit Linux or macOS version of Eclipse.

The Eclipse versions that can be downloaded usually use a release name and don't provide a version number for the Eclipse release on the download. If you are interested in finding a specific version, you can find the mapping of version number to the release name in this page and in this Wikipedia Entry in the Releases section.

To download Eclipse 4.4.2 go to the page https://www.eclipse.org/downloads/packages/ and look for the Luna release or directly to release Luna SR2. Download the **Eclipse IDE for Java EE** Developers package.



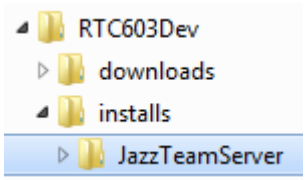
__5. If you did not already download there, move the files you downloaded before into the folder `C:\RTC603Dev\downloads`. This makes it easy to setup and redo the workshop with this version of RTC later. Mind the name change in version 7.



__6. Install the RTC Development Server.

__a. Extract the plain zip Jazz Team Server and the CCM Application, and Trial licenses for Rational Team Concert you downloaded. The file usually is named like JTS-CCM-keys-Win64_6.0.3.zip. Extract to `C:\RTC603Dev\installs\JazzTeamServer`.

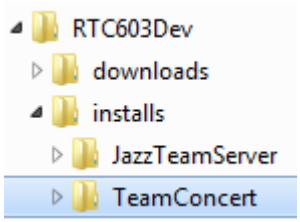
__b. Your `C:\RTC603Dev` folder will now look like below.



__7. Install the Eclipse client.

__a. Extract the Eclipse 4.4.2 you downloaded. The file is usually named like eclipse-standard-luna-SR2-win32-x86_64.zip. Extract the content to `C:\RTC603Dev\installs\TeamConcert.`

__b. Your `C:\RTC603Dev` folder will now look like below.



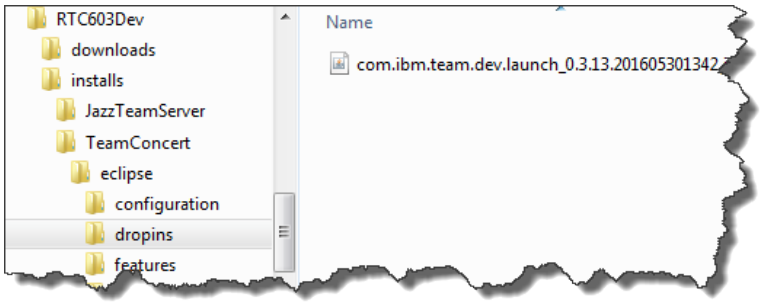
__8. Add the feature based launches capability to the RTC Eclipse client.

__a. Extract the feature based launches download file `launcher442.zip` into the containing folder, for example `C:\RTC603Dev\downloads.`

__b. In the extracted folder (e.g. `launcher442\plugins`) locate the JAR file `com.ibm.team.dev.launch_0.3.13.201605301342.jar.`

__c. Copy the JAR file into the Eclipse dropins folder located at `C:\RTC603Dev\installs\TeamConcert\eclipse\dropins`. Create the `dropins` folder if it does not exist.

__d. Your `C:\RTC603Dev` folder will now look like below.



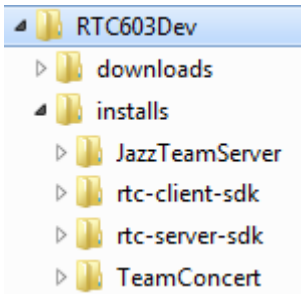
__e. Note that some users on Linux have reported that the file permissions on the jar placed in the dropins folder are set to 755 and that the feature based launches would not show up in the RTC Eclipse client until the permissions were changed to 644.

__9. Unzip the Server and Client SDK files.

__a. Extract the RTC Server SDK zip file `RTC-SDK-Server-6.0.3.zip` into the folder `C:\RTC603Dev\installs\rtc-server-sdk`. This zip file has path lengths longer than 250 characters and may cause trouble for some extractor tools on Windows. As mentioned above, one zip extractor tool that works is 7Zip.

__b. Extract the RTC Client SDK zip file named `RTC-SDK-Client-6.0.3.zip` into the folder `C:\RTC603Dev\installs\rtc-client-sdk`. The limitation for zip tools mentioned above applies here too.

__c. Your `C:\RTC603Dev` folder will now look like below.

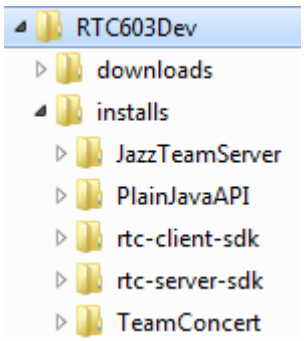


__10. Install the RTC Plain Java Client Libraries needed to setup the workshop.

__a. Extract the RTC Plain Java Client Libraries from the file `RTC-Client-plainJavaLib-X.Y.Z.zip` into the folder `C:\RTC603Dev\installs\PlainJavaAPI`.

__b. Extract the RTC Plain Java Client Libraries API Documentation from the file `RRTC-Client-plainJavaLib-API-javadoc-X.Y.Z.zip` into the folder `C:\RTC603Dev\installs\PlainJavaAPI`.

__c. Your `C:\RTC603Dev` folder will now look like below.

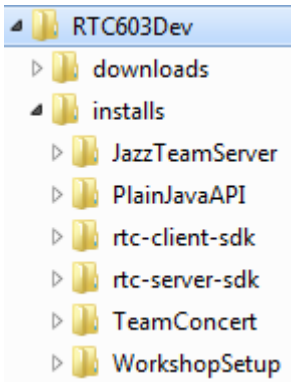


__11. Install the Workshop Setup tool.

__a. Along with this lab document(s), you should have received or downloaded the file `WorkshopSetup-V6-YYYYMMDD.zip`. You can find a version that is compatible with this workshop for download in the RTC Extension Workshop on Jazz.net.

__b. Unzip this file to `C:\RTC603Dev\installs\.`

__c. Your `C:\RTC603Dev\installs\` folder should now look like below.



__12. Prepare Eclipse for running with the RTC Server JRE and tweak the display to show the active Eclipse workspace.

__a. Navigate to the folder `C:\RTC603Dev\installs\TeamConcert\eclipse`

__b. Open the file eclipse.ini with a text editor for inspection and editing.

__i. Add the following lines at the top of the `eclipse.ini` file

```
-vm
C:/RTC603Dev/installs/JazzTeamServer/server/jre/bin
```

This provides the Eclipse client with the RTC Server JRE to run. You can run with other JRE's but it is a good idea to compile the developed extensions with the server JRE and you would have to add this JRE later in Eclipse. Eclipse will not run without a JRE. If it is not provided here it has to be provided in your environment. Note that it is also possible to use the bundled RTC Eclipse client, that contains a JDK.

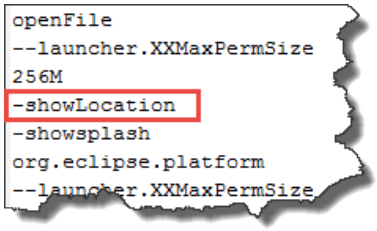
Eclipse fails to start

You have to make sure to provide a compatible Java VM. If you download a 64 bit version of Eclipse you have to provide a 64 bit version of Java.

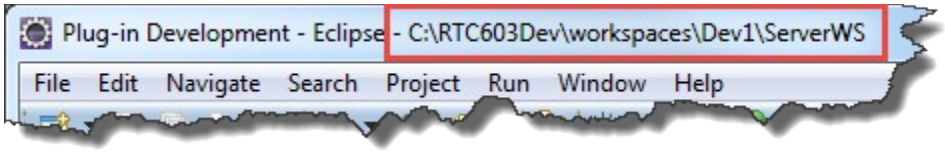
IBM Java can be downloaded from here in the Eclipse section.

The downloads for the Jazz Team Server and the CCM Application contains a valid IBM Java JRE. The Client for Eclipse IDE contains a valid IBM Java JDK.

__i. Add the option `-showLocation`. This option later shows the Eclipse workspace path in the Eclipse main window title area and helps with working with multiple Eclipse workspaces like in our case. For me adding the line in front of the line showing `-showsplash` has worked. The image below shows how this looks like.



The following screen shot shows the window title displaying the workspace used by this Eclipse instance. This is very useful as we will be working with multiple workspaces.



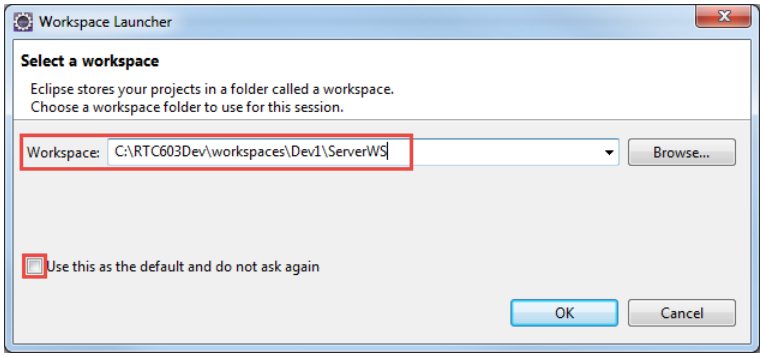
__ii. Optional, add the option `-Duser.language=en` If you want a consistent English menu language and happen to work on a machine with other regional or language settings. Add it as last option behind the `-vmargs` section

__13. Install the RTC Client from the P2 Install into Eclipse.

__a. Start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

__b. When prompted, select an Eclipse workspace. The first Eclipse workspace is going to be used for development for the Server API.

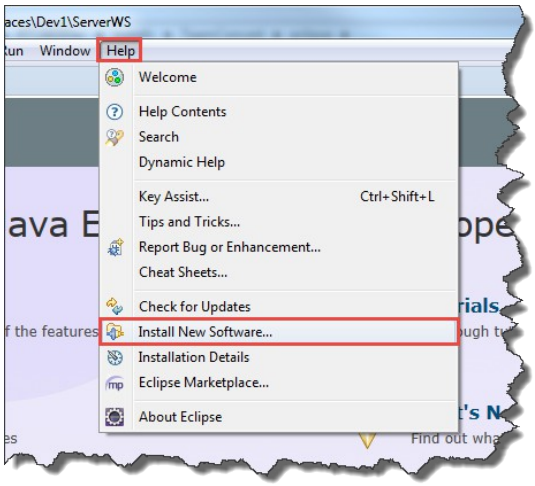
__i. Enter `C:\RTC603Dev\workspaces\Dev1\ServerWS` as path for the Eclipse workspace. Don't check the “Use as default” check box. Click OK.



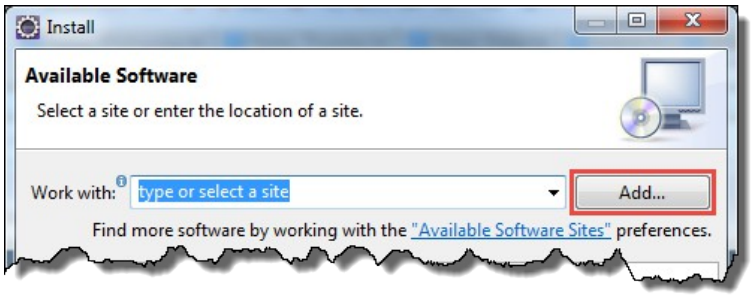
__ii. Note that it is “`Dev1\ServerWS`” and not “`Dev1ServerWS`”. Either would work, but by default when you launch a runtime or debug session the Eclipse workspace for the launched process is created as a sibling to your workspace folder.

By using the “`Dev1\ServerWS`” technique, this runtime workspace folder is created as a peer to “`ServerWS`” inside the “Dev1” folder. This makes it easier to have other isolated development workspaces such as “ C:\RTC603Dev\workspaces\Dev2\ServerWS” 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.

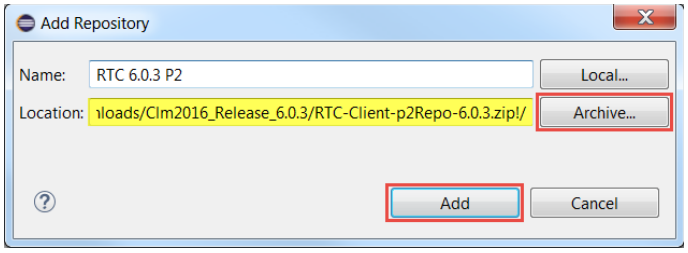
__c. Select the Menu **Help>Install New Software**.



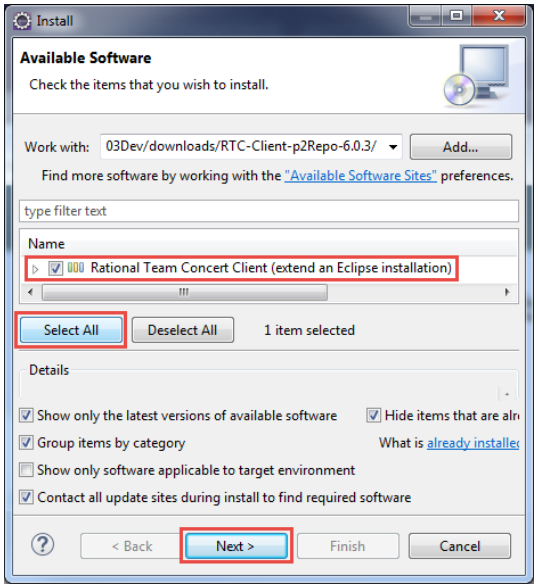
__d. Add the p2 Install Repository you downloaded and extracted in step Error: Reference source not found page Error: Reference source not found as a local location. Press **Add**.



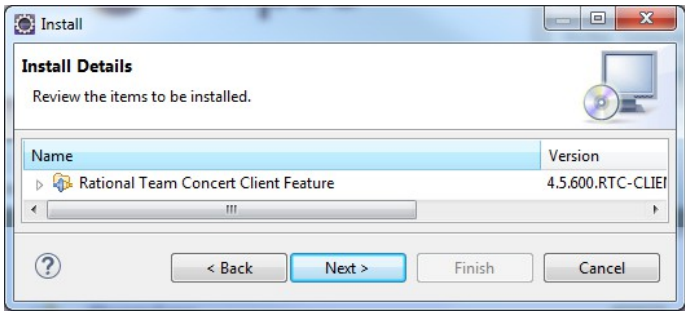
__i. Select **Add>Archive**, browse to the p2 Install Repository zip file you downloaded for example `C:/RTC603Dev/downloads/RTC-Client-p2Repo-6.0.3.zip`, add a name and press **OK**.



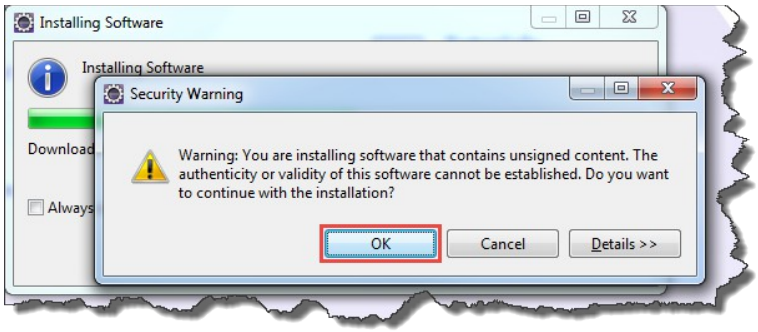
__e. Press **Select All** to select to install the Rational Team Concert Client into Eclipse. Then Press **Next**.



__f. Wait for the analysis to finish, this will take a good while, press **Next** in the Installation Details, then accept the license and press F**inish**.

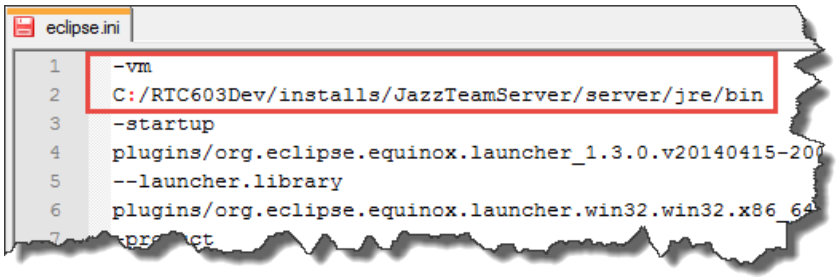


__g. If a warning about unsigned content comes up, select OK and let the install finish.



__h. Let Eclipse restart to finalize the installation and select the workspace `C:\RTC603Dev\workspaces\Dev1\ServerWS` again.

__i. Check the file `C:\RTC603Dev\installs\TeamConcert\eclipse\eclipse.ini`. Make sure the changes done above are still there and a compatible Java VM is provided. Sometimes the changes setting the virtual machine disappear. If this is the case, make sure to add the VM again.



1.2 Setup the RTC development server

You will now setup the RTC development server on WebSphere Liberty Profile. This server will be used in two roles:

1. As your RTC development server that provides you with the workshop data, in repository workspaces, streams, It also provides with Jazz SCM and a capability to version your code
2. As a test server to perform and test the final deployment of the server extension you will create in this workshop

You will enable the RTC development server for debugging. Later, you will learn how to debug on this server by attaching the Eclipse Java debugger to the running server and hitting a breakpoint.

Testing with the RTC Development server has these advantages:

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

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

You will also learn how to launch a Jetty based RTC debug server from Eclipse. This Jetty based RTC debug server will use its own separate test repository database. The Jetty based RTC debug server provide a fast and convenient way to develop and debug RTC Server extensions.

Developing, debugging and testing with a Jetty based RTC debug server has the advantages:

- Very fast turn around, including hot code update on a running server
- Delays deployment issues until the server extension is debugged
- It is easy to backup or recreate the test repository

The Jetty based RTC debug is the most convenient way to develop extensions. The only disadvantage is that you can accidentally mix client and server API which is an issue when deploying and testing on a test or production system.

__14. Setup to run the Development Server in debug mode.

__a. Open Windows Explorer and navigate to `C:\RTC603Dev\installs\JazzTeamServer\server`.

__b. Open the file **server.startup.bat** with a text editor.

__c. Find the line with the following text. It is near the end of the file in a section with many set JAVA_OPTS= statements.

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

__d. After that line add these lines.

```
set JAVA_OPTS=%JAVA_OPTS% -Duser.language=en
set WLP_DEBUG_SUSPEND=n
```

The first line makes sure the server is started with English language settings. The second setting allows the server to run normal when started in debug mode. Please check the todo's below for more information.

__e. Be sure to keep all the other options. Save your changes.

Please note, that the default debug port for WebSphere Liberty Profile profile is 7777. To keep compatibility to the previously bundled Tomcat application server, the debug port is set to 8000 in the file liberty.server.bat located in the same folder as the server.startup.bat. If you need to use a different port for some reason, you have to modify the setting there.

Please note that the default for WLP_DEBUG_SUSPEND is y, which means the server stops the startup and waits for a connection from a remote debugger before it even starts.

For Linux and macOS the lines to add are

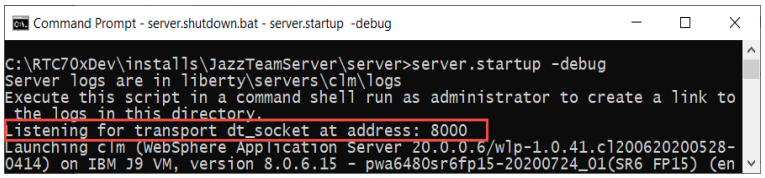
```
JAVA_OPTS=%JAVA_OPTS% -Duser.language=en
WLP_DEBUG_SUSPEND=n
export WLP_DEBUG_SUSPEND
```

__15. Complete setup of the RTC development server.

__a. Open a command window. Navigate to `C:\RTC603Dev\installs\JazzTeamServer\server`.

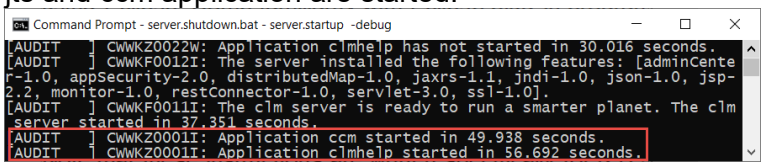
__b. Type **server.startup.bat -debug** and hit enter to run the **server.startup.bat** file (the same file you just edited) with the argument **-debug**.

Carefully observe progress in the server startup console. Note that the server is listening on the port 8000. If the console does not continue, make sure you have saved your recent changes to the server.startup.bat file. Close the command prompt and repeat the last three steps to retry starting the server.



__c. It is likely that Java requests network permissions. Private network would be OK for the workshop.

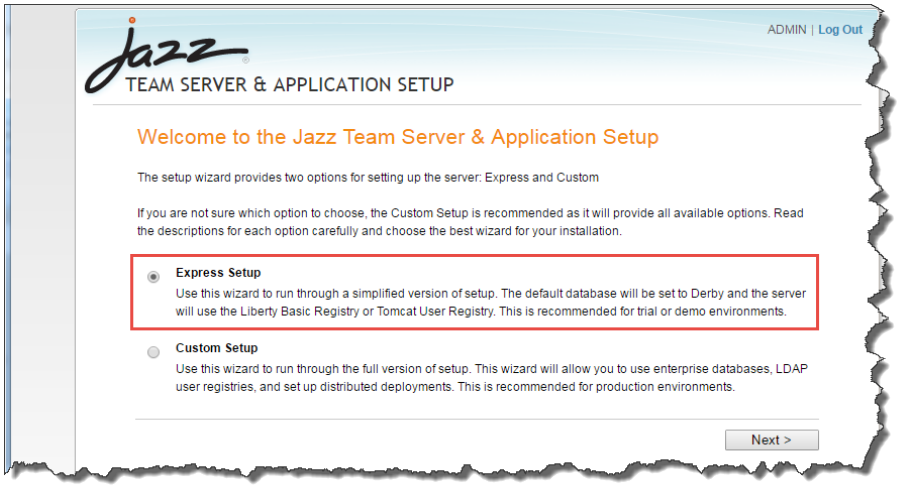
__d. Make sure the RTC development server has finished starting. This is the case after the jts and ccm application are started.



__e. Start your browser and enter the URL https://localhost:9443/jts/setup. In case of warnings that the connection is unsafe, add an exception and proceed.

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

__g. Use the **Express Setup** option.

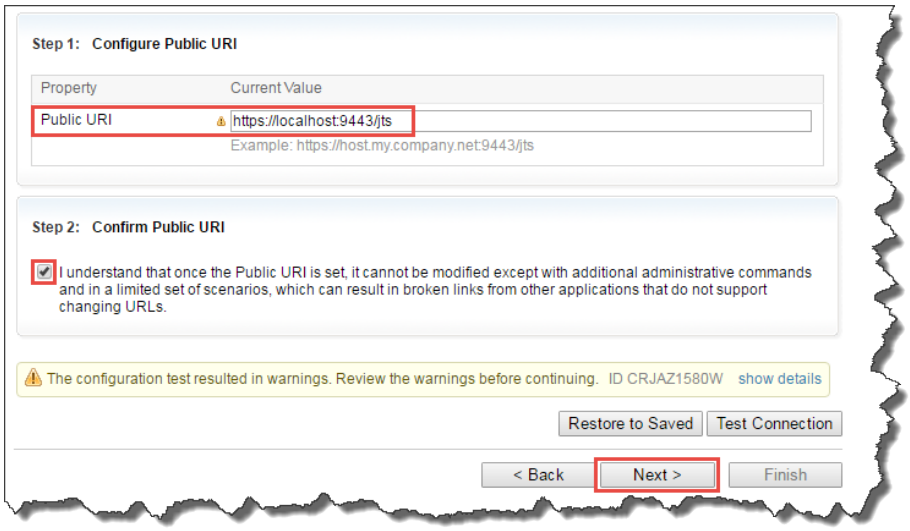


__i. Click **Next**.

__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.

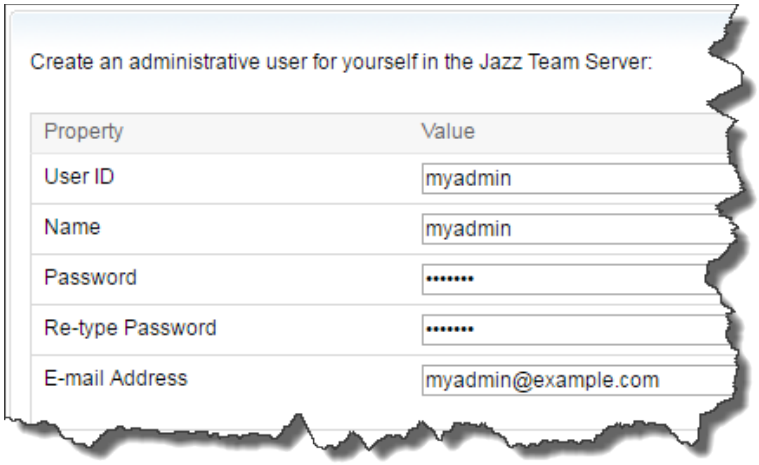
Test the connection, accept the warning in step 2 and click **Next**.



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

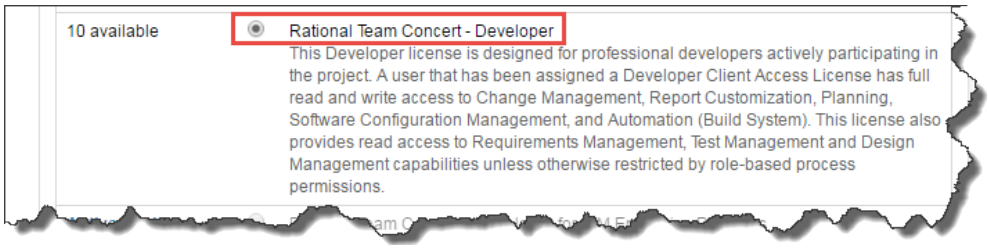
- As User ID, name, password and re-type password enter **myadmin**
- As e-mail enter myadmin@example.com

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 Rational Team Concert Developer license. Then click **Finish**.



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

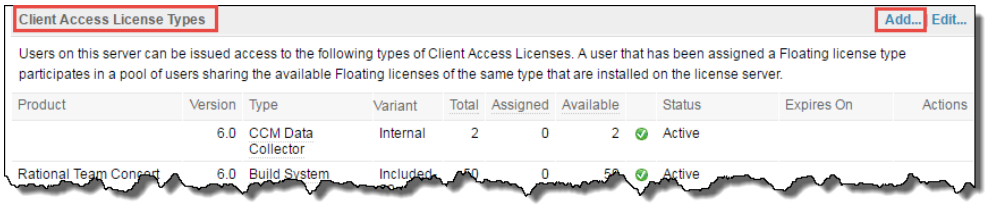
__16. Import the 10 Free Developer CALs. The license assignments in the repository will be preserved (myadmin has a Developer CAL).

__a. Locate the `RTC-Developer-10-C-License-6.0.3.zip` file you downloaded and remember where you placed this file. You will next upload it to your server.

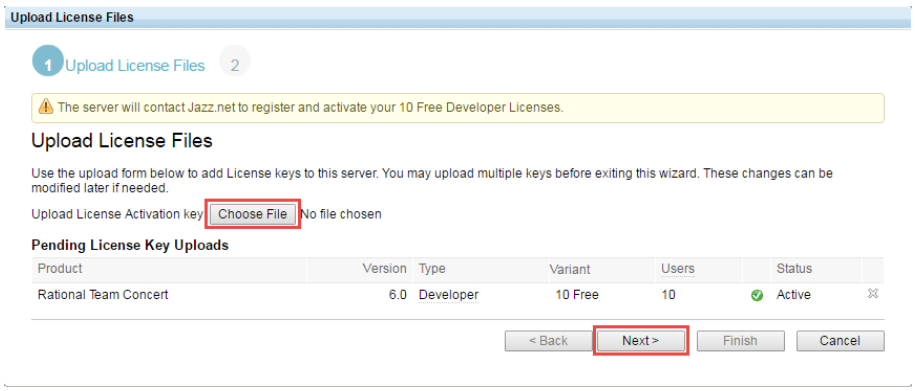
__b. Open or return to your browser and open this URL: https://localhost:9443/jts/admin#action=com.ibm.team.repository.admin.manageLicenses

__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-6.0.3.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.

Rational Team Concert	10 Free	Developer	10	✓	Active

__17. You will now setup the workshop RTC development server repository.

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

__a. Browse to the folder `C:\RTC603Dev\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.

WorkshopSetup.bat

```
1  set JAVA_HOME=..\JazzTeamServer\server\jre
2  set PLAIN_JAVA=../PlainJavaAPI
3  set REPOSITORY="https://localhost:9443/ccm"
4  set USERID="myadmin"
5  set PASSWORD="myadmin"
6
7  %JAVA_HOME%\bin\java -cp %PLAIN_JAVA%/*;./* com
   .ibm.js.rtcext.serversetup.ServerSetup %REPOSITORY% %USERID%
   %PASSWORD% %*
8  pause
```

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 JRE, where to find the Plain Java Client Libraries and the login information for the repository.

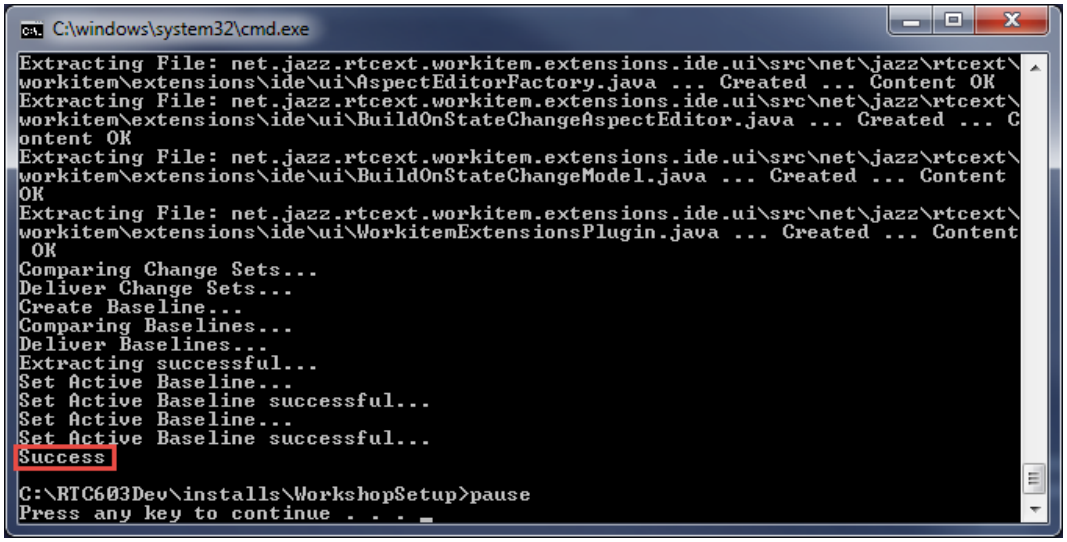
By default the JAVA_HOME file points to the JDK shipped with the RTC server, if at all possible use this setting. If not, make sure JAVA_HOME is set to a **Java JRE** version compatible with the RTC server used here.

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:\RTC603Dev\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 JRE and make sure it is available.

Troubleshooting

Check that you are using a compatible JRE, 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 the Plain Java Client Libraries are installed into the specified folder.

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 folders in `C:\RTC603Dev\installs\WorkshopSetup\data`.

1.3 Setup the Eclipse workspace for Server SDK Development

In this section you will setup your RTC Eclipse workspace for developing RTC Server plug-ins. This consists of specifying what target 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 Plug-in Development Environment (PDE) know about all the RTC platform server API source code.

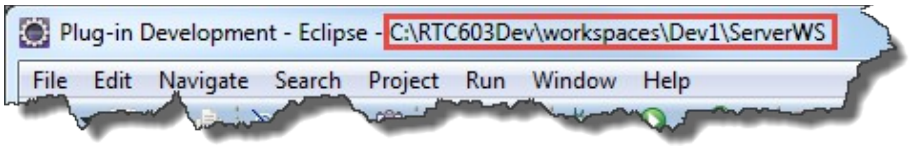
Be patient, some of the operations take a considerable amount of time.

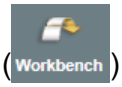
__18. Return to the Eclipse client with the workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS` you have already running and prepare the RTC Server SDK target platform.

__a. If your RTC Server SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

__b. When prompted, select the Eclipse workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS`.

__c. Check the desired Eclipse workspace is in use.

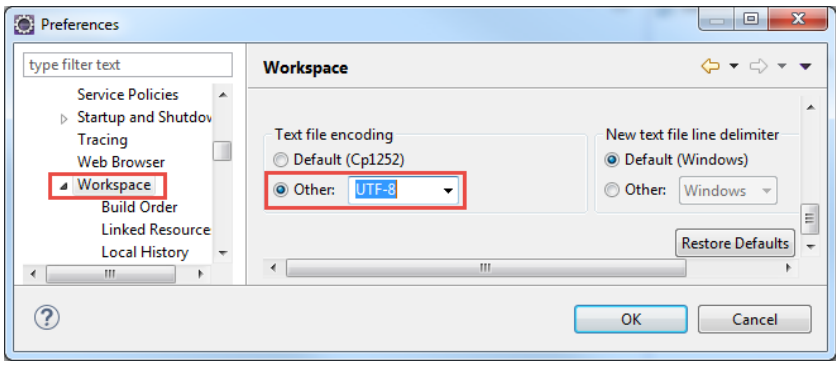


__d. Minimize the **Welcome** screen via this (

) button near the top of the window.

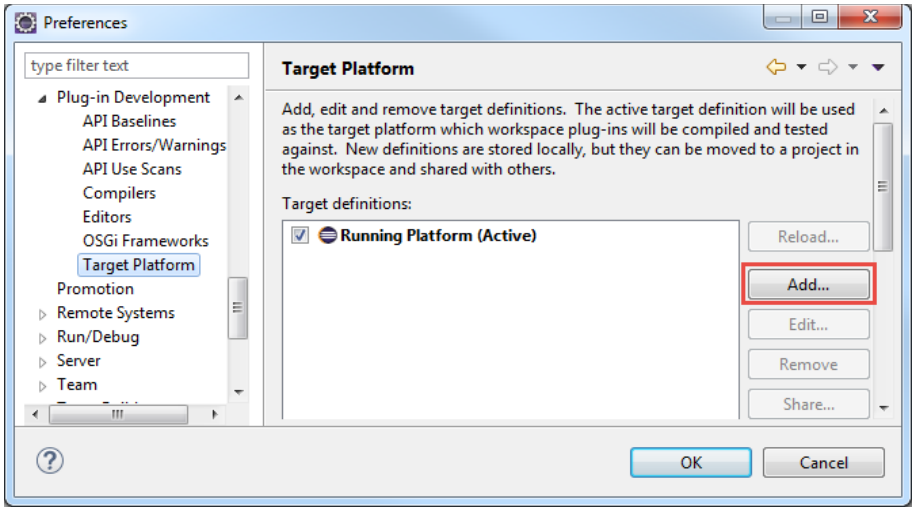
__19. Configure the Eclipse workspace and switch Text file encoding to UTF 8

__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.

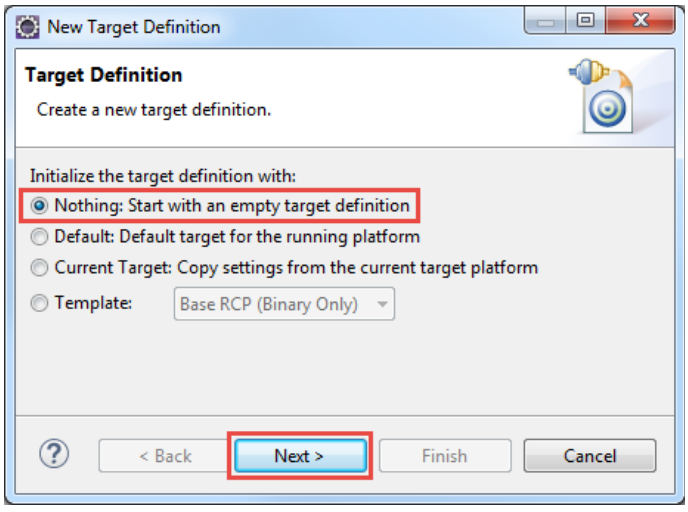


__20. Create a new target platform.

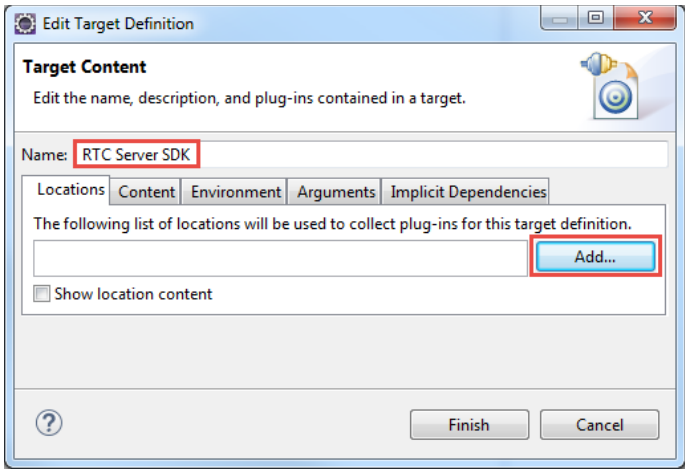
__a. From the menu bar, select **Window > Preferences**. In the **Preferences** dialog, select **Plug-in Development > Target Platform**. Wait for the load process to finish, then click **Add...**



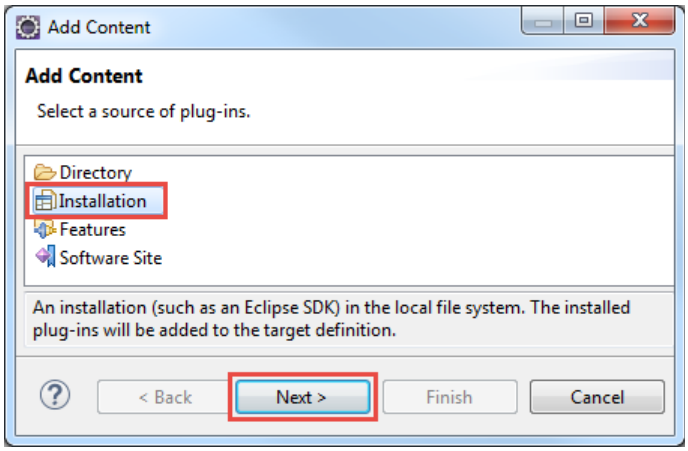
__b. In the **New Target Definition** wizard, select **Nothing: Start with an empty target definition** and then click **Next**.



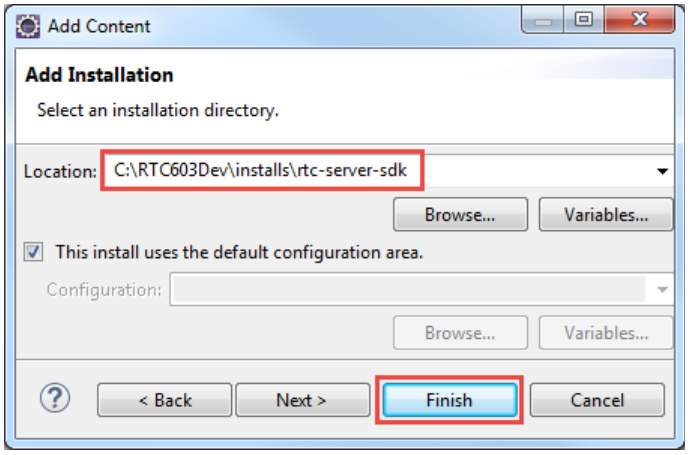
__c. On the second page of the wizard, enter *RTC Server SDK* as the **Name** and click **Add...**



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

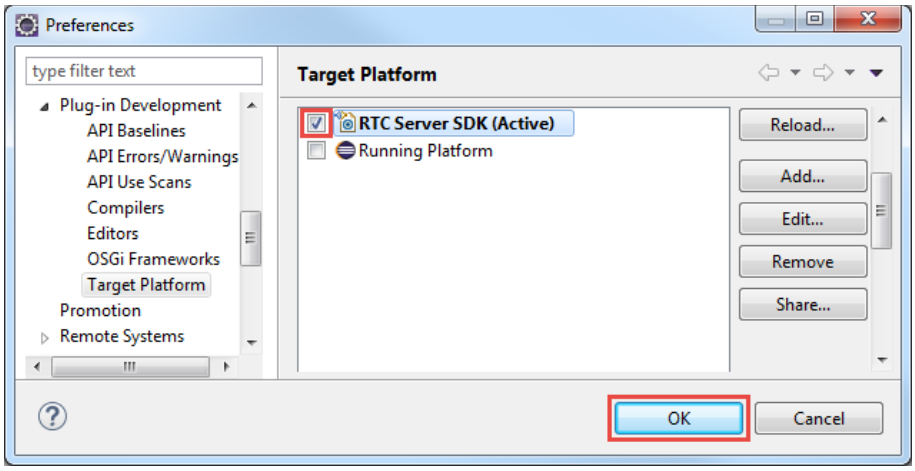


__e. On the second page of the wizard, enter `C:\RTC603Dev\installs\rtc-server-sdk` as the **Location** , or browse to the location and then click **Finish**.



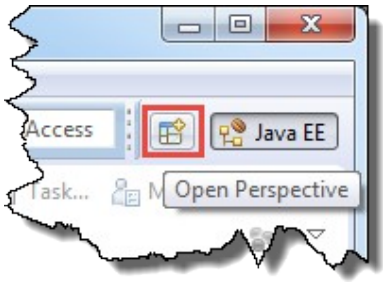
__f. After the operation completes, click **Finish** in the **New Target Definition** wizard.

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

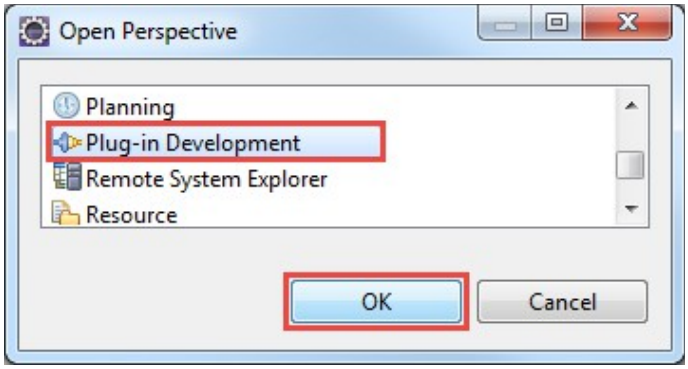


__21. Open the Plug-in Development perspective.

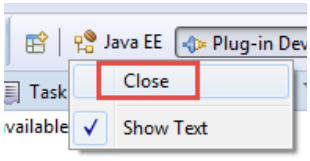
__a. In the toolbar toward the right, click the **Open Perspective** button.



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

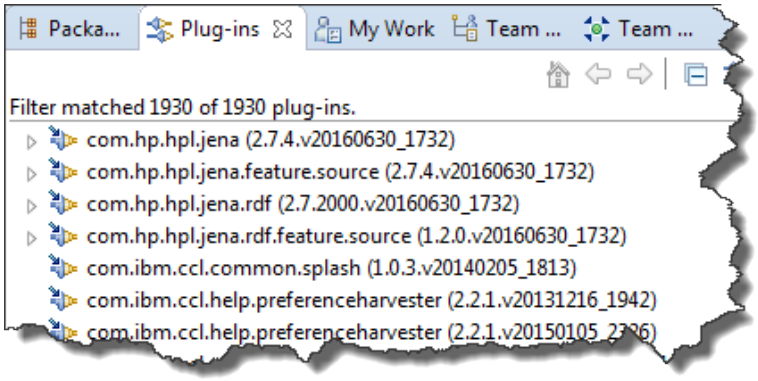


__c. Close the Java EE and any other perspective except **Plug-in Development**.

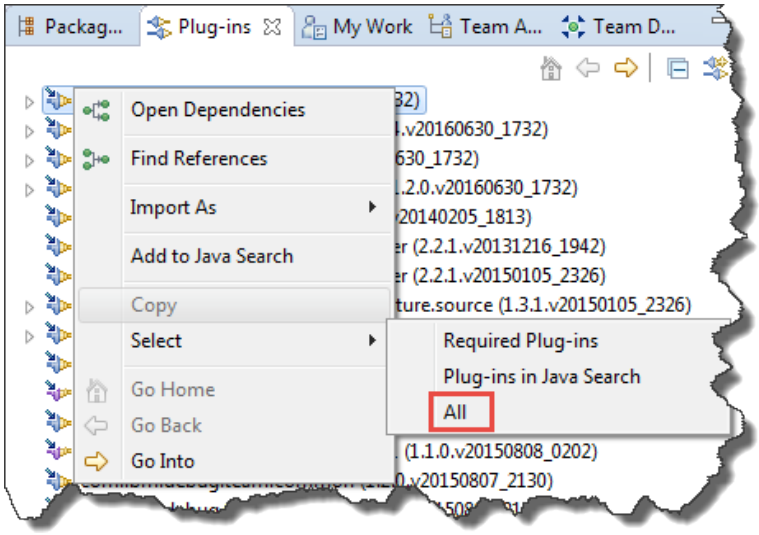


__22. Add RTC source code to Java search.

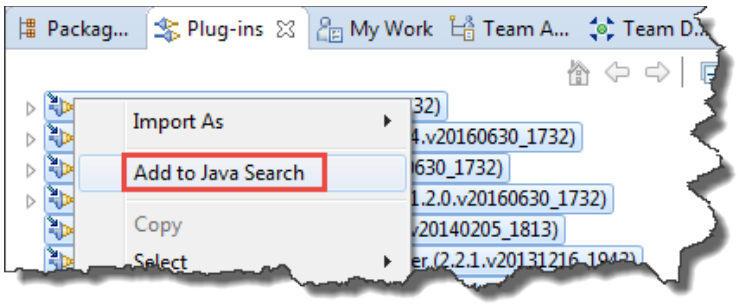
__a. On the left, select the **Plug-ins** view.



__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.



1.4 Finalize setup of the Eclipse workspace for Server SDK Development

In this section you will finalize the setup your RTC Eclipse workspace for developing RTC server plug-ins. You will connect to the RTC development server, load the components needed for server API development and import required files into the workspace.

__23. Start the RTC development server using **`server.startup -debug`**, if it is not already started.

__a. Open a command window. Navigate to `C:\RTC603Dev\installs\JazzTeamServer\server`.

__b. Type **server.startup.bat -debug** and hit enter to run it.

__24. Return to the Eclipse client with the workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS` you have already running and prepare the RTC Server SDK target platform.

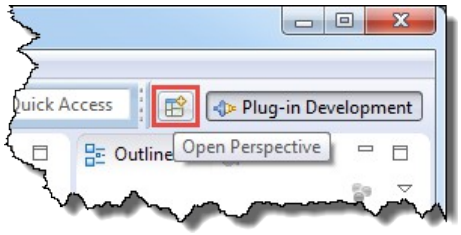
__a. If your RTC Server SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

__i. When prompted, select the Eclipse workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS`.

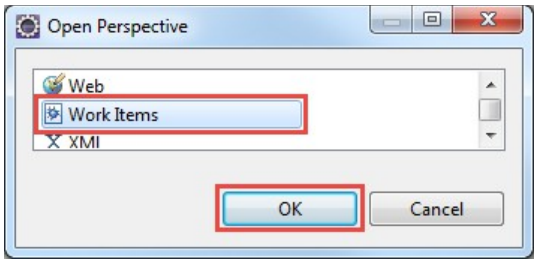
__b. Check the desired Eclipse workspace is used.

__25. Open the Work Items perspective.

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

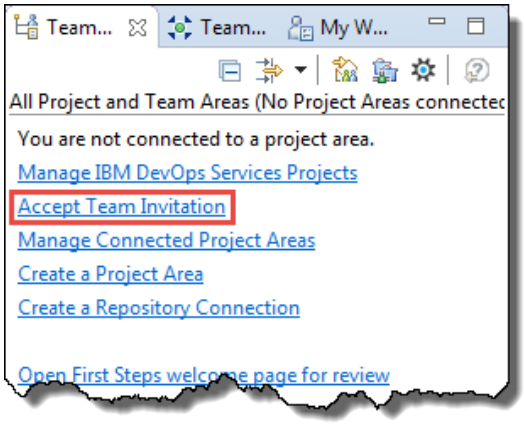


__b. In the **Open Perspective** dialog, select **Work Items** and then click **OK**.



__26. Connect to the project area.

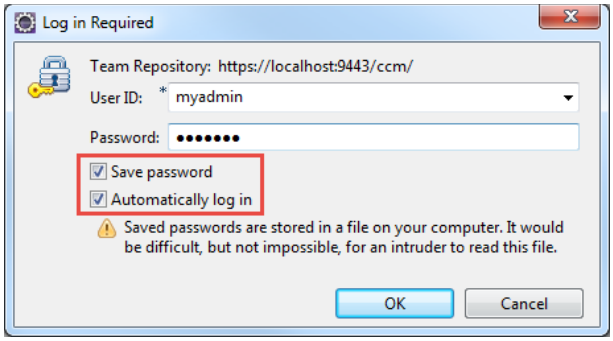
__a. On the left, switch to the **Team Artifacts** view and click the **Accept Team Invitation** link.



__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**.



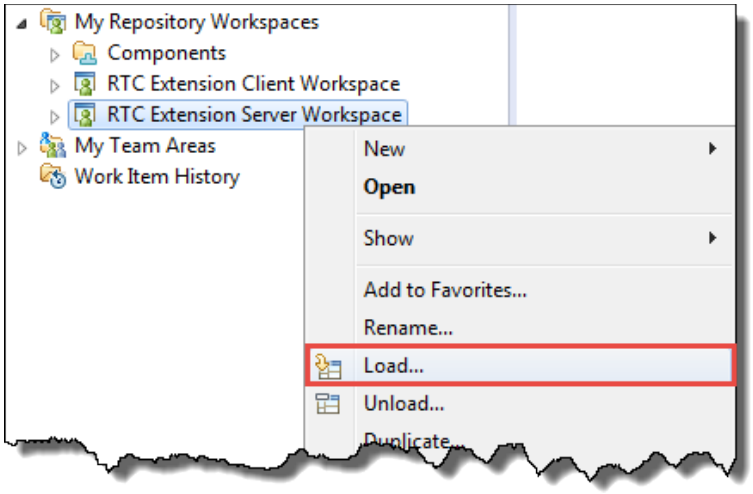
__d. If prompted with a **Repository Connection Certificate Problem**, select the **Accept this certificate permanently** radio button and then click **OK**.

__e. If asked for a password for secure storage, cancel. You will not be able to store the password and have to enter it each time you want to log in.

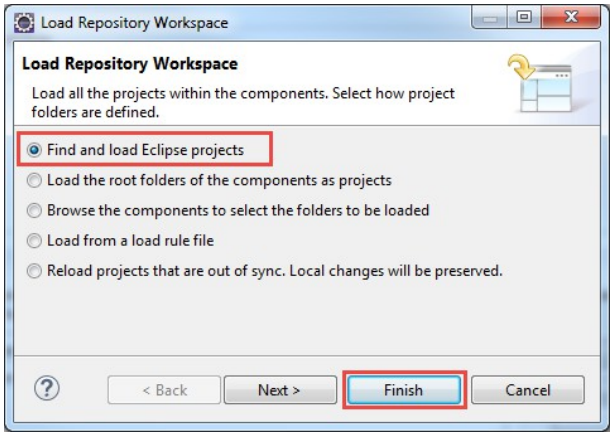
__f. Close the project area editor that opens.

__27. Load the workshop server development repository workspace.

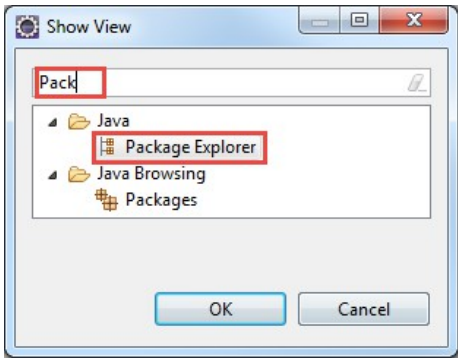
__a. In the **Team Artifacts** view, expand the **My Repository Workspaces** node, right click the **RTC Extension Server Workspace** and then select the **Load…** action from then menu. Note that in later versions of EWM/RTC the node was renamed to 'My Source Control' and sub-nodes where added. The node 'My Repository Workspaces' from the screen shot is now in there and named 'My Workspaces'.



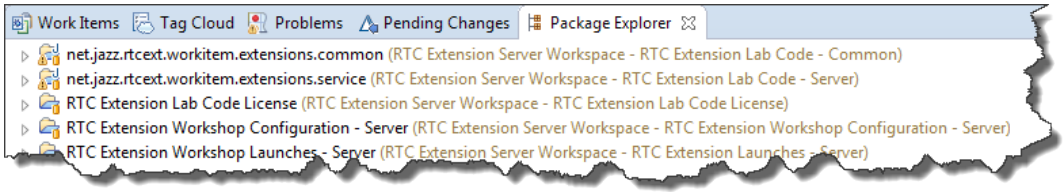
__b. In the **Load Repository Workspace** wizard, make sure **Find and load Eclipse projects** is selected and then click **Finish**.



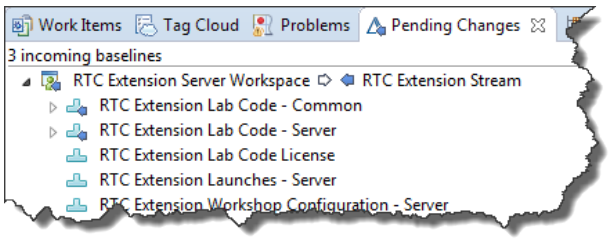
__c. Open a **Package Explorer** view. Open **Window>Show View**, select **Other**. In the filter type **Package**, select the **Package Explorer** and click **OK**.



__d. 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 project (**RTC Extension Lab Code License**) contains the license agreement for the sample code you are using in this workshop. The third (**RTC Extension Workshop Configuration - Server**) contains files that are used to configure the Jetty debug server. The forth (**RTC Extension Workshop Launches – Server**) contains Eclipse launch configurations used for server API development. In the rest of this lab you will learn how to use these launches.

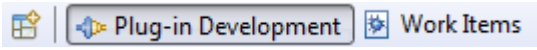


__e. You will also notice in the **Pending Changes** view, that there are incoming change sets and baselines. Do not accept them. You will make use of them in later labs. If the **Pending Changes** view is not open, select **Window > Show View > Other…** from the menu bar, type `pending` into the filter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



__28. Open the Plug-in Development perspective.

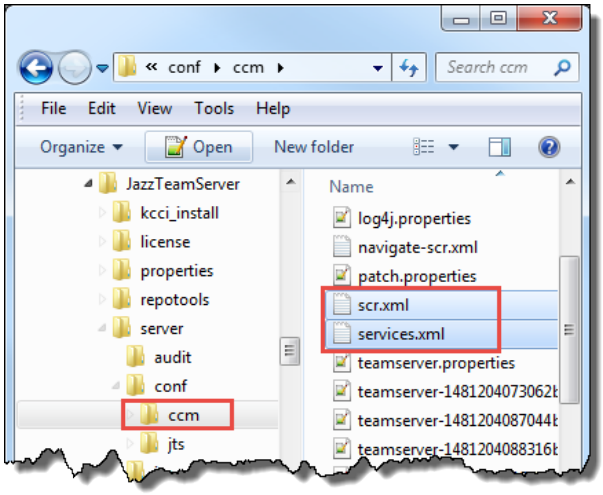
__a. In the toolbar toward the right, click Plug-in Development to switch the perspective.



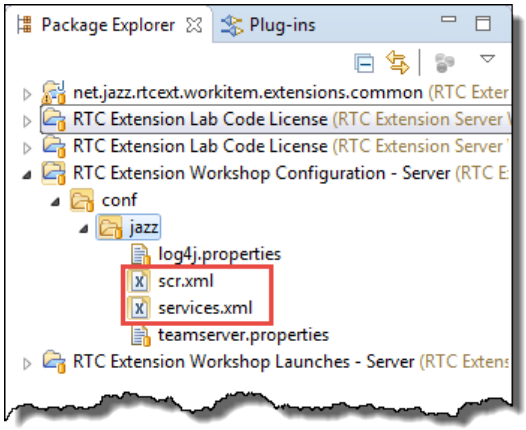
__29. Gather remaining configuration files for Jetty based launches.

__a. The **RTC Extension Workshop Configuration – Server** project also contains some configuration files for use by the Jetty based launches later. 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 servers ccm application configuration. You will find them in the `C:\RTC603Dev\installs\JazzTeamServer\server\conf\ccm` folder.



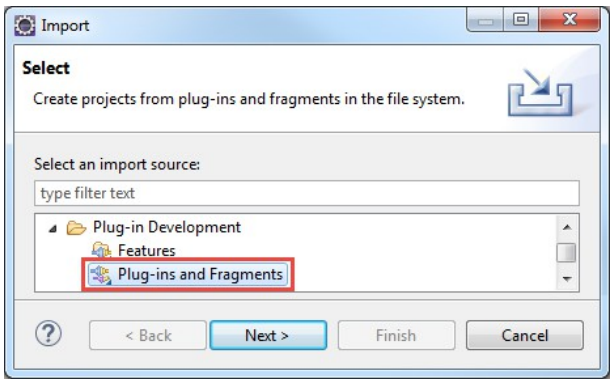
__c. You can copy them from the Windows Explorer and paste them into the **conf\jazz** folder in the **RTC Extension Workshop Configuration** project in the **Package Explorer** view.



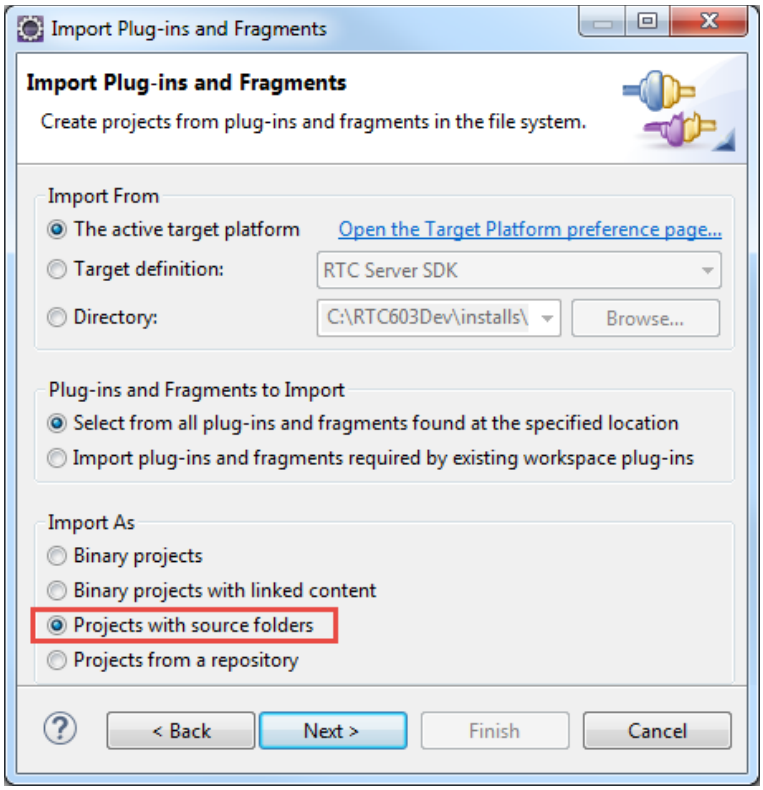
__d. This will give you two unresolved local changes in the **Pending Changes** view. You do not need to do anything with these.

__30. Import plug-ins for Jetty based launches. There are three plug-ins 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.

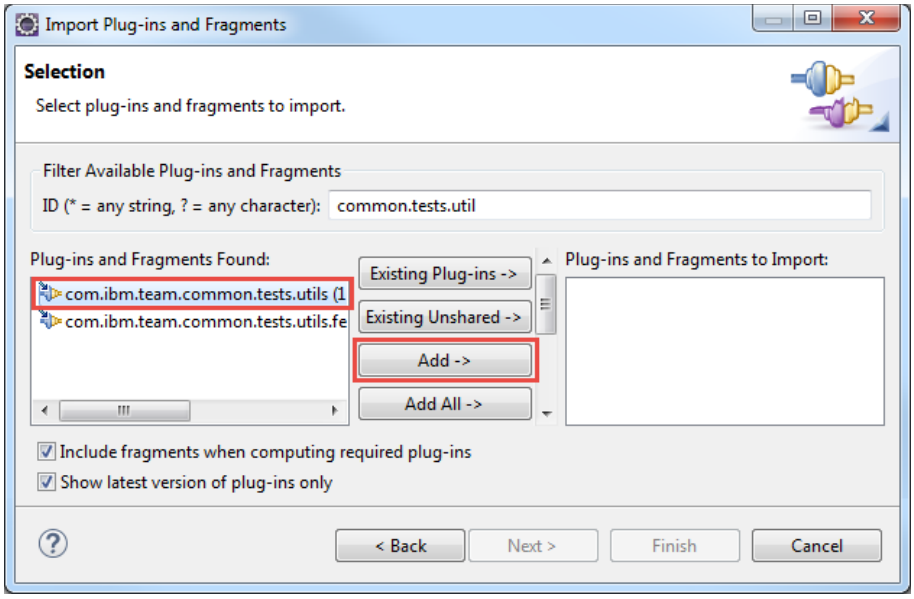
__a. First, import the JUnit test plug-in that contains the database creation code. From the menu bar, select **File > Import…** and then in the **Import** wizard, select **Plug-in Development > Plug-ins and Fragments** as shown here and then click **Next**.



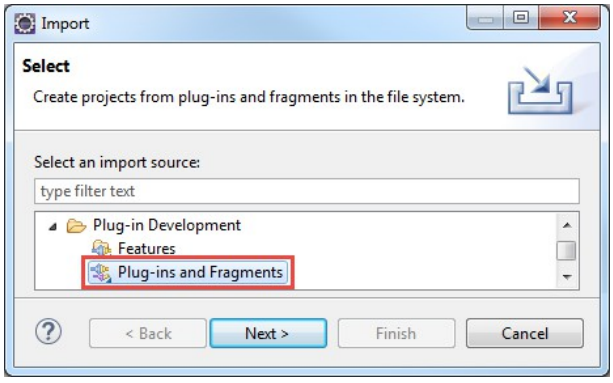
__i. On the second page of the wizard, make sure your selections match those shown here. The only one you should have to change is highlighted. Then, click **Next**.



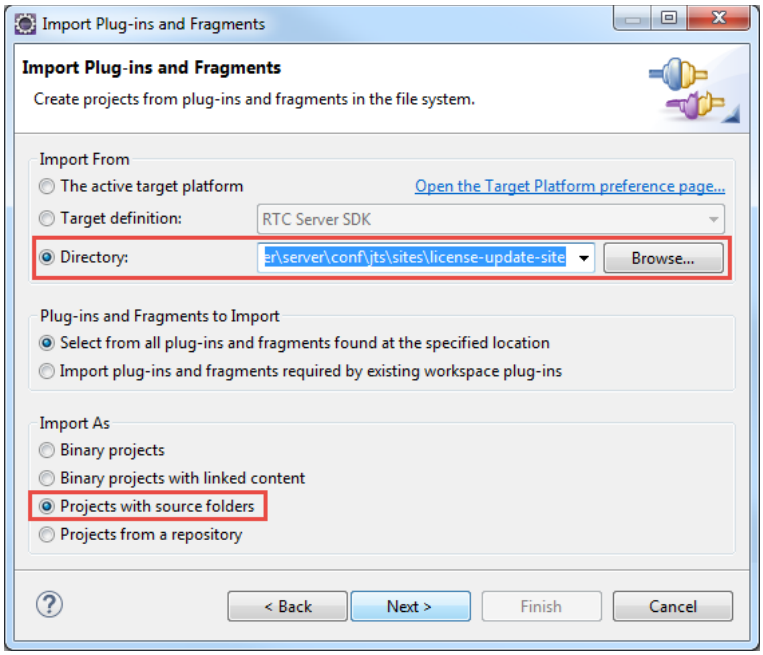
__ii. On the third page of the wizard, enter `common.tests.utils` into the **ID** field. This will filter the plug-ins list. Select the **com.ibm.team.common.tests.utils** plug-in in the list, click **Add -->** and then click **Finish**.



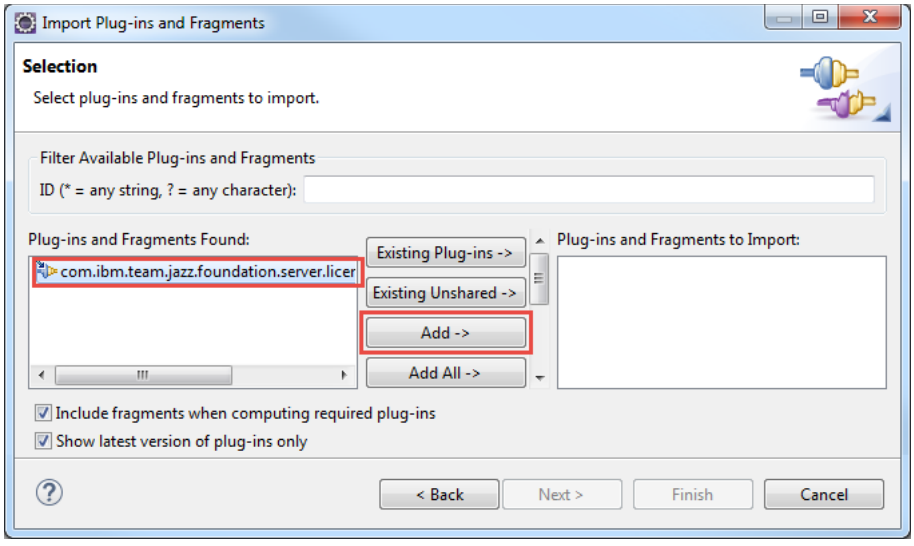
__b. Next, import the server license from the server installation. This will override the development time server license you would otherwise be using in a Jetty launch with the permanent server license. It is likely that the development license has expired. As before, from the menu bar, select **File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



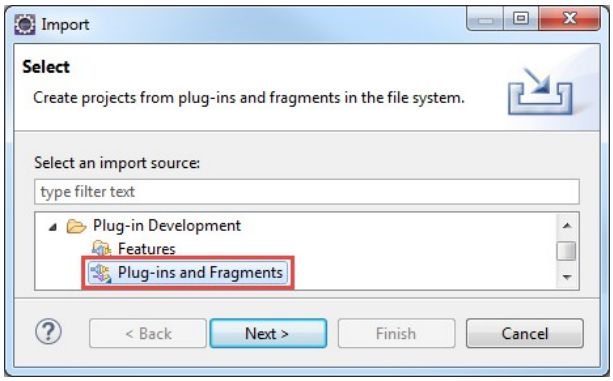
__i. This time on the second page of the wizard, make sure your selections match those shown here. The major difference from last time is the selection of a different place to import from. The **Directory** field should be set to (use the **Browse…** button to find it): `C:\RTC603Dev\installs\JazzTeamServer\server\conf\jts\sites\license-update-site` and click **OK**. Then click **Next.**



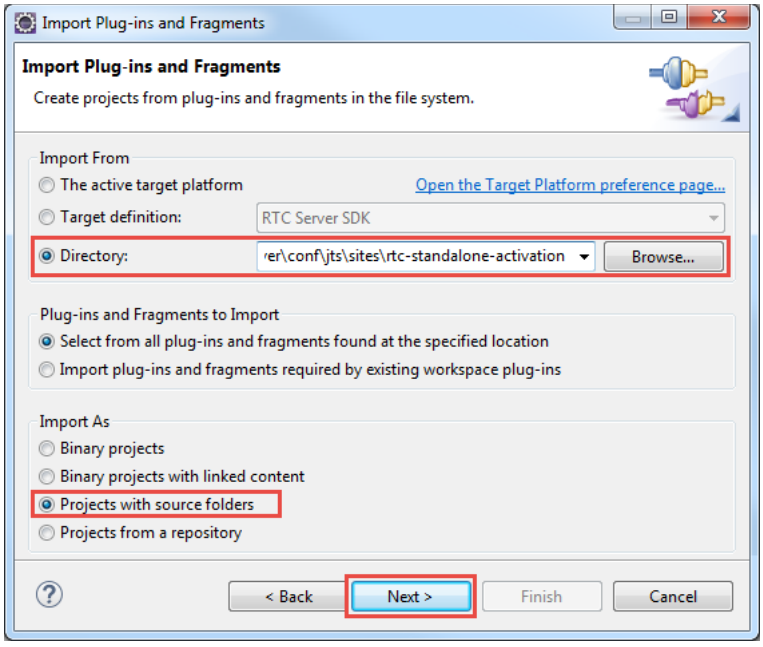
__ii. On the third page of the wizard, select the **com.ibm.team.jazz.foundation.server.licenses.enterprise-ea** plug-in in the list. Then click **Add -->** and finally click **Finish**.



__c. Finally, import the client access licenses (CALs) from the rtc development 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**.

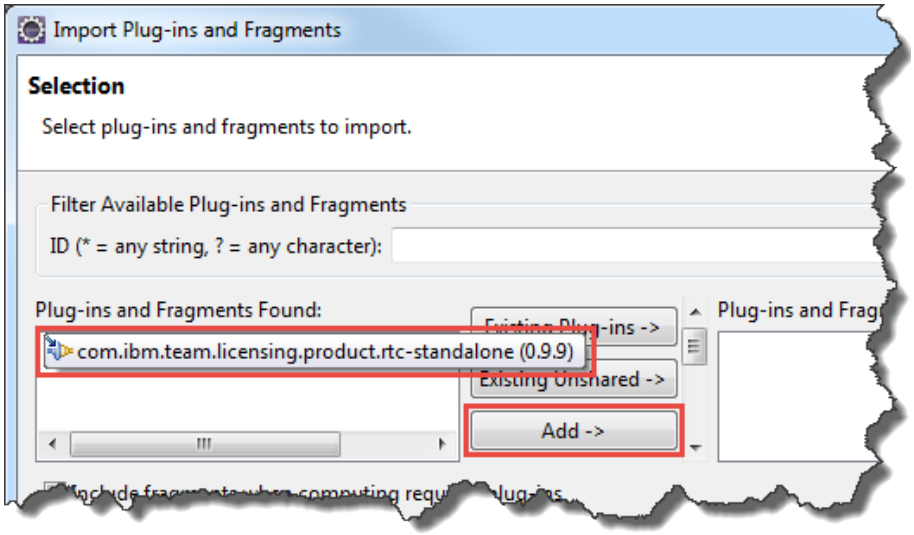


__i. This time on the second page of the wizard, make sure your selections match those shown here. The major difference from last time is the selection of a different place to import from. The **Directory** field should be set to (use the **Browse…** button to find it): `C:\RTC603Dev\installs\JazzTeamServer\server\conf\jts\sites\rtc-standalone-activation` and click **OK**. Then click **Next**.



__ii. On the third page of the wizard, select the `com.ibm.team.licensing.product.rtc-standalone` plug-in in the list.

Then click **Add -->** and finally click **Finish**.



__a. In case you installed using the Web Installer or the Installation Manager version choose `com.ibm.team.licensing.product.clm`.

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 using Web Installer or Installation Manager

If you installed RTC using the Web Installer or the Installation Manager, the folder `rtc-standalone-activation` will not exist. In this case look for the folder: `C:\RTC603Dev\installs\JazzTeamServer\server\conf\jts\sites\clmactivation` and import the plug-in `com.ibm.team.licensing.product.clm`.

You can not find any of these folders

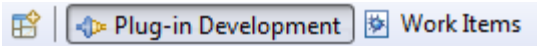
You probably forgot to install the "**Rational Team Concert Required Base License Keys, Including Trials**" installation package when using a Web Install or an Installation Manager Repository install.

You will need to return to the RTC development 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.

1.5 Test connecting the Eclipse debugger to the RTC development server

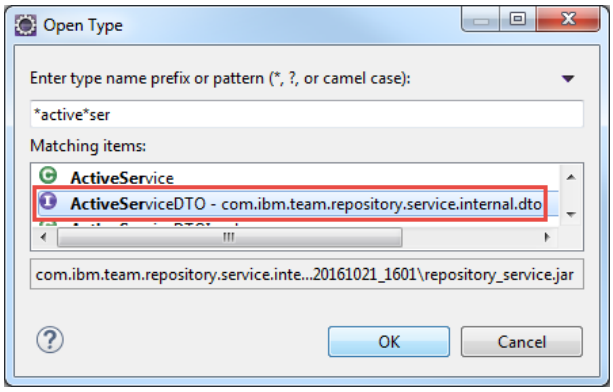
__31. Set a breakpoint to be used to verify the debugging connection.

__a. Switch to the Plug-in Development perspective. In the toolbar toward the right, click Plug-in Development to switch the perspective.

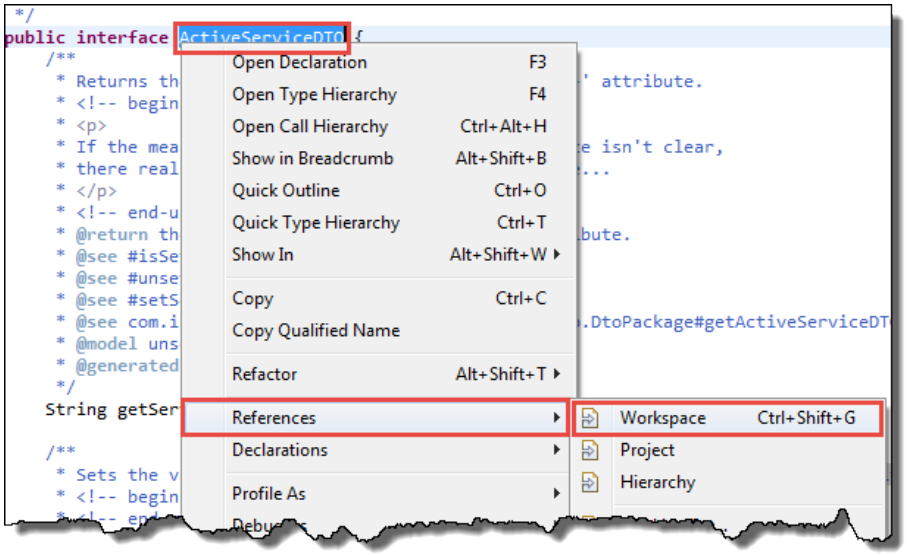


__b. Select **Navigate > Open Type…** from the menu bar.

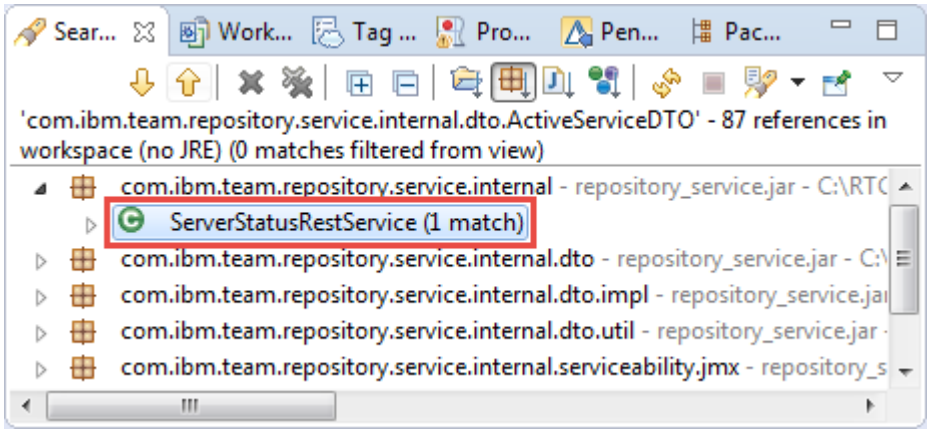
__c. In the **Open Type** dialog type `*active*ser` in the pattern field. Several types will appear (depending on the version). Select the **ActiveServiceDTO** interface as shown here and then click **OK**.



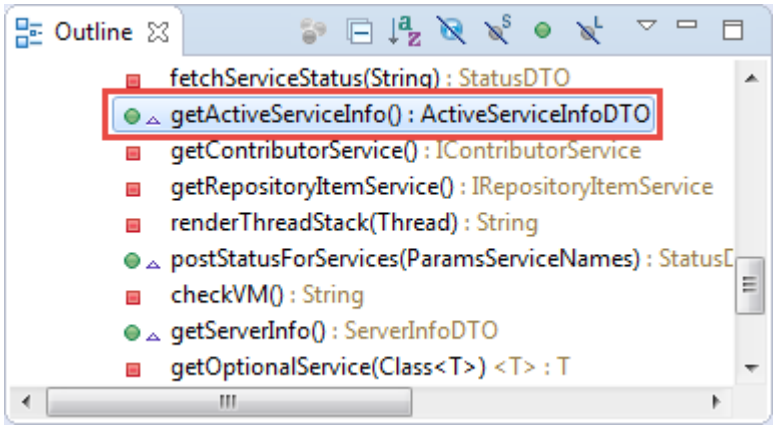
__d. When the Java editor opens on the class, the class name will be highlighted. Right click the class name and select **References > Workspace**.



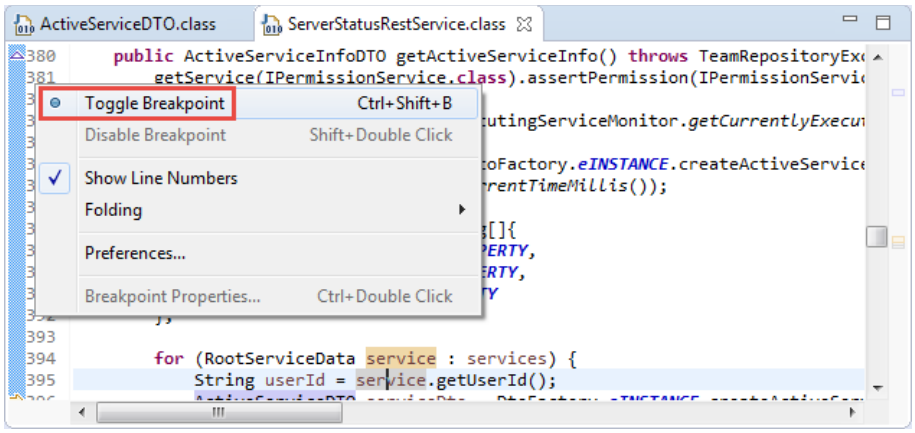
__e. The first entry in the **Search** results view is the one you want. Double click the **ServerStatusRestService** class to open an editor on it.



__f. Open the **Outline** view. It shows the structure of the ServerStatusRestService class. In the **Outline** view, click the **getActiveServiceInfo()** method.



__g. 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

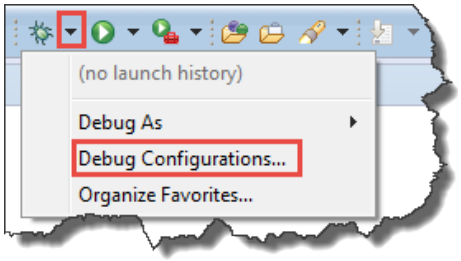
The screen shot above is from the 6.0.3 SDK. In other versions the code might look different from the screen shot below. Set the breakpoint at the first line with code in the method.

Searching the API

Searching for interfaces, classes and methods and searching for references and usage in the SDK is a powerful tool helping to explore and understand the API.

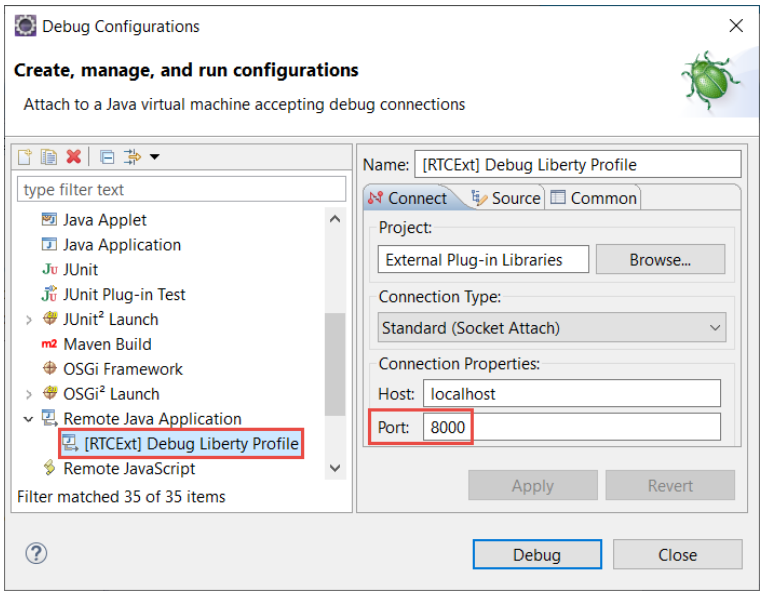
__32. Attach the Eclipse debugger to the RTC development server.

__a. From the **Debug** toolbar icon drop-down select **Debug Configurations…**



__b. In the **Debug Configurations** dialog, expand the **Remote Java Application** launch type, select the **[RTCExt] Debug Running Liberty Profile** launch configuration.

__i. If you use a different debug port, adjust the **Port** value here before debugging.



__ii. 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.

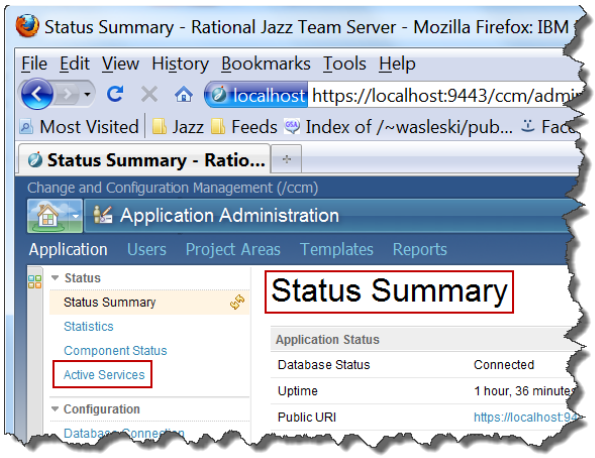
__iii. Click Debug to start the [RTCExt] Debug Running Liberty Profile launch. Note how the Eclipse debugger connects to the RTC Development server.

__33. Use the RTC Web UI to trigger the breakpoint.

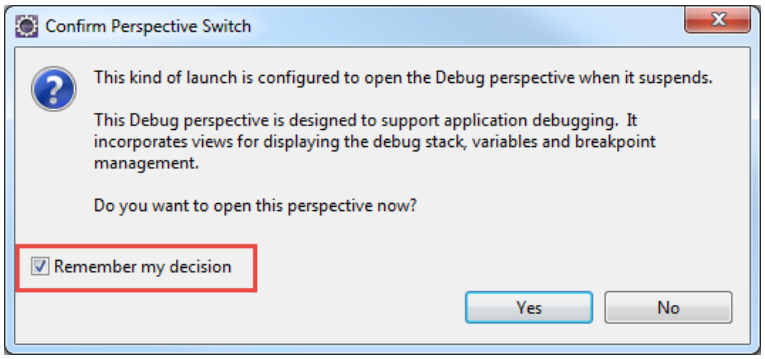
__a. Open your browser (if it is not already open) and enter the URL https://localhost:9443/ccm/admin.

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

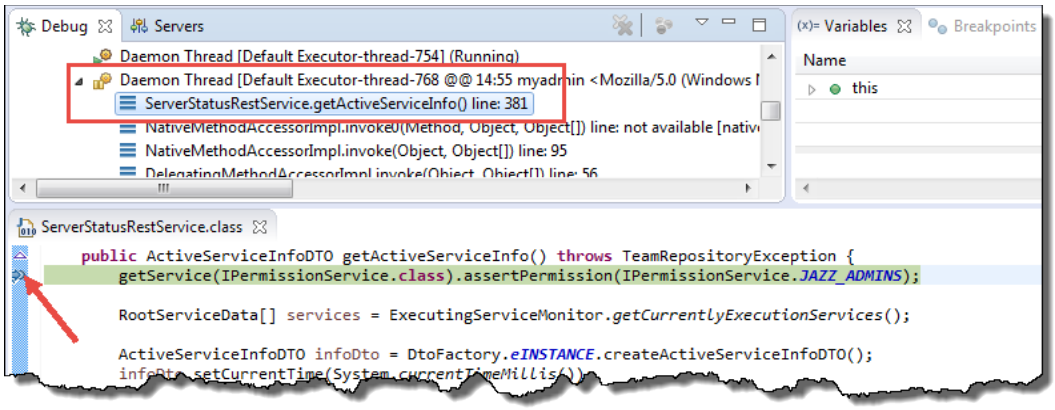
__c. When the **Status Summary** page appears, click the **Active Services** link on the left.



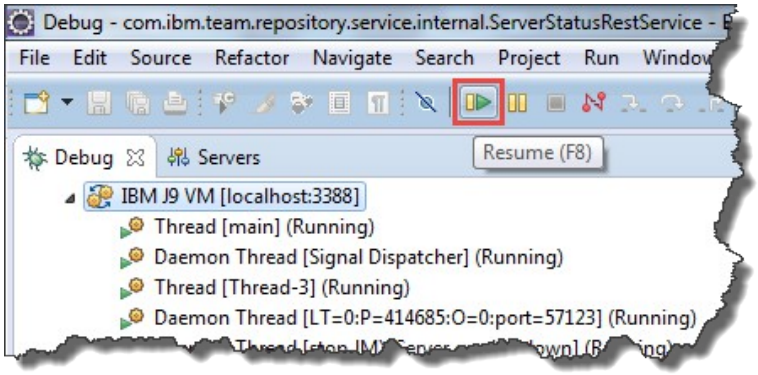
__d. The breakpoint will trigger and the RTC Eclipse client should come to the foreground (or flash in the Windows taskbar if minimized). If you are prompted to switch to the Debug perspective, click the **Remember my decision** checkbox if you wish, and then click **Yes**.



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

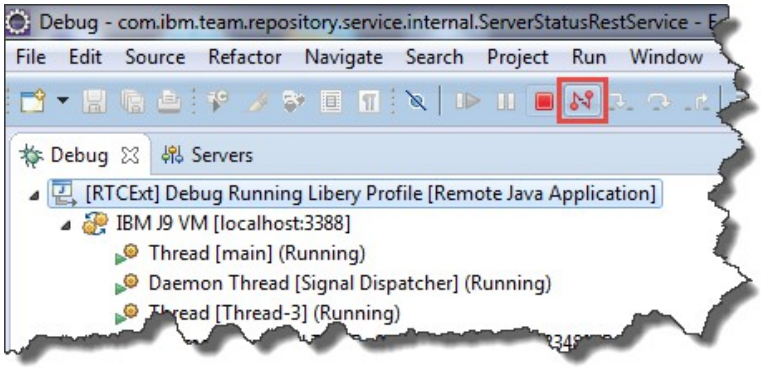


__f. Click the **Resume** toolbar button to resume execution of the RTC development server.



__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 Liberty Profile based RTC development server by clicking the **Disconnect** toolbar button.



1.6 Test the Jetty based debug server launch

As mentioned earlier, you will now launch a Jetty based RTC debug server from Eclipse. The Jetty based RTC debug server will use a separate repository database. It will also use separate ports. This will give you a development debugging and test environment that is separate from your Development RTC Server environment.

Debugging and testing with Jetty has a couple advantages:

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

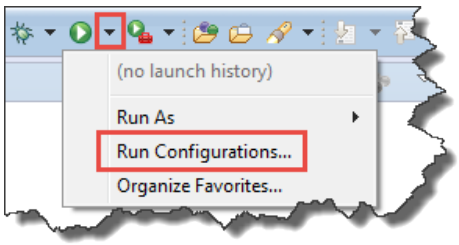
The primary disadvantage is that this launch runs the server as one application at the "\jazz" context root and not as separate JTS and CCM applications. This is generally fine for development and you do have the Liberty Profile RTC development server with split applications for final testing.

__34. 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.

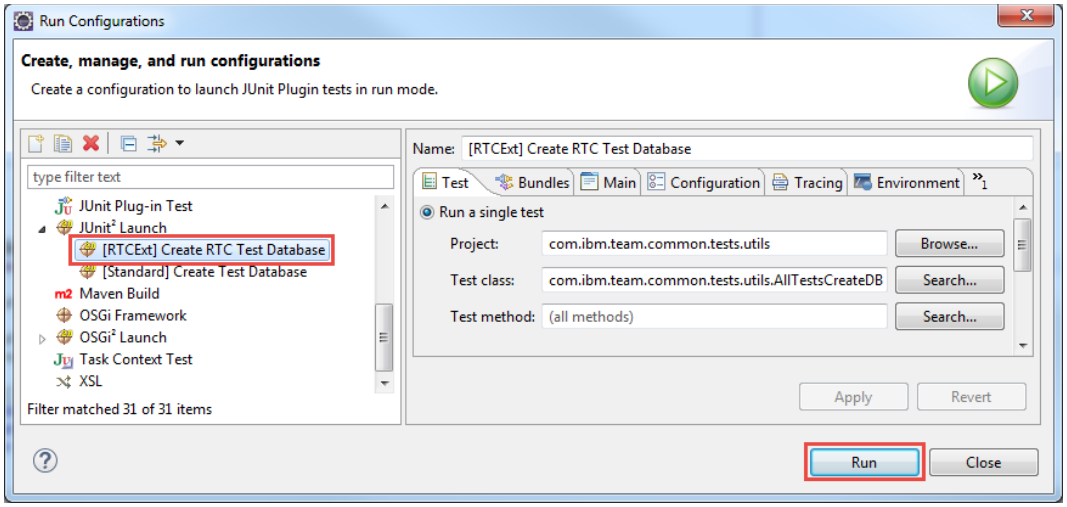
__a. The database creation test you are about to run uses a Jetty server during initialization of the database. Unfortunately, that server must run at the same ports as the Liberty Profile based RTC development server you currently have running. You will need to temporarily stop the Liberty Profile based RTC development server. You will be able to restart it after the database is created. This will not be a problem when running your Jetty debug and test server later. It and the RTC development server will use different ports.

In the Windows Explorer navigate to `C:\RTC603Dev\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 drop-down menu off the **Run** toolbar icon.



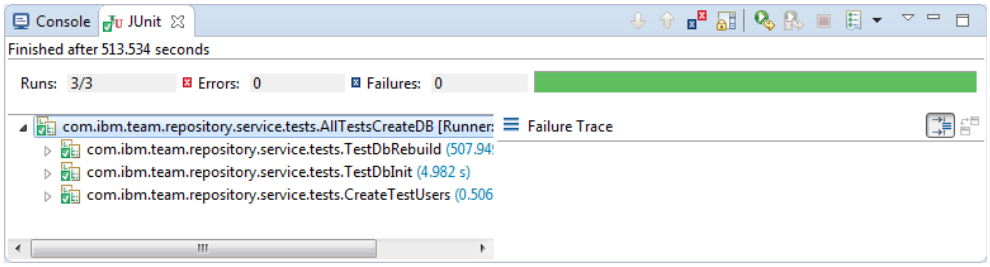
__c. On the **Run Configurations** dialog, select **JUnit² Launch > [RTCExt] Create RTC Test Database** and then click **Run**. Note that if you switch to the **Bundles** tab, you will see that three of the bundles you imported earlier (the test bundle that creates the database and the licenses) are 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, errors included, and a red terminate button. The **JUnit** view will also be active but usually be hidden by the console window. Be patient, this takes a good while and unless the launch terminates itself, let it keep running.

When the red terminate button in the **Console** view goes gray, the launch is finished.

Check the **JUnit** test status. If it shows 3 of 3 runs, no errors, no failures and green check marks like below, it ran successfully. Ignore the errors the **Console** view might show.



__e. The important point is that the **JUnit** view shows success. If it fails, make sure you have shut down the Liberty Profile RTC development server and retry until it succeeds. See the tips below for what you can do to try to make it run successfully. It is impossible to proceed with the workshop if this step fails.

JUnit keeps failing

If the JUnit test keeps failing:

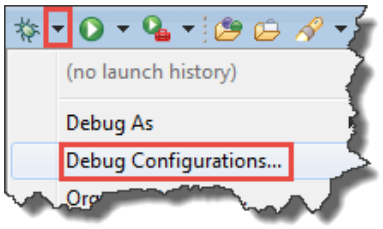
- make sure that the Liberty development server is shut down and the port 9443 is available. Consider restarting the computer.
- Check the console and the log files if the issue could be related to out of memory and adjust the memory settings in the launch.
- Navigate to `C:\RTC603Dev\workspaces\Dev1` and delete the folders `server` and `junit-team-server-tests-workspace`.

Run the JUnit test again.

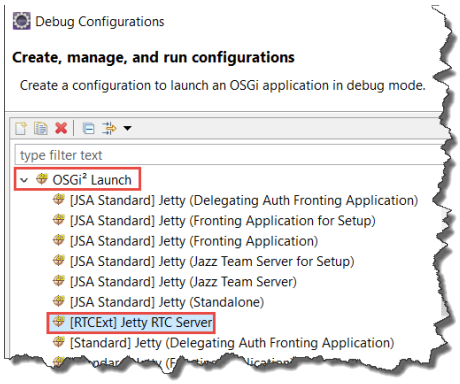
__f. You can now restart your Liberty Profile RTC development server. Open a Windows Explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file. Note, it is now not started in debug mode.

__35. Launch the Jetty based server for debugging.

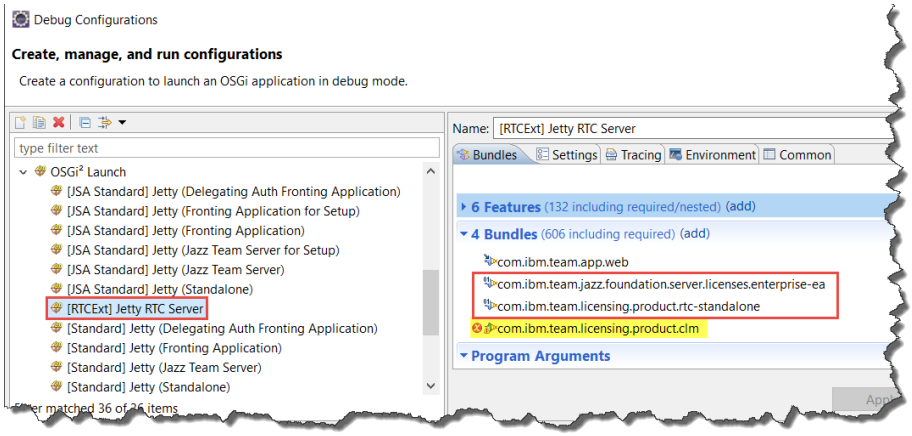
__a. Select **Debug Configurations…** from the drop-down menu of the **Debug** toolbar icon.



__b. In the **Debug Configurations** dialog, select **OSGi² Launch > [RTCExt] Jetty RTC Server**.



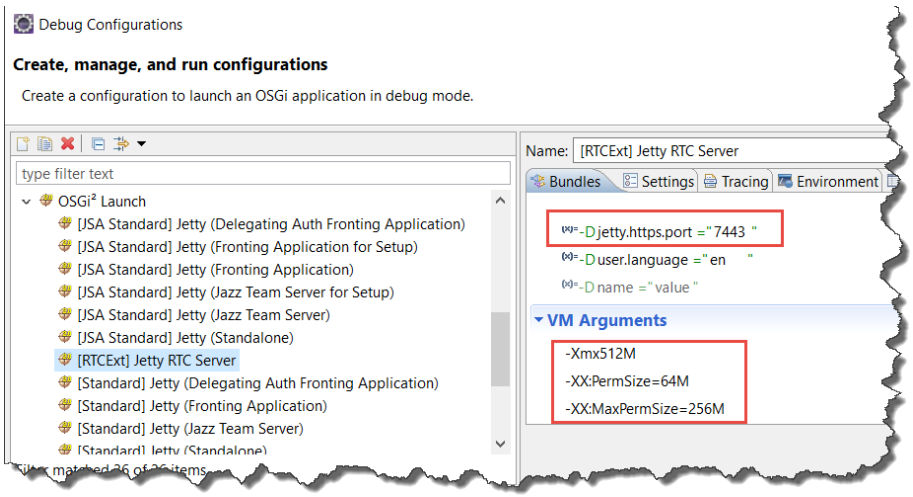
__i. Inspect the **Bundles** tab, and note that two of the bundles you imported earlier (the license bundles `com.ibm.team.jazz.foundation.server.licenses.enterprise-ea` and `com.ibm.team.licensing.product.rtc-standalone`) are included in this launch.



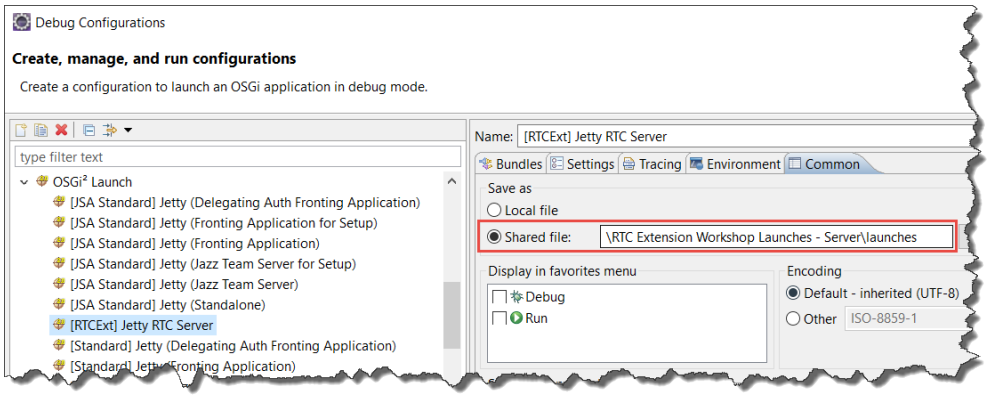
__ii. Note that the bundle for `com.ibm.team.licensing.product.clm` shows an error. This is not a problem as `com.ibm.team.licensing.product.rtc-standalone` is available. Dependent on the install only one of these license files is available. You will learn about adding your own bundles to launches in subsequent labs.

__iii. You may also notices some other launches that start with [Standard]. These come from the test plug-in you imported from the RTC SDK. You will not use them in this workshop.

__iv. 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 as used by the RTC development server. VM Arguments are provided as well.

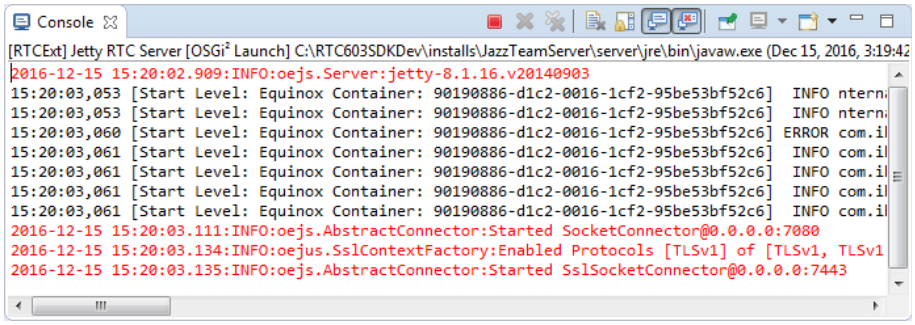


__v. If you switch to the **Common** tab, you will notice the setting shared file which allows to store the launch file in a project area instead of the workspace specific default location in `.metadata\.plugins\org.eclipse.debug.core\.launches`. This makes it possible to keep the launch under version control as well.



__c. Click **Debug** to run the [RTCExt] Jetty RTC Server launch file to debug the RTC server on Jetty.

__d. 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.



__36. Connect with your browser.

__a. Start your browser and navigate to this URL: https://localhost:7443/jazz/admin (note that the port is different). You may need to add another security exception.

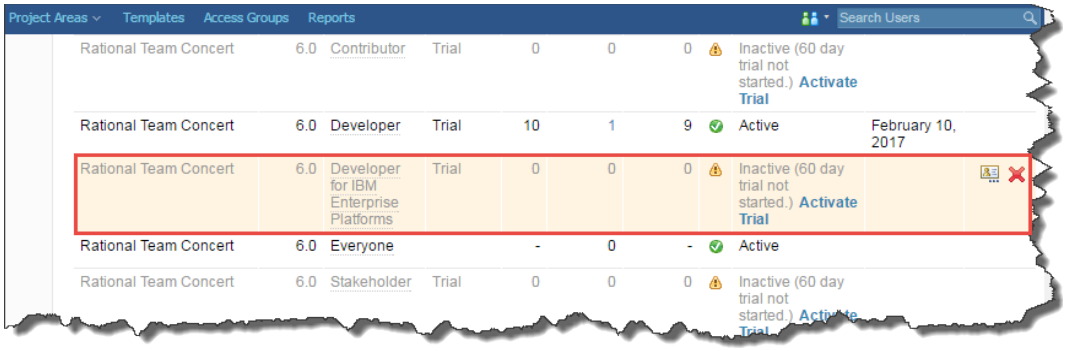
__b. Log in with **`TestJazzAdmin1`** as both the **User ID** and **Password**. For this workshop we will use the `myadmin` user for the Liberty Profile RTC development server and this other administrator id for the Jetty launched debug server. This will hopefully make things a little less confusing as 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

__37. Open the license key management on the server. Check if the licenses are activated. In older product versions you need to activate the trial licenses to be able to use them. In newer versions the developer trial licenses are already activated and you can continue after verifying the fact. You will now perform this step in the Jazz Team Server Administration pages.

__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 Developer”. You could activate the trial license nut they will expire. So we use the 10 Free Developer licenses instead



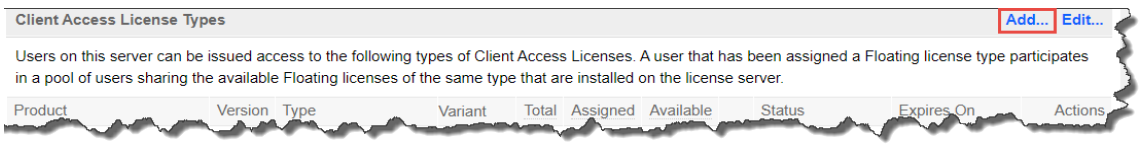
__38. Upload and use the 10 Free licenses

__a. The trial licenses for the debug server will expire, this can be avoided using free or other licenses. The suggested approach is to upload and enable the 10 Free Developer licenses. This makes the setup for this workspace permanently usable.

Expiring Licenses

You can use and upload the 10 Free licenses to keep your debugging repository usable for an extended period.

__b. To upload and enable the 10 Free Developer licenses, locate the section Client Access License Type on the same screen, and click Add...

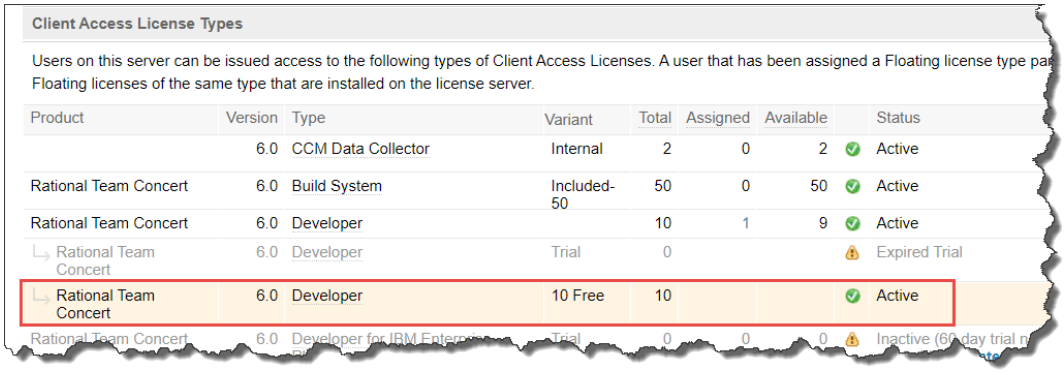


__i. On the Upload License File page click Add File.

__ii. Navigate to the folder download folder e.g. `C:\RTC603Dev\downloads\` to and select the 10 Free Developer license file e.g. `RTC-Developer-10-C-License-6.0.3.zip`.

__iii. Click Next, then accept the license agreement and click **Finish**.

__iv. You should now see the 10 free Developer license. You can delete the Trial License.

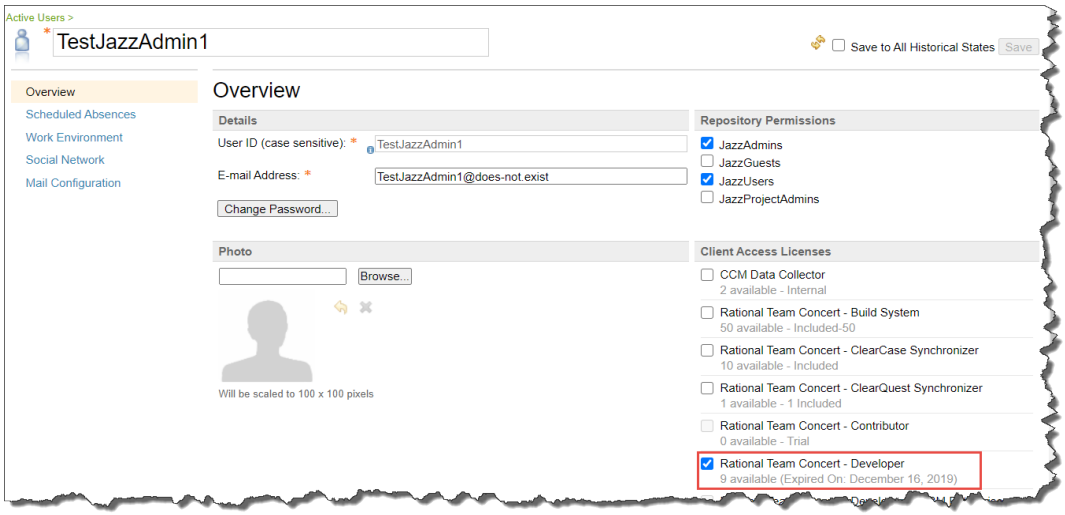


__39. Assign Developer licenses to the test users for the workshop.

__a. The workshop will use the user TestJazzAdmin1. To be able to perform operations on the EWM project area, this user needs a license assigned.

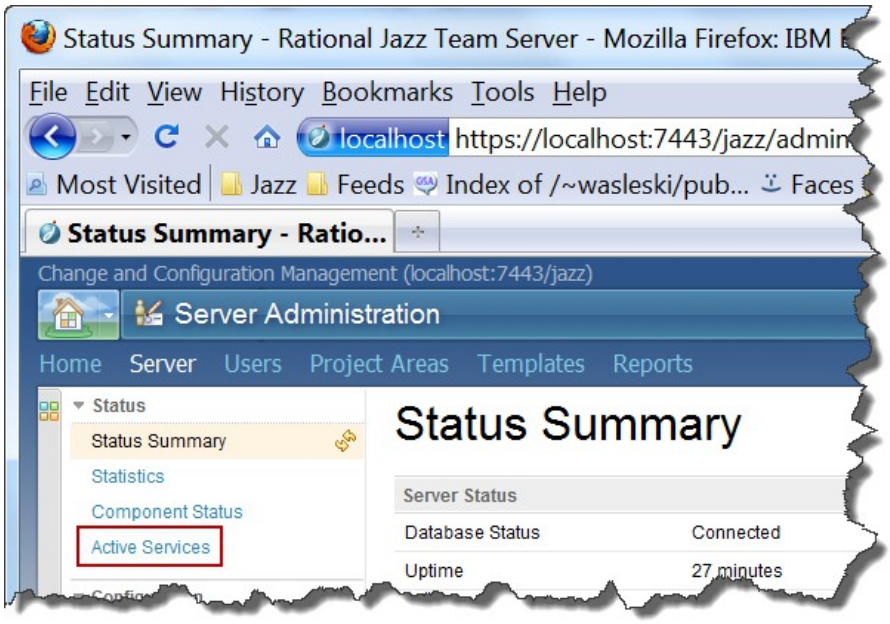
__b. Browse to the User management page https://localhost:7443/jazz/admin#action=com.ibm.team.repository.manageUsers

__c. Click at the user TestJazzAdmin1 and make sure the user has a 10 Free Developer license assigned.

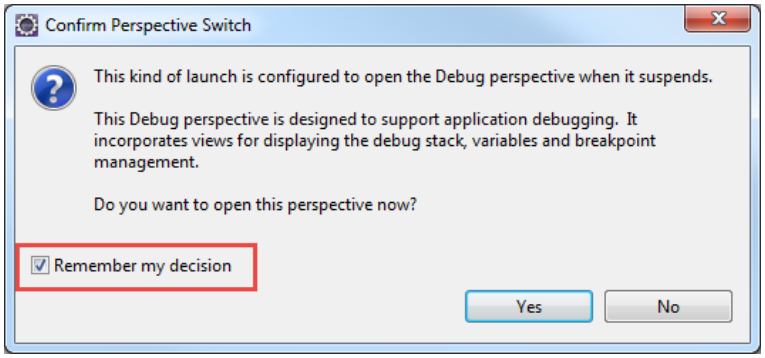


__40. Trigger the breakpoint set earlier.

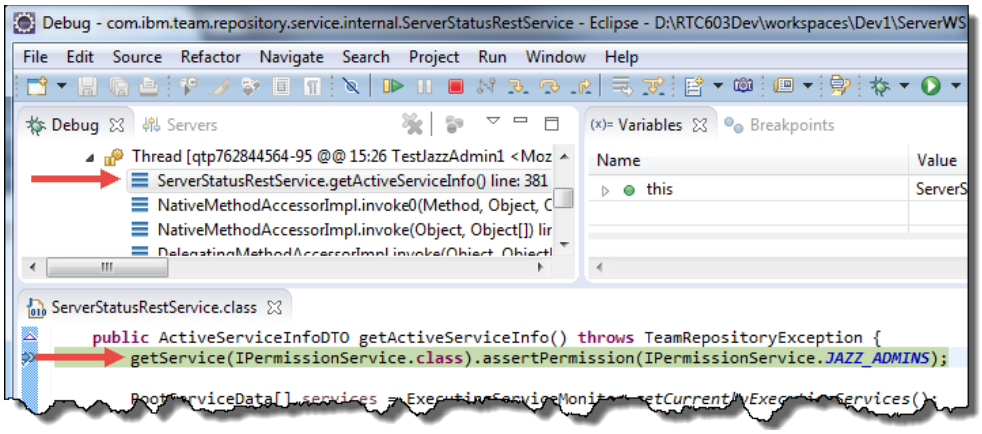
__a. Click scroll up. As before, click the **Active Services** link on the left.



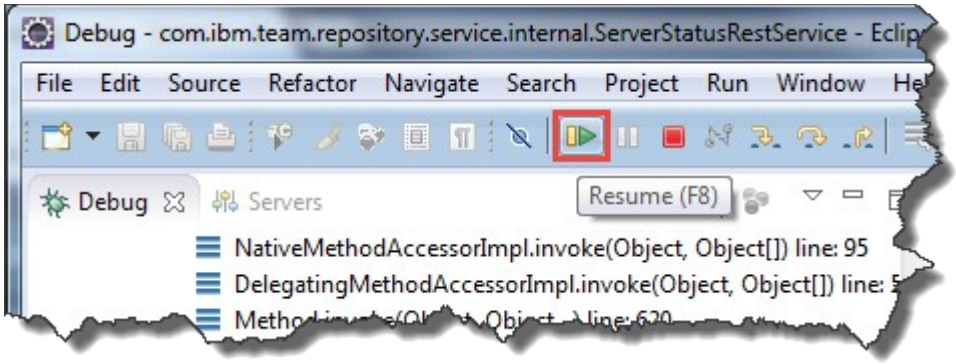
__b. The breakpoint will trigger and the RTC Eclipse client should come to the foreground or flash in the Windows taskbar. If you are prompted to switch to the **Debug** perspective, click the **Remember my decision** checkbox if you wish, and then click **Yes**.



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



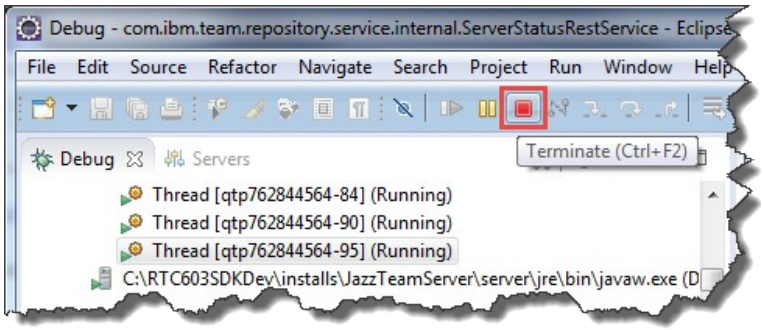
__d. Click the **Resume** toolbar button to resume execution of the server.



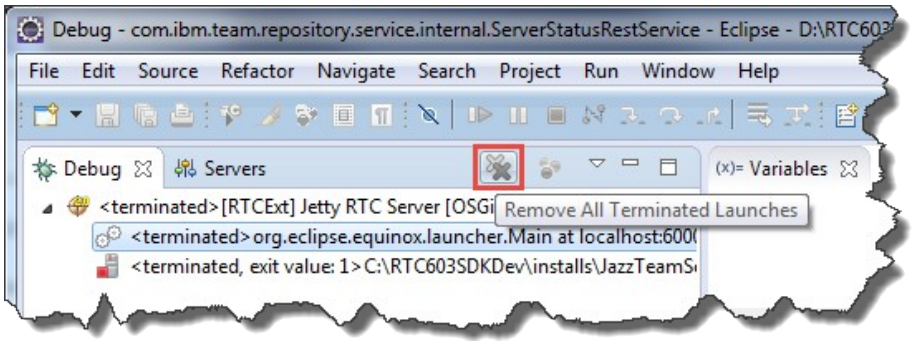
__41. Complete the test.

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

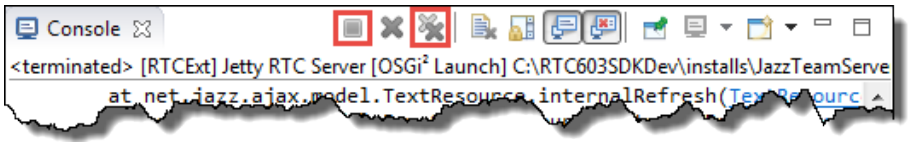
__b. You can now return to the RTC Eclipse client and terminate the server by clicking the **Terminate** toolbar icon in the **Debug** view as shown here in the Debug or in the **Console** view.



__c. Remove all terminated launches by clicking the toolbar icon. This provides you with a better overview about what still runs and what not. 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 launches, remove all terminated launches, check nothing is running any more and start again. Both options, terminating the launch and removing all terminated launches are available in the Debug view and the Console view.



Understand what is running

Make sure to understand what is running. It is easy to overlook running launches and accidentally start more than one launch.

Launches can have conflicting ports so only one instance will be able to run. If the system does not behave as expected, please check how many instances run. Stop all launches and start again, making sure only one is launched.

1.7 Setup the Eclipse workspace for Client SDK Development

In this section you will setup your RTC Eclipse workspace for developing RTC Eclipse Client plug-ins. This consists of specifying what target 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 Plug-in Development Environment (PDE) know about all the RTC platform client API source code.

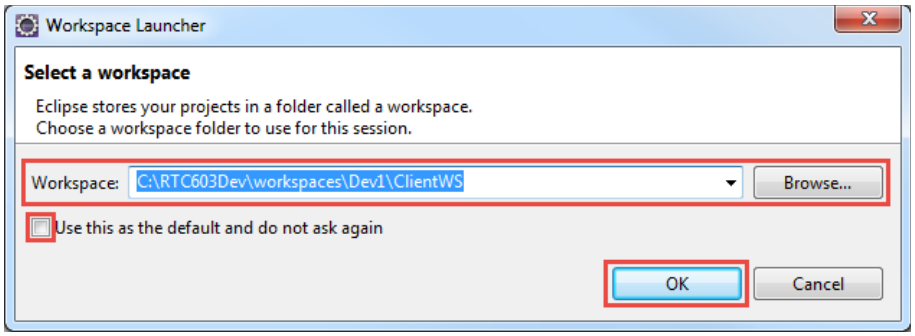
This is very similar to 1.3 Setup the Eclipse workspace for Server SDK Development with a new workspace and using the RTC Client SDK.

__42. Start a new Eclipse client with a new workspace used for RTC Eclipse client API development and prepare the RTC Client SDK target platform.

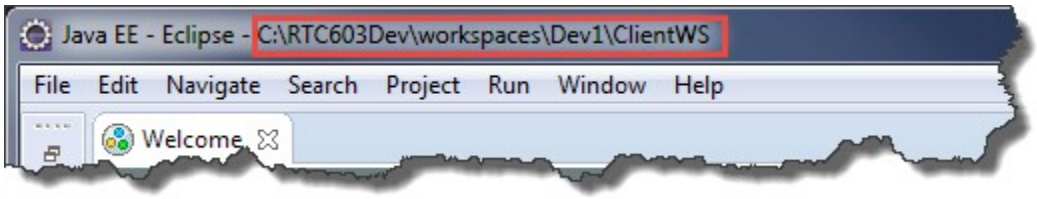
__a. Start a new Eclipse client. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

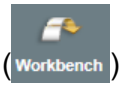
__b. When prompted, select an Eclipse workspace. This Eclipse workspace is going to be used for development for the RTC Eclipse Client API.

__i. These instructions will use `C:\RTC603Dev\workspaces\Dev1\ClientWS` as path for this Eclipse workspace. Enter the path for the new workspace. Don't check the “Use as default” check box. Click OK.



__ii. Check the desired Eclipse workspace is in use.

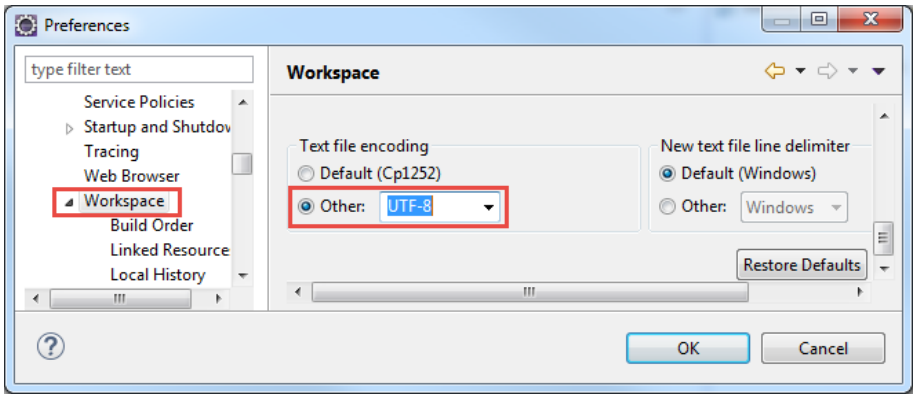


__c. Minimize the **Welcome** screen via this (

) button near the top of the window.

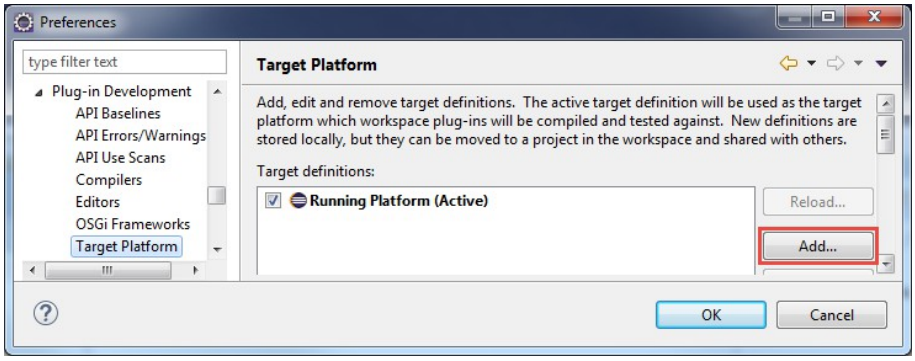
__43. Configure the Eclipse workspace and switch Text file encoding to UTF 8

__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.

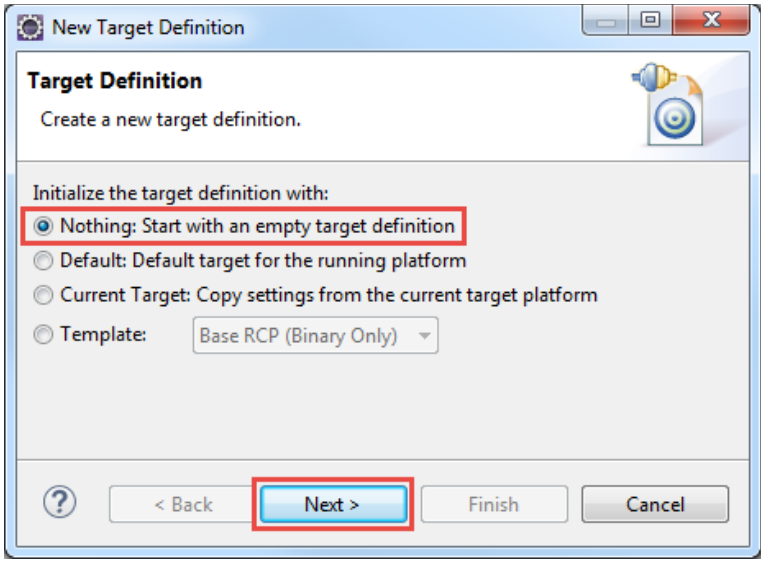


__44. Create a new target platform.

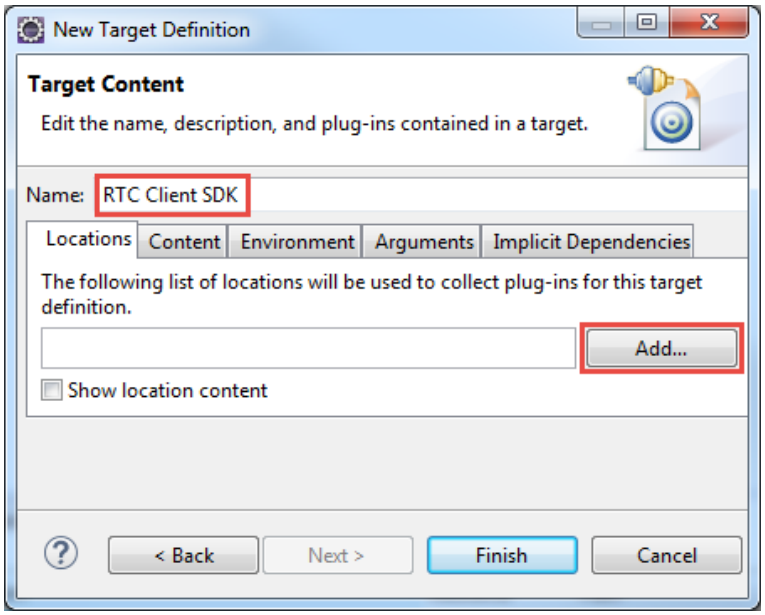
__a. From the menu bar, select **Window > Preferences**. In the **Preferences** dialog, select **Plug-in Development > Target Platform**. Wait for the load process to finish, then click **Add...**



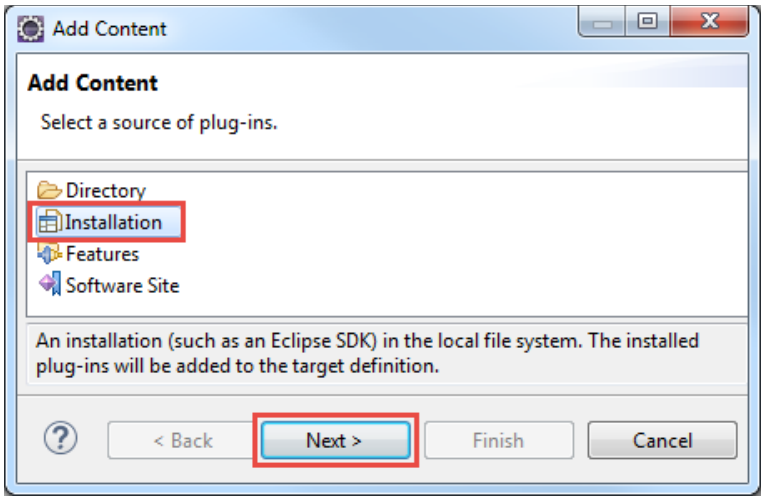
__b. In the **New Target Definition** wizard, select **Nothing: Start with an empty target definition** and then click **Next**.



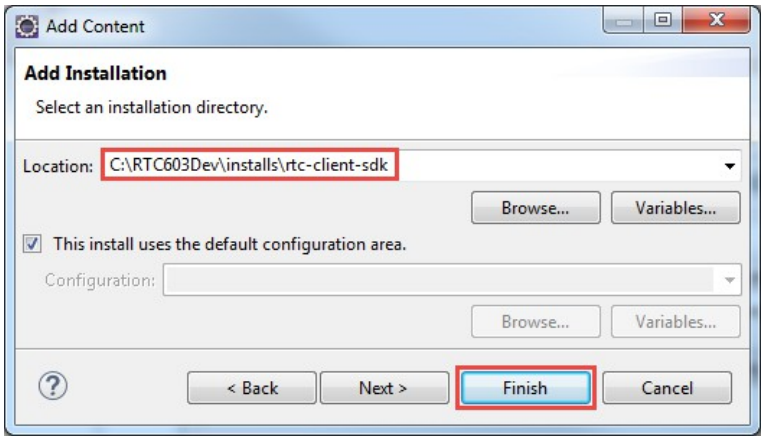
__c. On the second page of the wizard, enter *RTC Client SDK* as the **Name** and click **Add...**



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

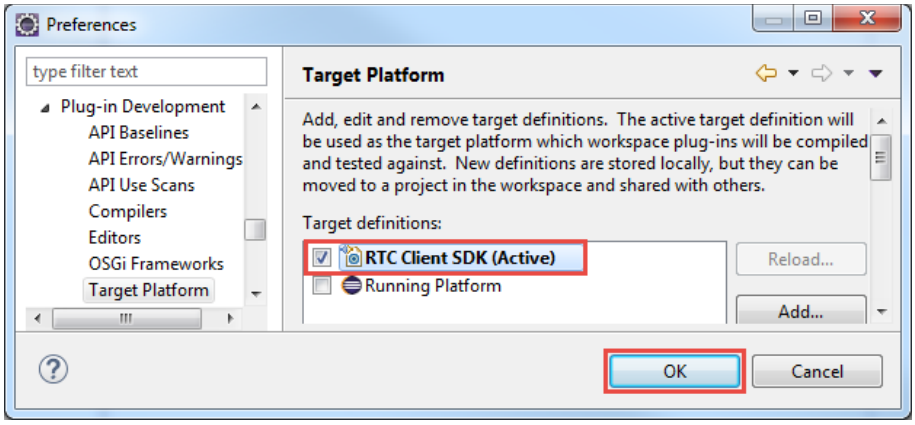


__e. On the second page of the wizard, enter `C:\RTC603Dev\installs\rtc-client-sdk` as the **Location**, or browse to the location and then click **Finish**.



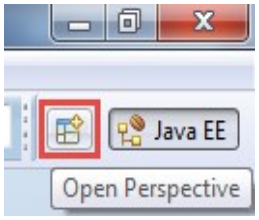
__f. After the operation completes, click **Finish** in the **New Target Definition** wizard.

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

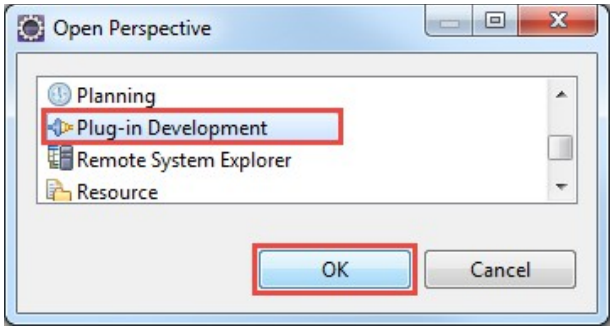


__45. Open the Plug-in Development perspective.

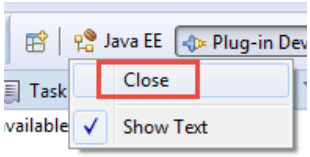
__a. In the toolbar toward the right, click the **Open Perspective** button.



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

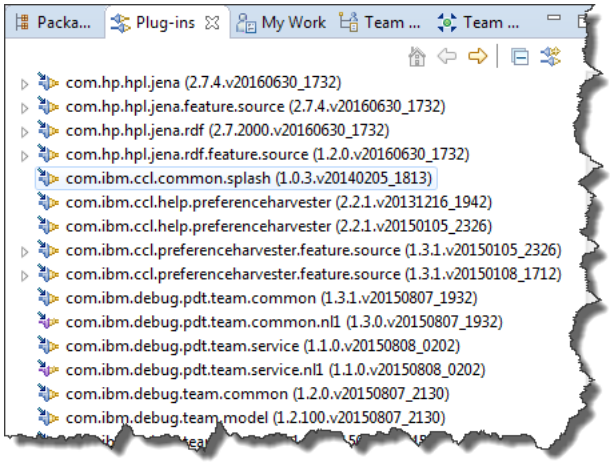


__c. Close the Java EE and any other perspective except **Plug-in Development**.

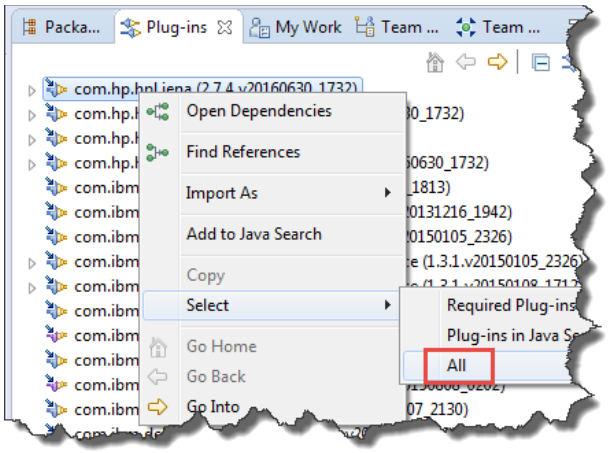


__46. Add RTC source code to Java search.

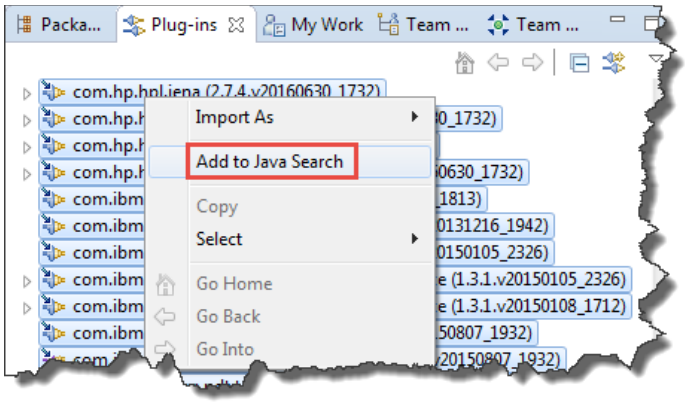
__a. On the left, select the **Plug-ins** view.



__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. This menu has been renamed to 'Add to Java Workspace Scope' in newer Eclipse versions.



1.8 Finalize setup of the Eclipse workspace for Client SDK Development

In this section you will finalize the setup your RTC Eclipse workspace for developing RTC Eclipse Client SDK plug-ins. You will connect to the RTC development Server, load the components needed for client API development and import required files into the workspace.

This is similar to 1.4 Finalize setup of the Eclipse workspace for Server SDK Development with changes for the client SDK.

__47. Start the RTC development server, if it is not already started.

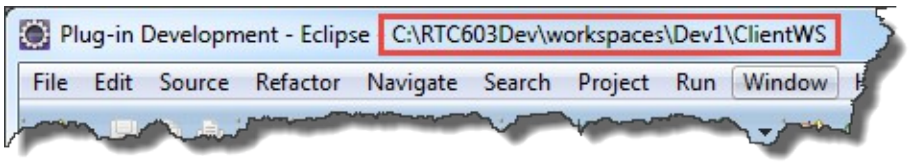
__a. Open a Windows Explorer and navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file.

__48. Return to the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS` you have already running .

__a. If your RTC Client SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

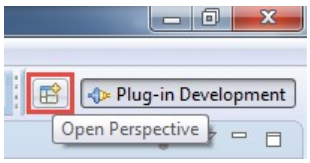
__i. When prompted, select the Eclipse workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS`. Don't check the “Use as default” check box.

__b. Check the desired Eclipse workspace is used.

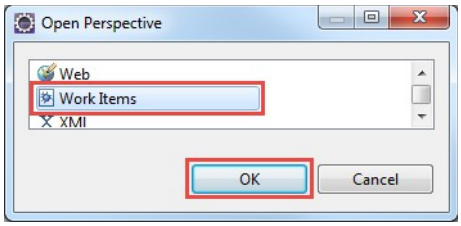


__49. Open the Work Items perspective.

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

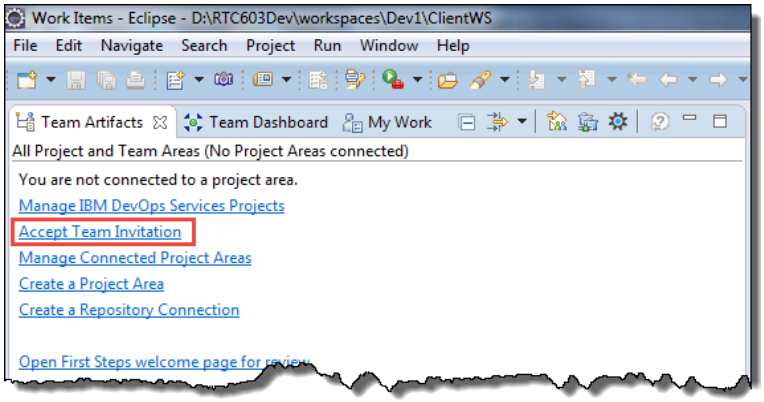


__b. In the **Open Perspective** dialog, select **Work Items** and then click **OK**.



__50. Connect to the project area.

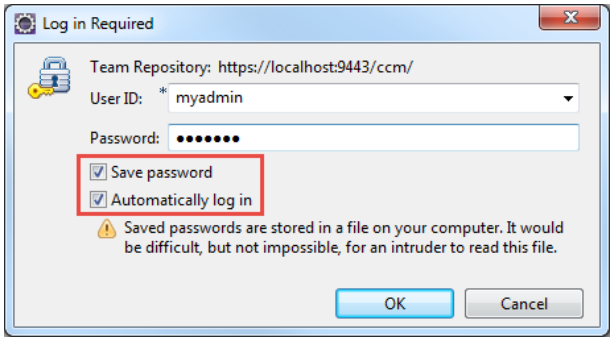
__a. On the left, switch to the **Team Artifacts** view and click the **Accept Team Invitation** link.



__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**.



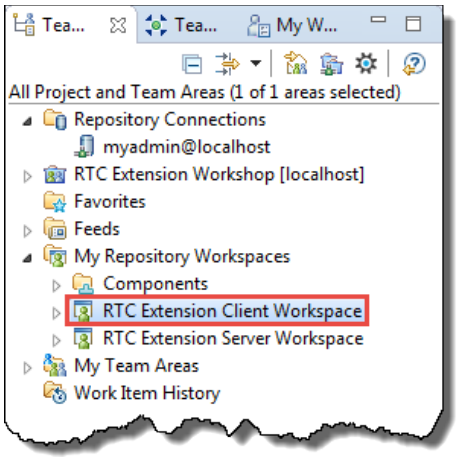
__d. If prompted with a **Repository Connection Certificate Problem**, select the **Accept this certificate permanently** radio button and then click **OK**.

__e. If asked for a password for secure storage, cancel. You will not be able to store the password and have to enter it each time you want to log in.

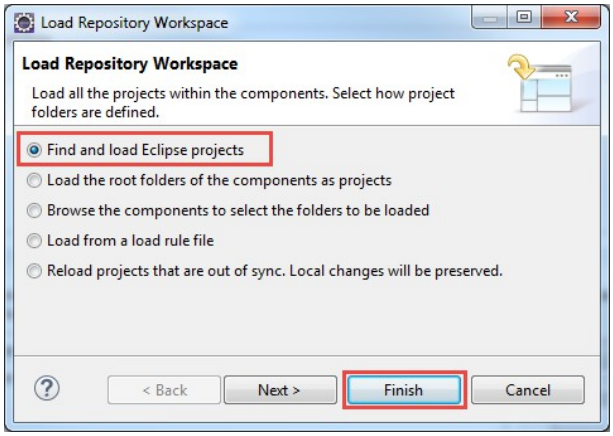
__f. Close the project area editor that opens.

__51. Load the workshop client development repository workspace.

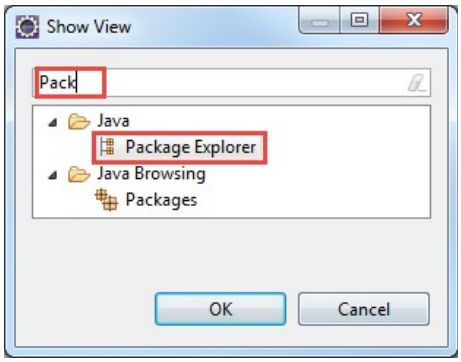
__a. In the **Team Artifacts** view, expand the **My Repository Workspaces** node, right click the **RTC Extension Client Workspace** and then select the **Load…** action from then menu. Note that in later versions of EWM/RTC the node was renamed to 'My Source Control' and sub-nodes where added. The node 'My Repository Workspaces' from the screen shot is now named 'My Workspaces'.



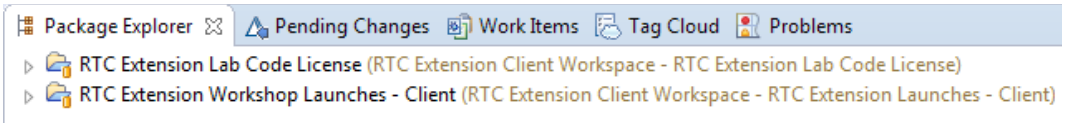
__b. In the **Load Repository Workspace** wizard, make sure **Find and load Eclipse projects** is selected and then click **Finish**.



__c. Open a **Package Explorer** view. Open **Window>Show View**, select **Other**. In the filter type **Package**, select the **Package Explorer** and click **OK**.



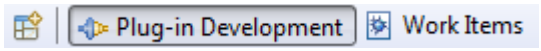
__d. Verify that there are now two new Eclipse projects in your **Package Explorer** view. The project (**RTC Extension Lab Code License**) contains the license agreement for the sample code you are using in this workshop. The second (**RTC Extension Workshop Launches – Client**) contains an Eclipse launch configuration used to start an Eclipse client for RTC Eclipse Client API development. In the rest of this lab you will learn how to use the launch.



__e. You will also notice in the **Pending Changes** view, that there are unloaded components, and incoming change sets and baselines. Do not accept them yet. 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.

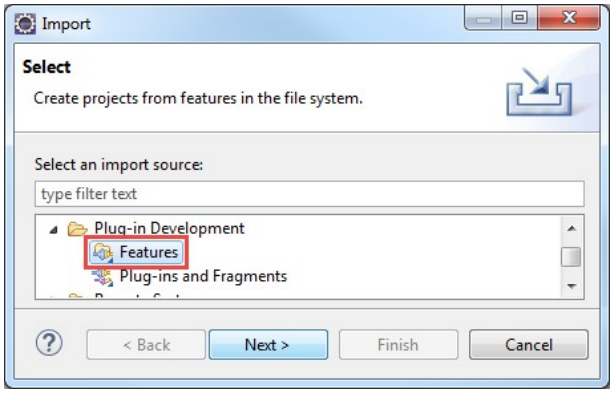
__52. Open the **Plug-in Development perspective**.

__a. In the toolbar toward the right, click Plug-in Development to switch the perspective.



__53. Import a feature to make launching the RTC Eclipse client much easier. It is also 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**.



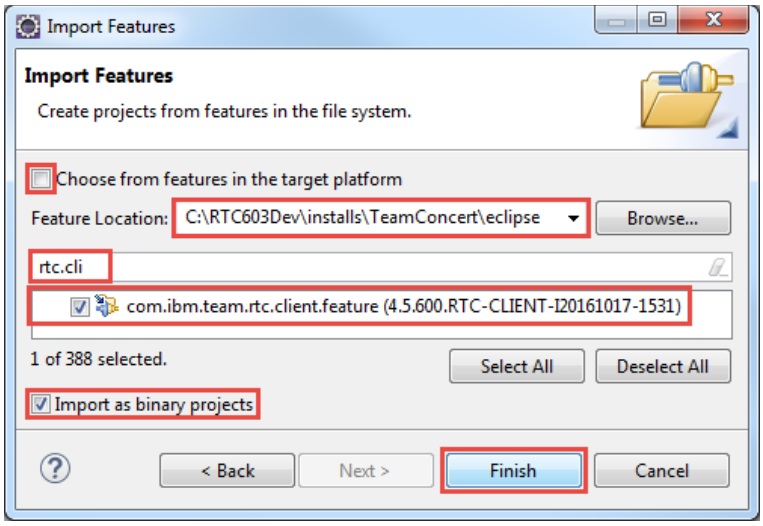
__b. On the second page of the wizard

__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:\RTC603Dev\installs\TeamConcert\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**.

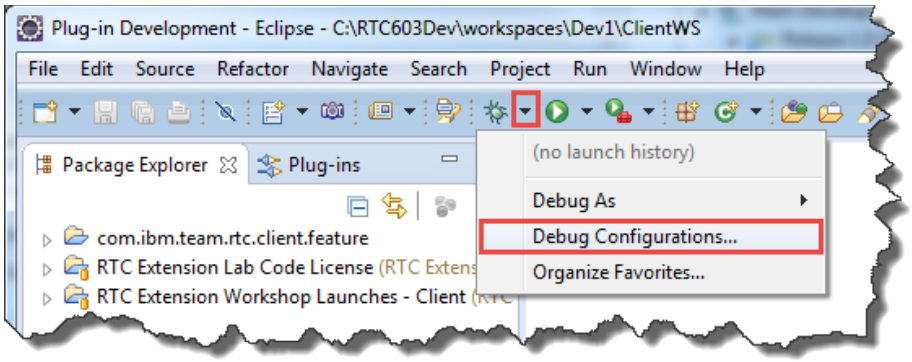
__c. Check the imported project shows up in the package Explorer view.

1.9 Test the RTC Eclipse client launch

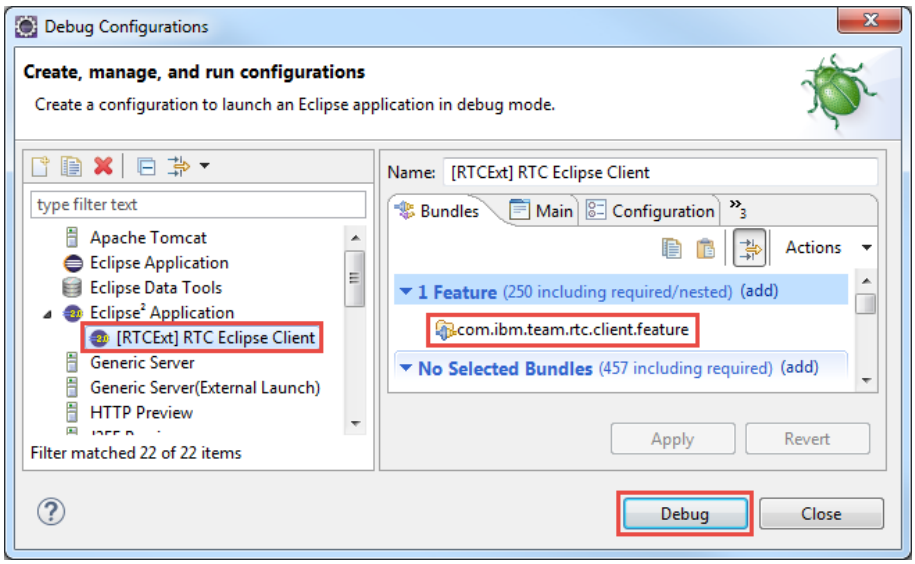
Up to this point you have only been debugging RTC servers. You will sometimes want to extend and debug RTC Eclipse clients too. You will test a launch for that in this section. You can use this launch to run or debug the RTC Client. This is possible because the server and the client are debugged in two different Eclipse workspaces. Debugging two different launches in one workspace can result in deadlock situations.

__54. Launch the RTC Eclipse client under debug.

__a. Select **Debug Configurations…** from the drop-down menu of the **Debug** toolbar icon.



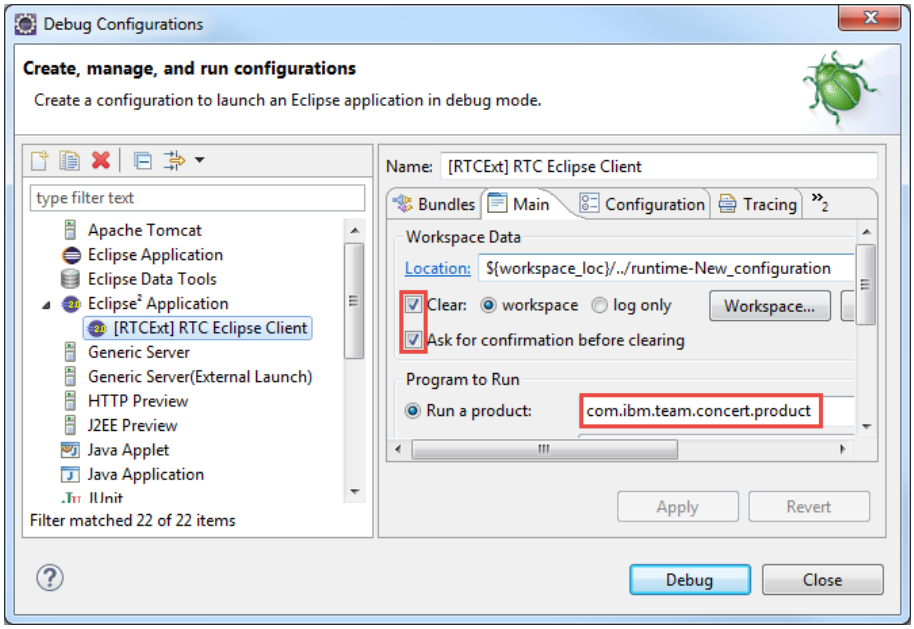
__b. In the **Debug Configurations** dialog, select **Eclipse² Application > [RTCExt] RTC Eclipse Client** and then click **Debug**. Note that on this **Bundles** tab, that the feature you imported earlier (the RTC client feature) is included in this launch.



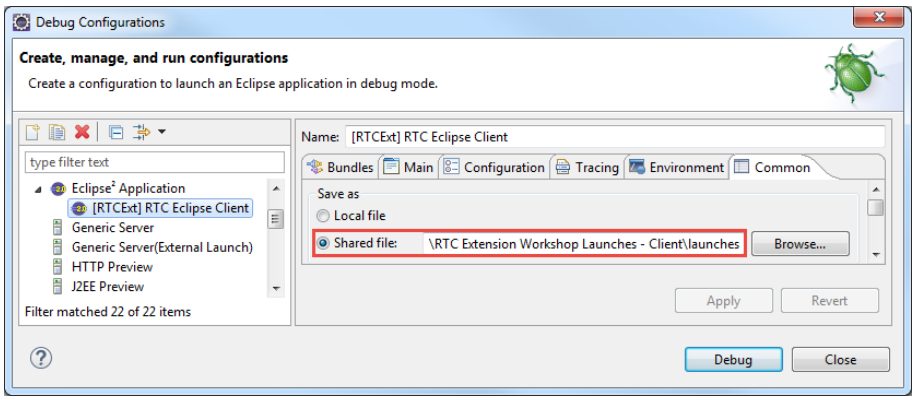
__c. If you switch to the **Main** tab, you will notice two important settings.

__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**.



__d. If you switch to the **Common** tab, you will notice the setting shared file which allows to store the launch file in a project area instead of the workspace specific default location in `.metadata\.plugins\org.eclipse.debug.core\.launches`. This makes it possible to keep the launch under version control as well.



__e. Click **Debug** to start the [RTCExt] RTC Eclipse Client launch file.

__f. 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).

__55. 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.

__56. Close the RTC Eclipse client you just launched under debug.

__57. Shutdown unless proceeding to lab 2.

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

__b. Return to the Windows Explorer and navigate to `C:\RTC603Dev\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.

Please note the optional section in this lab below, that explains how to backup your workspaces and how to setup a workspace for development with the Plain Java Client Libraries.

1.10 Set up a workspace for Plain Java Client Library development

The Extensions Workshop was created to show how to develop server and client plug-in based extensions. It is also possible to develop and run plain Java applications like the WorkshopSetup tool against EWM/RTC by using the Plain Java Client Libraries.

The Plain Java Client Libraries do not ship with source code, but it is possible to use the source code of the SDKs to see the Plain Java Client Libraries API source code.

This optional section explains how to set up a development environment for the Plain Java Client Libraries with full visibility of the SDK Source code and the ability to run and debug a Plain Java Client Libraries application, based on the workspaces created above.

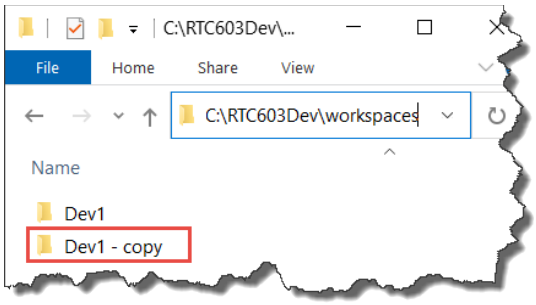
__58. Open the folder `C:\RTC603Dev\installs\PlainJavaApi\doc` and browse the JavaDoc documentation.

__59. Copy an existing set of workspaces to get a new one to work with.

__a. The way how the Eclipse workspaces are set up, allows to easily separate this optional section form the main workshop. We will copy an existing set of workspaces and create a new one that will then be used.

__b. Make sure to close all Eclipse clients.

__c. Open a windows explorer and navigate to **`C:\RTC603Dev\workspaces`**. Select the folder Dev1 and create a copy using CTRL-C CTRL-V.



__d. Change the new folder name to **Dev2**.

__60. Note: If you have not yet started with the next labs, you can create a backup of the workspaces in order to keep the starting point for the labs. This is certainly a good idea given how long it took to get here. Compress the Dev1 folder with 7Zip and keep the compressed file as backup. You can also keep a backup of the final workspaces after the workshop finished.

__61. Open a new Eclipse client with the new copy of the workspace.

__a. Start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

__b. When prompted, use `C:\RTC603Dev\workspaces\Dev2\ServerWS` as path for the Eclipse workspace. Don't check the “Use as default” check box. Click **OK**.

Which Eclipse Workspace to use?

The folder `C:\RTC603Dev\workspaces\Dev2` contains the Eclipse workspaces **ServerWS** and **ClientWS**. What is the difference between them and which one to use?

The workspace **ServerWS** contains a server SDK, which is actually a client SDK and a server SDK. This SDK contains all the Java API there is. It also contains a lot of JUnit tests and other example source code.

The workspace **ClientWS** only contains a client SDK that supports newer Eclipse versions compared to the server and client SDK in ServerWS.

The following lab uses the ServerWS. This allows to search for client, common and server API, if needed. It also allows to use the existing Jetty Server for testing.

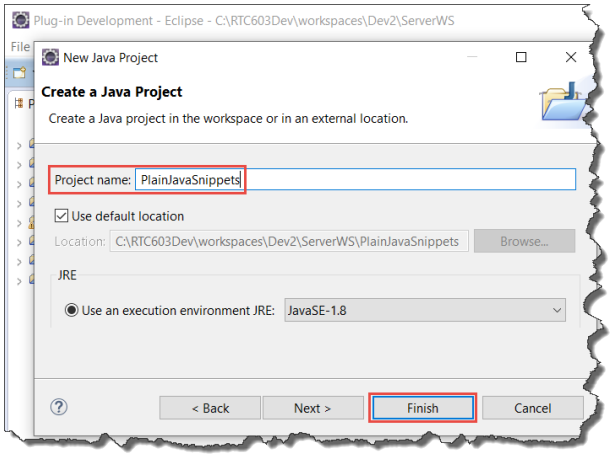
__c. Switch to the Plug-in Development perspective. Note the Eclipse projects related to the imported features are all visible. Keep the projects as they are.

__d. Note that the workspace is fully configured with the Target Platform and source code. The class files that were used for debugging are still visible with source code. If you have closed them you can open them using **Navigate>Open Type** and search for ***active*ser** again. Open the **ActiveServiceDTO** interface and see the code.

__62. Create a new Java Project to explore the Plain Java Client Library Snippets.

__a. In the Plug-in Development perspective create a new Project using **File>New>Project**. In the wizard type Java into the filter. Select **Java>Java Project**. Click **Next**.

__b. Name the project **PlainJavaSnippets** and click **Finish**.



__c. Let Eclipse change to the Java Perspective. Now unfold the project, **right click** on the folder **src** and create a new package. Name it **snippets**.

__63. Import the Plain Java Snippets code into the project.

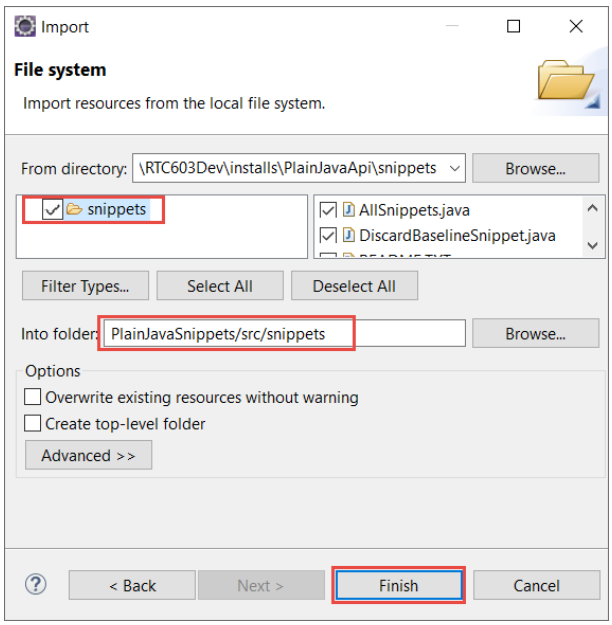
__a. **Right click** on the new package snippets and select **Import** in the context menu.

__b. In the wizard select **General>File System.** Click **Next**.

__c. Browse and select the folder `C:\RTC603Dev\installs\PlainJavaApi\snippets`.

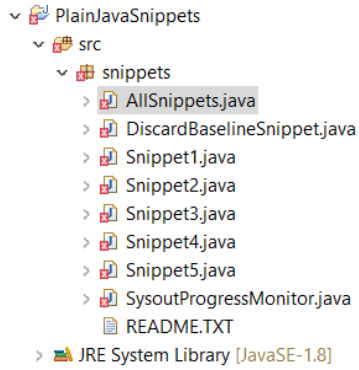
__i. **Check** the check box in front of the folder snippets to import everything.

__ii. In the **Into Folder** selection field, make sure it shows your package src/snippets in your PlainJavaSnippets project as presented below, then click **Finish**.

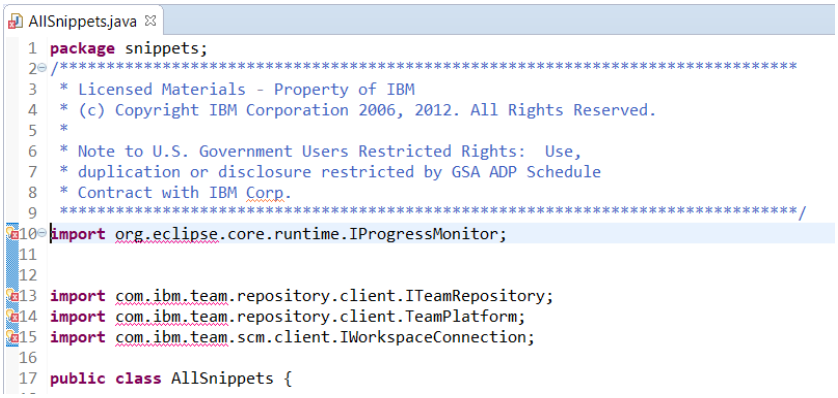


__64. Examine the source code

__a. Unfold the package **snippet** and note the errors.

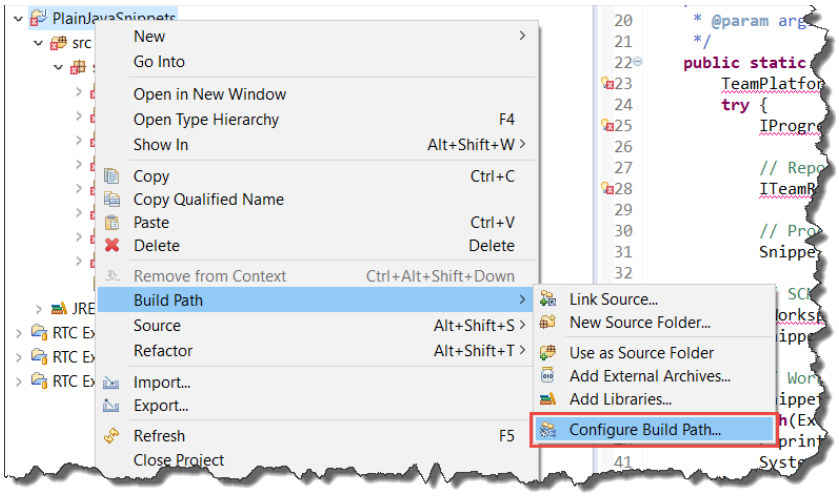


__b. Double click **AllSnippets.java** to open it. Unfold the imports and review the first lines. Note that the compiler can not resolve any of the import dependencies.



__65. Add the plain java client libraries to the **Build Path** of the project.

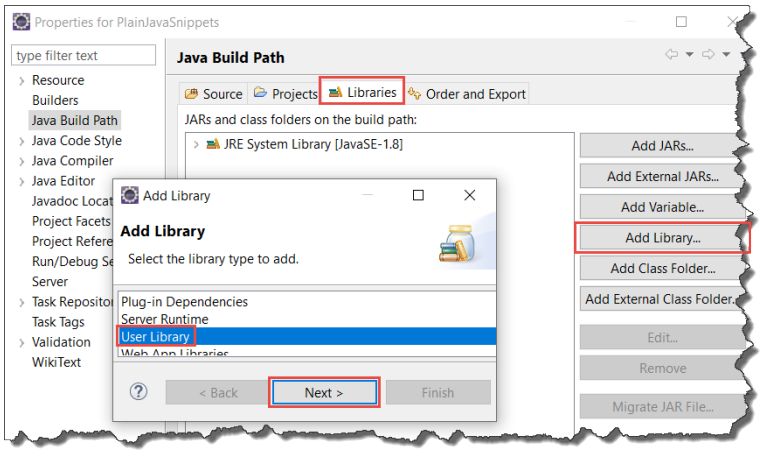
__a. **Right-click** on the project and select **Build Path>Configure Build Path**.



__b. Add a user library.

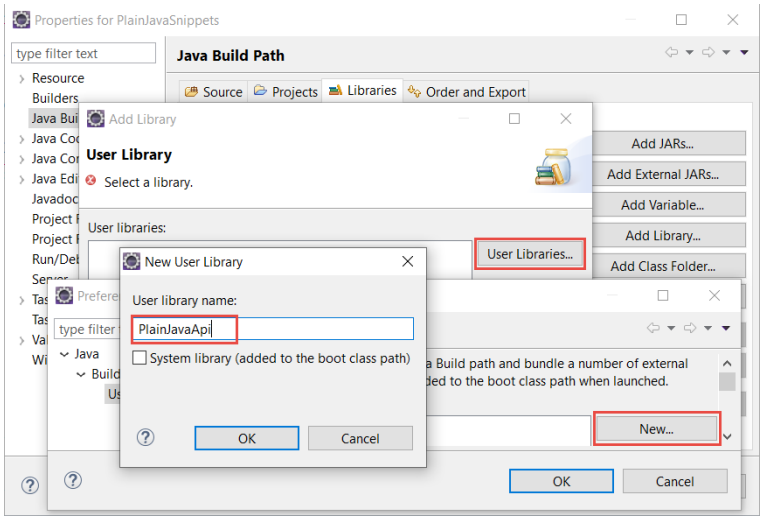
__i. In the Java Build Path wizard select the tab **Libraries**. **Click** the **Add Library** button.

__ii. In the Add Library wizard select **User Library** and click **Next**.

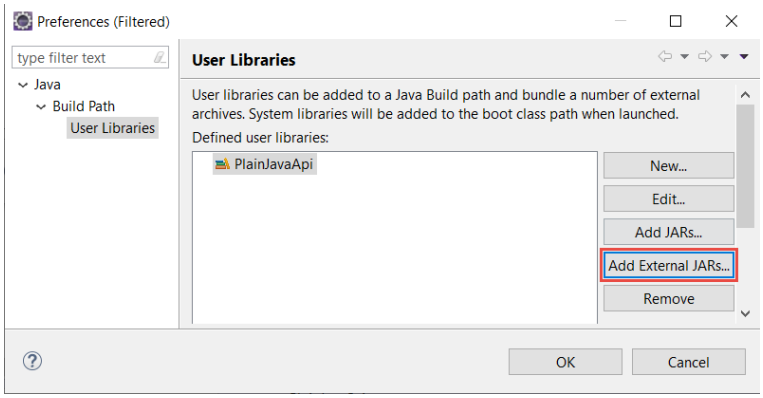


__iii. Click on the button **User Libraries…** . In the User Library wizard click **New...**.

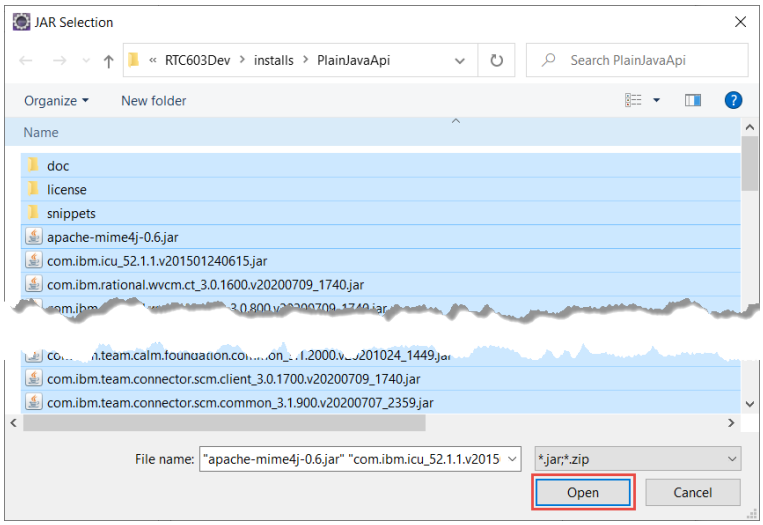
__iv. In the **New User Library** wizard, enter then name **PlainJavaApi**. Click **OK**.



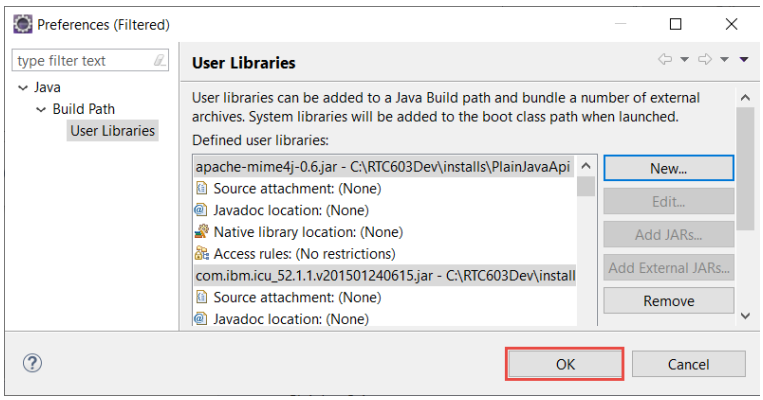
__v. In the User Libraries wizard click **Add External JARs...**.



__vi. Browse to the folder `C:\RTC603Dev\installs\PlainJavaApi`. Select all files and folders and click **Open**.

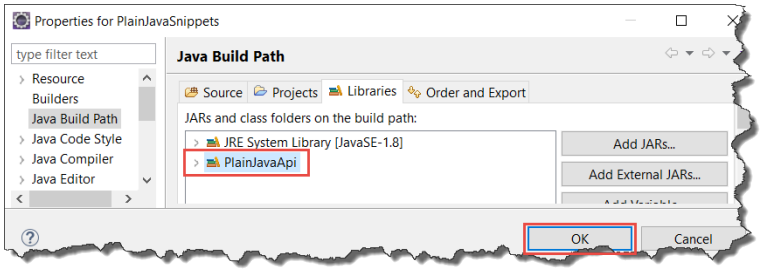


__vii. In the User Libraries wizard review the added JAR files and click **OK**.



__viii. Back on the User Library wizard click **Finish**.

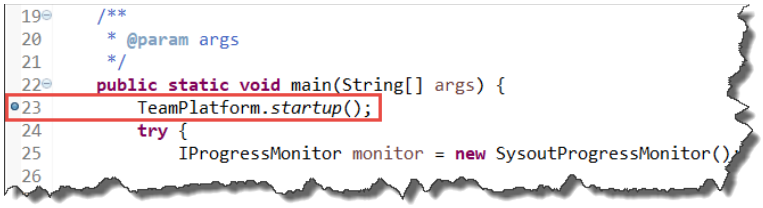
__ix. Check the Library is now added and click **OK**.



__x. Note the compiler errors are gone after the rebuild has finished. Run a **Project>Clean** if they persist.

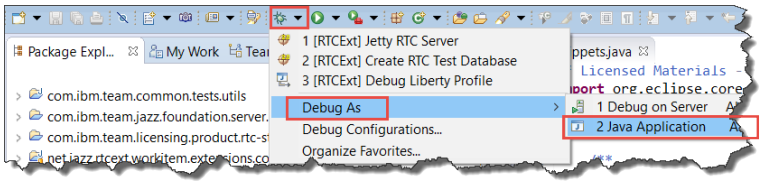
__66. Prepare the snippets for debugging.

__a. Open the AllSnippets.java. Set a breakpoint in the first line of the main() method.



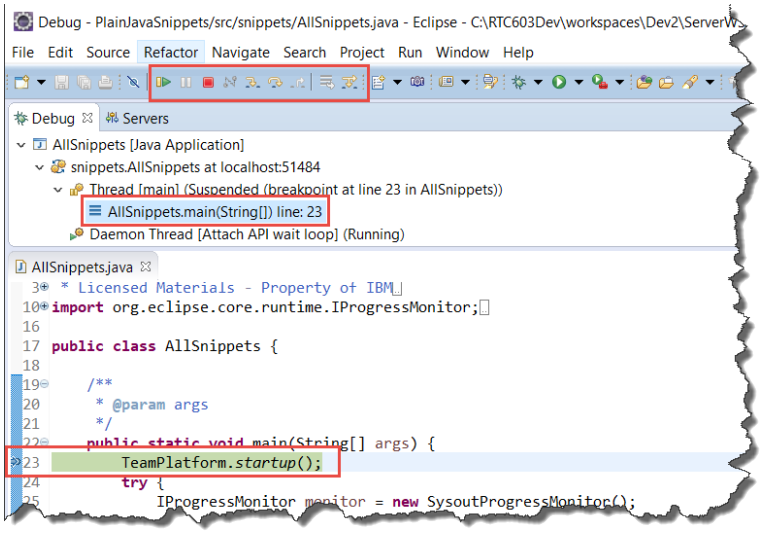
__b. Create a new debug launch for AllSnippets.java.

__i. Right click on AllSnippets.java in the Package Explorer. Click on the triangle right to the debug icon in the menu. Select **Debug As>Java Application**.



__ii. A debug configuration is created and the main method for AllSnippets is called.

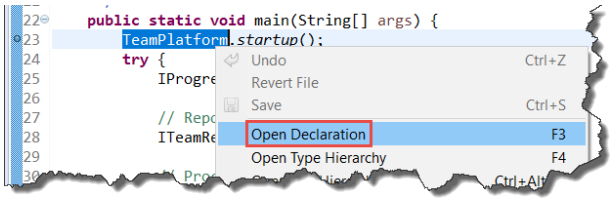
__iii. The breakpoint will hit. Allow Eclipse to enter the debug perspective.



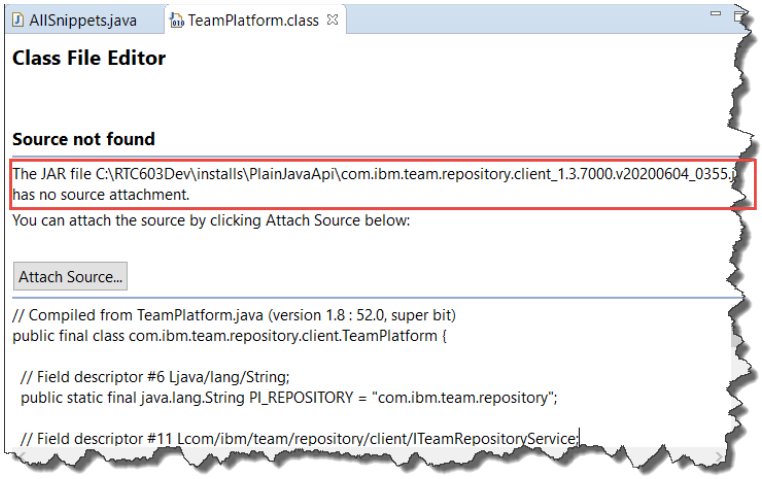
__iv. Step over one or two statements, then terminate the debug session.

__67. Try to look at the API code.

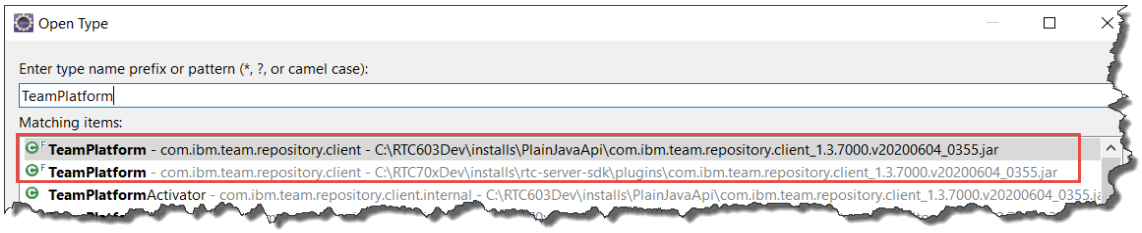
__a. In the class editor for AllSnippets.java, **right click** on the class TeamPlatform. Select **Open Declaration**.



__b. A class editor opens that shows information about the class, but does not show any code, because there is no source code shipped with the Plain Java Client Libraries.

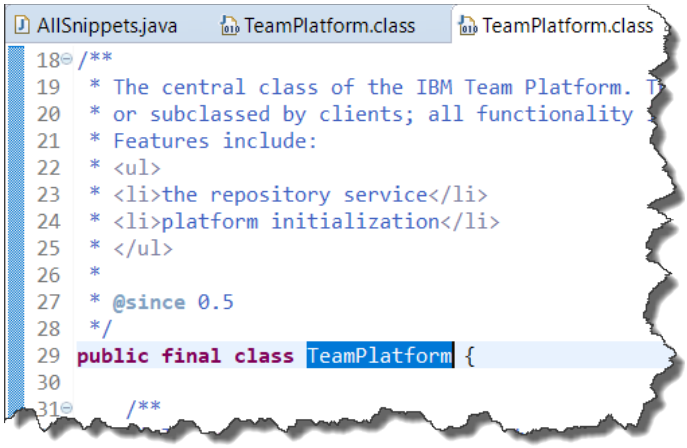


__c. In the Menu use **Navigate>Open Type**. Type **TeamPlatform** into the selection field and inspect the search result.



Note the two hits for JAR files containing TeamPlatform. The details show that one of them is located in the folder `C:\RTC603Dev\installs\PlainJavaApi` the other one is located in the folder `C:\RTC603Dev\installs\rtc-server-sdk`. Select the one located in `C:\RTC603Dev\installs\PlainJavaApi` and click **OK**. The class editor without code opens.

__d. Perform the last steps again but select the TeamPlatform that is located in the folder `C:\RTC603Dev\installs\rtc-server-sdk`. The class editor now shows the source code of the class `com.ibm.team.repository.client.TeamPlatform`.

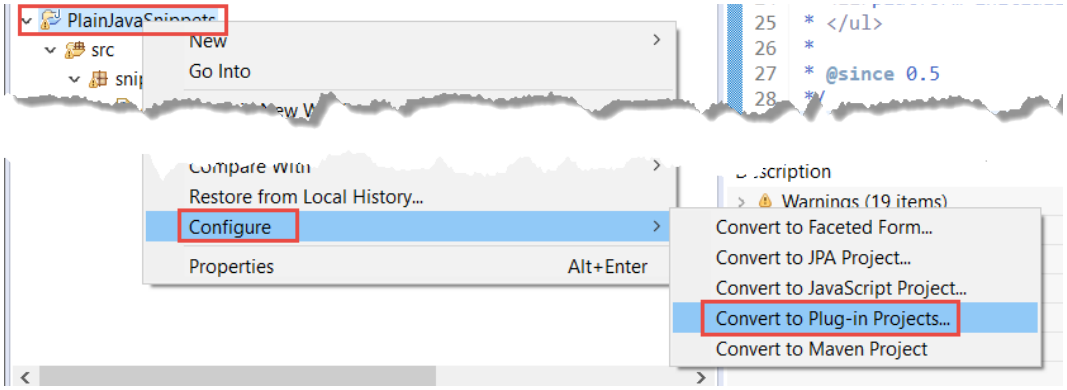


The reason for this is, that the source code for the RTC/EWM API classes is part of the server and the client SDK. The Plain Java Client Libraries contain the same classes, based on the same code, but not the source code. The mechanism to make the source code searchable and display it when opened is based on the Plug-in Development Environment (PDE) that is part of Eclipse and allows extending Eclipse and its UI based on the Rich Client Platform (RCP) to be extended with plug-ins.

It is possible to trick Eclipse into providing the source code that is available in the Plug-in Development Environment while developing a plain Java application that is not meant to be deployed as plug-in by pretending to develop a plug-in.

__68. Configure the Java project PlainJavaSnippets into a plug-in project and configure the result to show the source code for the Plain Java Client Libraries in the SDK.

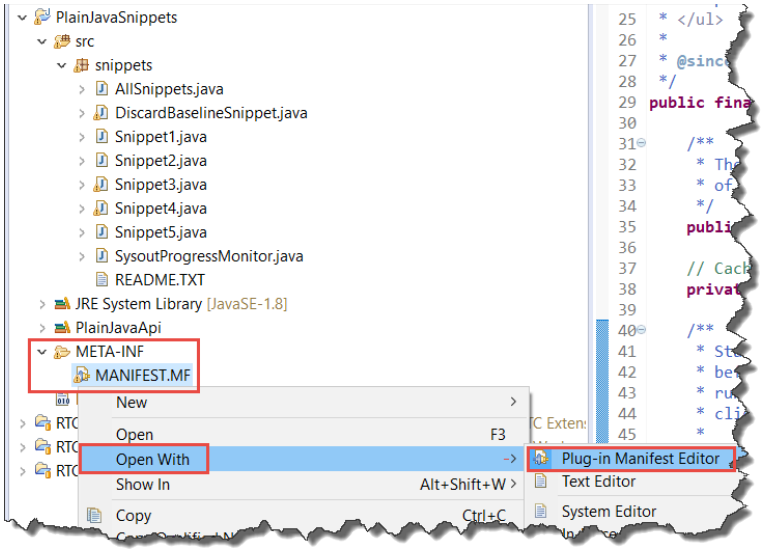
__a. Right click the project PlainJavaSnippets in the Package Explorer. Select **Configure>Convert to Plug-in Project...**.



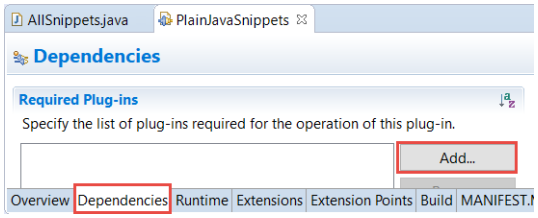
__b. In the Configure Existing Projects make sure only the project **PlainJavaSnippets** is selected then click **Finish**. Wait until the project is converted.

__69. Open the Plug-in Manifest.

__a. Select the file MANIFEST.MF that was created in the new folder META-INF and open it with the Plug-in Manifest Editor.



__b. In the editor switch to the tab Dependencies. You will only see code for packages that are in the dependency list of Required Plug-ins. The button **Add...** is used to add the dependencies needed.



__c. Look at the imports in the file AllSnippets.java to understand what plug-ins/packages need to be referenced. The package names are usually a prefix of the fully qualified class name. For example `com.ibm.team.repository.client.` The RTC/EWM API often uses the name common or client in the package name. The prefix before that is the domain name space e.g. `com.ibm.team.repository`. Analyze the imports below:

```
import org.eclipse.core.runtime.IProgressMonitor;


import com.ibm.team.repository.client.ITeamRepository;
import com.ibm.team.repository.client.TeamPlatform;
import com.ibm.team.scm.client.IWorkspaceConnection;
```

This shows that at least the following packages are needed:

```
org.eclipse.core.runtime
com.ibm.team.repository.client
com.ibm.team.scm.client
```

__d. Checking the other imports for the other classes reveals the following additional dependencies:

```
com.ibm.team.repository.common
com.ibm.team.scm.common
com.ibm.team.filesystem.client
com.ibm.team.filesystem.common
com.ibm.team.process.common
com.ibm.team.process.client
com.ibm.team.foundation.common
com.ibm.team.workitem.client
com.ibm.team.workitem.common
```

__e. Add these dependencies to the dependency list. Click **Add...**. Type the package name org.eclipse.core.runtime in the search field. Notice that the matching items list becomes smaller while you type. This can help to find potentially interesting packages later.

In the matching items select `org.eclipse.core.runtime` and click **OK**.

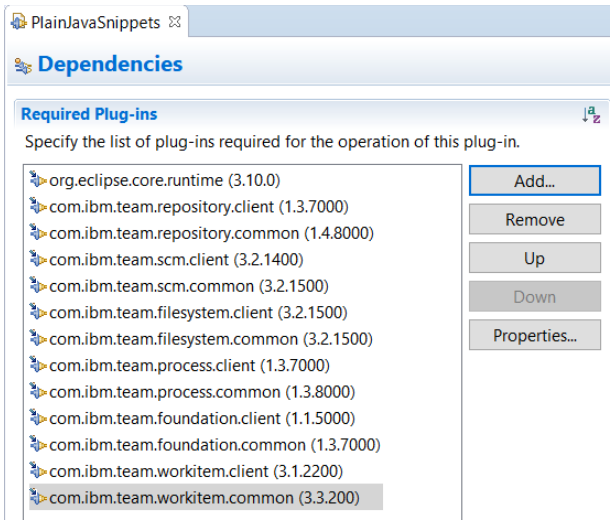
Type `com.ibm.team.repository.` and notice that there are many choices. Select `com.ibm.team.repository.`**`client`** and `com.ibm.team.repository.`**`common`** using the **CTRL** key and then click **OK**.

__f. Repeat this for the namespaces `com.ibm.team.scm.`, `com.ibm.team.filesystem.`, `com.ibm.team.process.`, `com.ibm.team.foundation.`, `com.ibm.team.workitem.`

__i. Add the **common** and **client** packages for all of them.

__ii. Note that there are more choices that might be of interest later. It is likely better to have more dependencies as too few. When source code does not show during debugging, it is an indicator that there are packages missing.

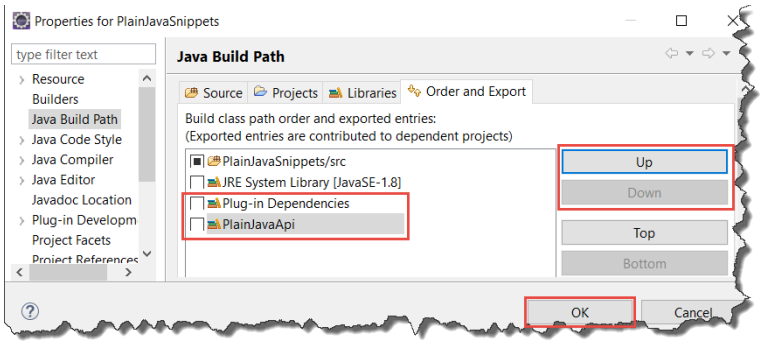
__g. Save your changes



__70. Order the libraries to make sure the Plug-in Dependencies are looked up first.

__a. Right click on the **PlainJavaSnippets** project, select **Build Path>Configure Build Path...**.

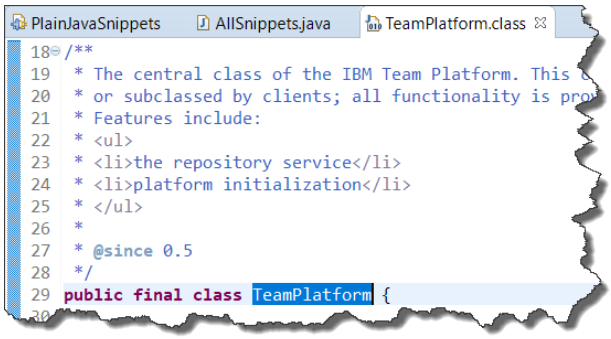
__b. On the Order and Export tab select the entry named **PlainJavaApi** and use the Down button to move it below the entry named **Plug-in Dependencies**. Then click **OK**.



__71. Try to find the source code again.

__a. Open the file AllSnippets.java in an editor.

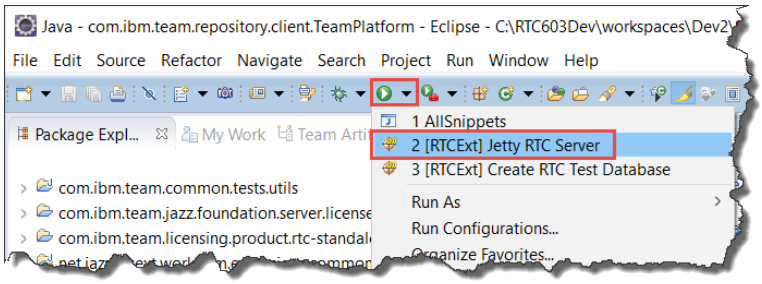
__b. Right click on the TeamPlatform class in the first line of the main() method. Select **Open Declaration**. The editor that opens should now show the source code.



__c. It is important to maintain the dependency list and the correct order of the libraries in the Build Path to be able to see the source code.

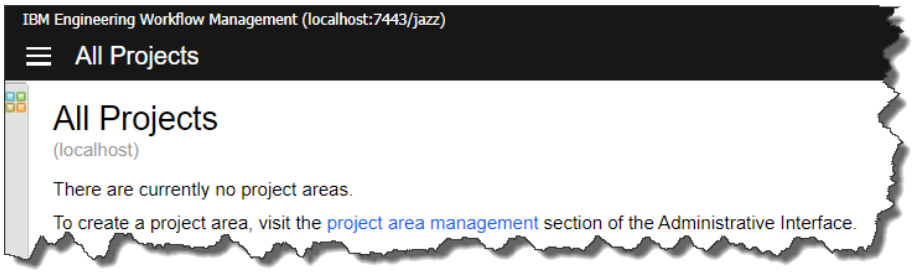
__72. Start the Jetty Debug server to have a target server to run AllSnippets.java as regular Java application.

__a. Select and **run** the configuration **[RTCExt] Jetty RTC Server**.



__b. Check the console and wait for the application to come up.

__c. Connect with a Web browser to https://localhost:7443/jazz/web and login with the user TestJazzAdmin1 and the same password. There are no projects yet.



__73. Run the Application as Java application in Eclipse.

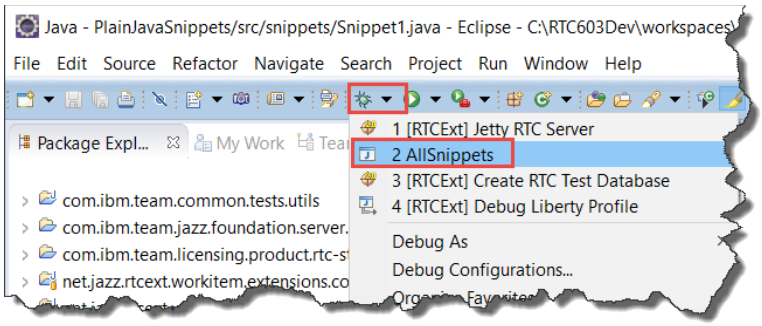
__a. Open and read the file README.TXT in the PlainJavaSnippets project. Note the user test with password test. We don't have such a user in the Jetty debug server so we need to adjust the application.

Open the file **Snippet1.java**. Change the default repositoryAddress to "https://localhost:7443/jazz" and change the default snippetUserAndPassword to “TestJazzAdmin1”. **Save** your changes.

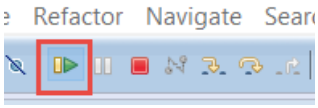
```
public class Snippet1 {

    private static String REPOSITORY_ADDRESS = System.getProperty("repositoryAddress", "https://localhost:7443/jazz");
    private static String USER_AND_PASSWORD = System.getProperty("snippetUserAndPassword", "TestJazzAdmin1");
```

__b. Select the AllSnippets Java application launch in the debug configuration menu.



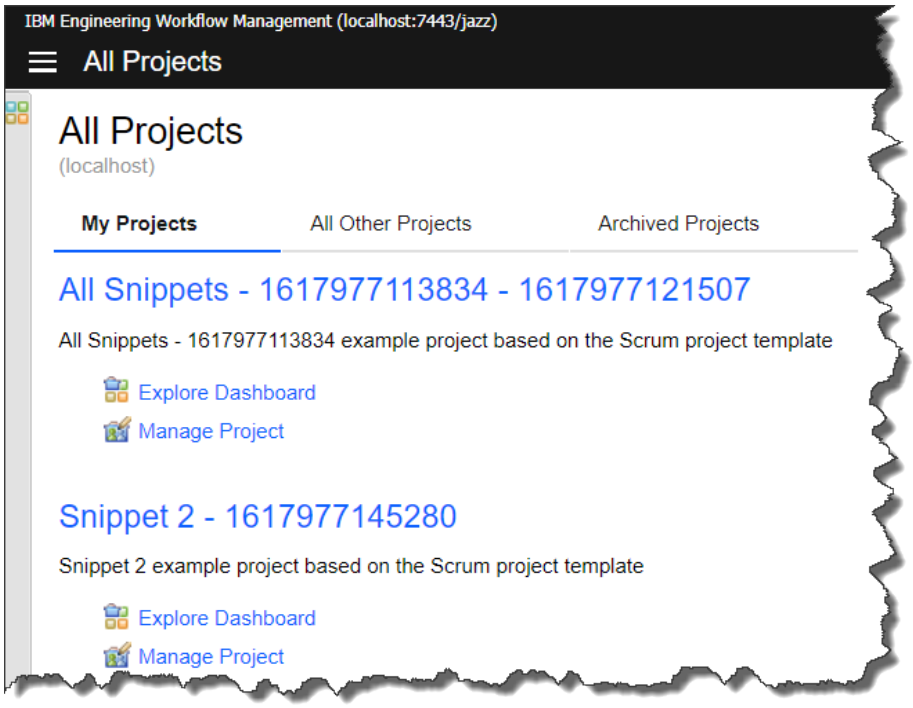
__c. The debugger will stop in the first line of main as before. Click the Resume button to continue running the application.



Observe the console. It will show plain java and Jetty messages, because all runs in this one context.

__d. The snippets will run and eventually fail because of a work item save precondition which can be ignored.

__e. Refresh the browser. The snippets have creates several project areas and performed some other activities.



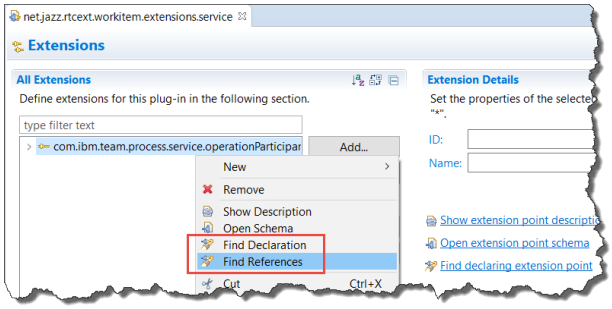
__74. Browse through the snippets to understand what they do. Explore the data they have created.

__75. Stop the debug server.

__76. You have now a fully working environment to develop Plain Java Client Library based applications. That the Eclipse projects are Plug-in development projects, does not matter for how the application is finally built and run. It is possible to create runnable and other flavors of jar files from the java code. The plug-in nature of the project is only needed to allow to use the PDE capabilities to search and show the source code of the API.

__77. Explore the search capabilities

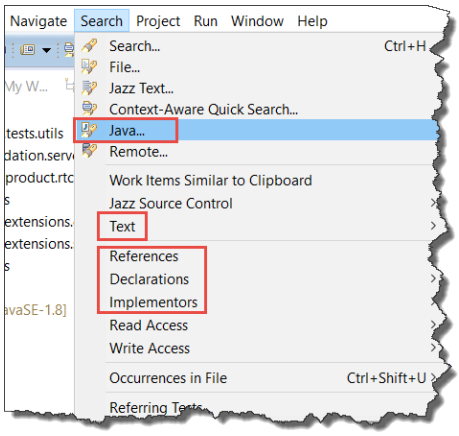
__a. Please note, the SDK bundles JUnit and other tests for the client and server APIs. It is possible to search the SDK, including the test code for examples, once the SDK is set up as explained. Eclipse provides capabilities to search for **references**, **declarations**, **implementations** for a number of objects including Java classes, interfaces, Extension points, Extensions.



This allows to search and find the source code for the classes, JUnit test code, plug-in manifests, extensions and other code in the SDK that can be explored for example usage of the API.

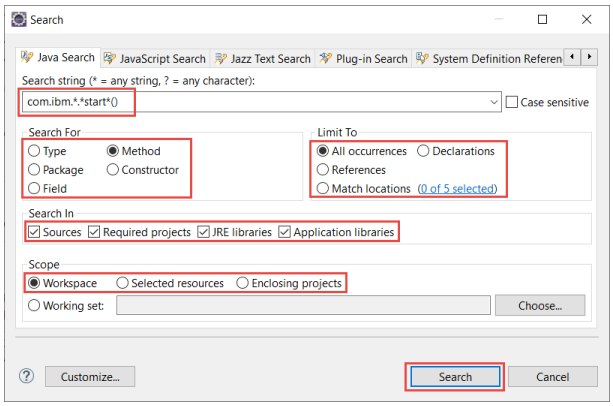
__b. Eclipse provides the **Navigate** menu that is used in the workshop.

__c. Eclipse provides the **Search** menu. This allows to search for a host of objects in various places and relationships to the current selection.

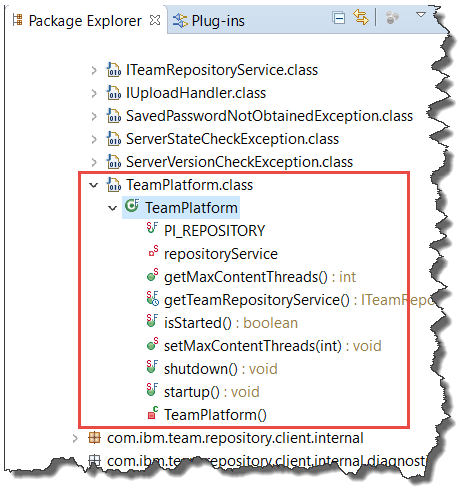


Note that the option to search Text requires text to be selected in an editor. You can enter text in editors such as the class editor. You can also create a text file for that purpose and enter text there.

__d. The Java search allows to search for Class names, Java Methods, including using pattern search using asterisk, in case some guesswork is needed.



__e. For Java objects e.g. in the SDK, it is possible to open the declaration and then show them in a view such as the package explorer. This allows to see which other classes exist in this specific package and further explore the API.



__f. Try to find the Classes used in the snippet and their source code with the methods shown above.

__78. Supported API

Please be aware that the Plain Java Client Libraries only contain a subset of the Client SDK. They do not contain the Server libraries at all, they do not contain the Rich Client Platform based portions of the client SDK and only a limited set of APIs (Work Item, Build, FileSystem, SCM, Interop). See the Plain Java client libraries JavaDoc for the classes that are supported. Classes that are not documented there are not supported.

Not supported means, IBM support will not be able to help with questions and problems with not supported API. The unsupported API can also change at any time.

__79. Make sure to add any API used to the plug-in dependencies to make sure the source can be found.

__80. Namespaces and APIs

Understand the name space pattern browsing and searching in the API.

__a. Namespaces that contain the section **client** are **client API** that is only available in the client SDK.

__b. Namespaces that contain the section **common** are **common API** that is available in the server and in the client SDK.

__c. Namespaces that contain the section **server** or sometimes **service** are **server API** that is only available in the server.

__d. Namespaces that contain the section **internal** are **internal API** which direct usage is not supported.

__e. Namespaces that contain the section **rcp** are **Rich Client API** which are part of the SDK that provides UI and other Eclipse client related code and are not part of the supported API but make up for good code usage examples.

__f. Namespaces that contain the section **test** or **tests** are unit and other **test code** and are not part of the supported API but make up for good code usage examples.

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.

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 information on how to create more complex components, see: this wiki page.

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 'ITeamBuildRequestService' is not API (restriction on required library 'C:\RTC603Dev\installs\rtc-server-sdk\plugins\com.ibm.team.build.common_3.1.1400.v20160826_0053.jar')

You can ignore this warnings.

2.1 Create a basic server side service

__81. Start the RTC development server, if it is not already started.

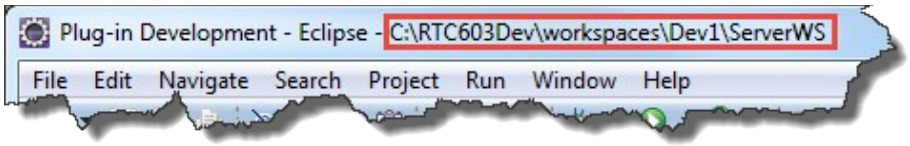
__a. Open a Windows Explorer and navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file.

__82. Return to the Eclipse client with the workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS` from the last lab.

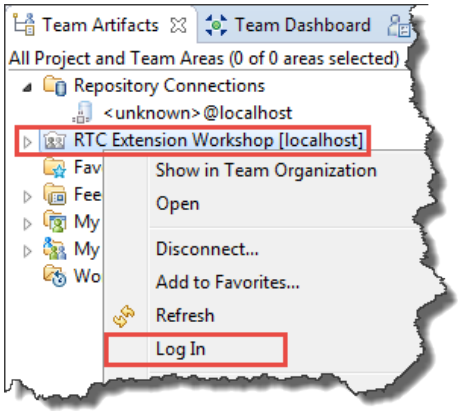
__a. If your RTC Server SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

__i. When prompted, select the Eclipse workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS`. Don't check the “Use as default” check box.

__b. Check the desired Eclipse workspace is in use.

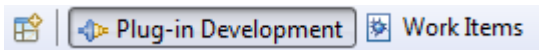


__c. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view in the **Work Items** perspective and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.



__83. Review the lab two code.

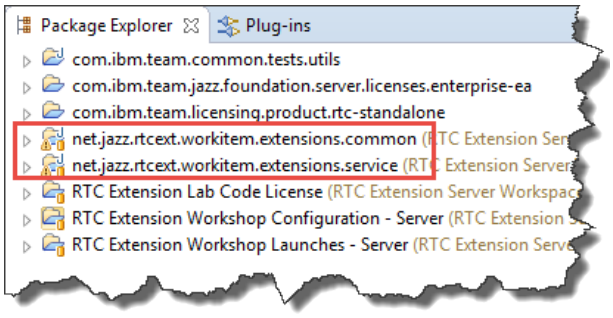
__a. In your RTC Eclipse client switch to the **Plug-in Development perspective**. In the toolbar toward the right, click **Plug-in Development** to switch the perspective.



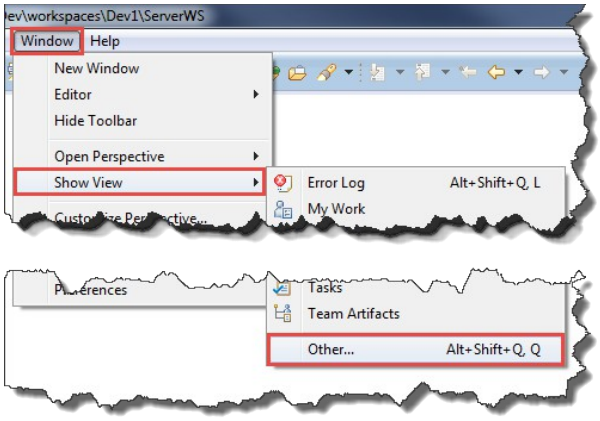
__i. If the **Plug-in Development** perspective is not available, open it now by selecting **Window > Open Perspective > Other… > Plug-in Development** from the menu bar.

__b. In lab one, you loaded the repository workspace `RTC Extension Server Workspace` into the Eclipse workspace `C:\RTC603Dev\workspaces\Dev1\ServerWS`. Along with the launches that you used in lab one, this also loaded the lab two code.

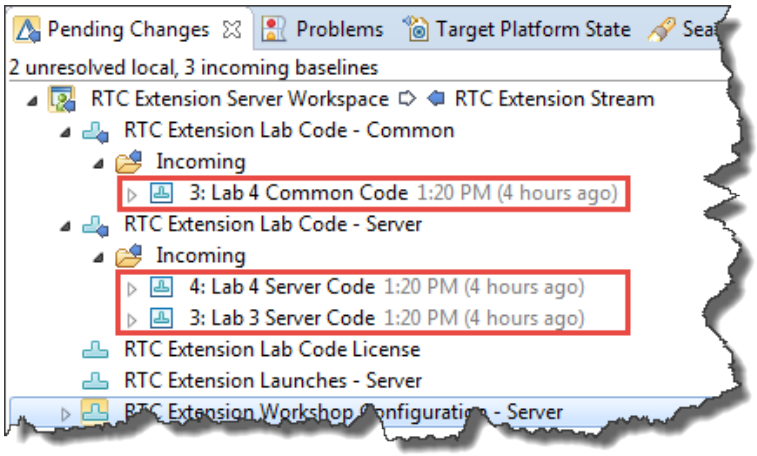
__c. Return to the **Package Explorer** view. Verify that the two projects that define the common (**net.jazz.rtcext.workitem.extensions.common**) and service (**net.jazz.rtcext.workitem.extensions.service**) parts of your component are present. In the rest of this lab you will learn about the various parts of this initially simple participant.



__d. Open the **Pending Changes** view using **Window>Show View>Other...** select Pending Changes.



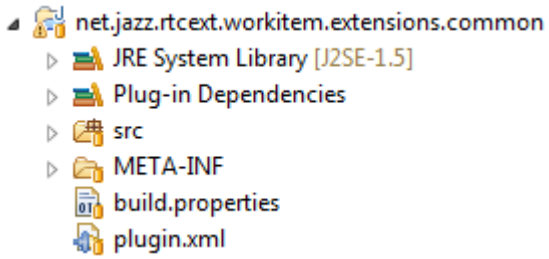
__e. You will also notice in the **Pending Changes** view, that there are incoming change sets and baselines. Do not accept them. You will make use of them in later labs. If the **Pending Changes** view is not open, select **Window > Show View > Other…** from the menubar, type `pending` into the filter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



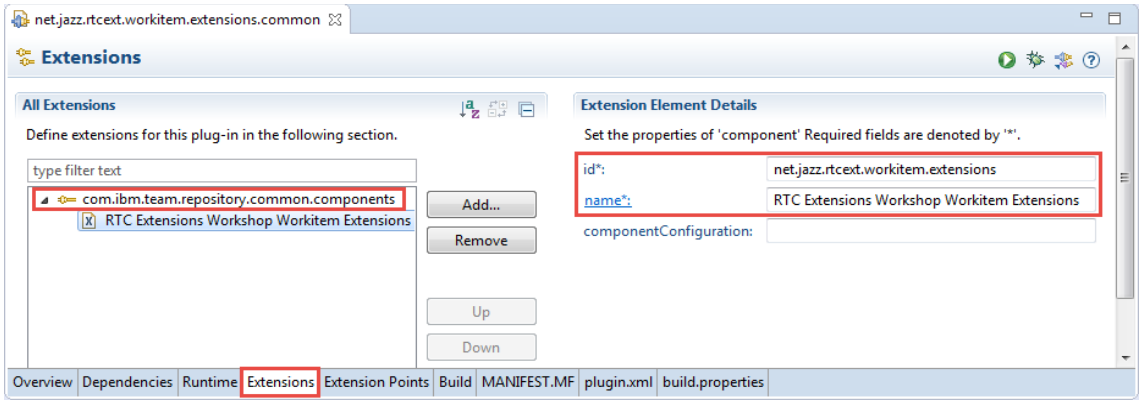
__84. Understanding the common plug-in Eclipse project.

__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. It is a best practice to separate a common project defining the component and basic definitions.

__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.

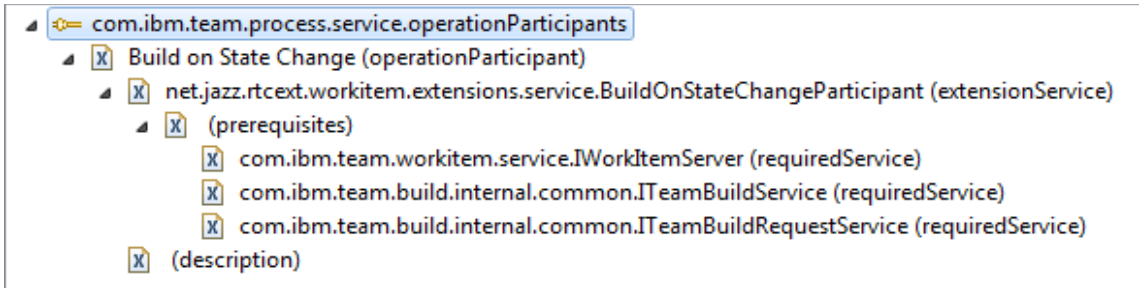
```
/**
 * The component id is used to identify the component to Jazz. It is also
 * used by service definitions to identify which component they belong to.
 */
public static String COMPONENT_ID ="net.jazz.rtcext.workitem.extensions";
```

__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.

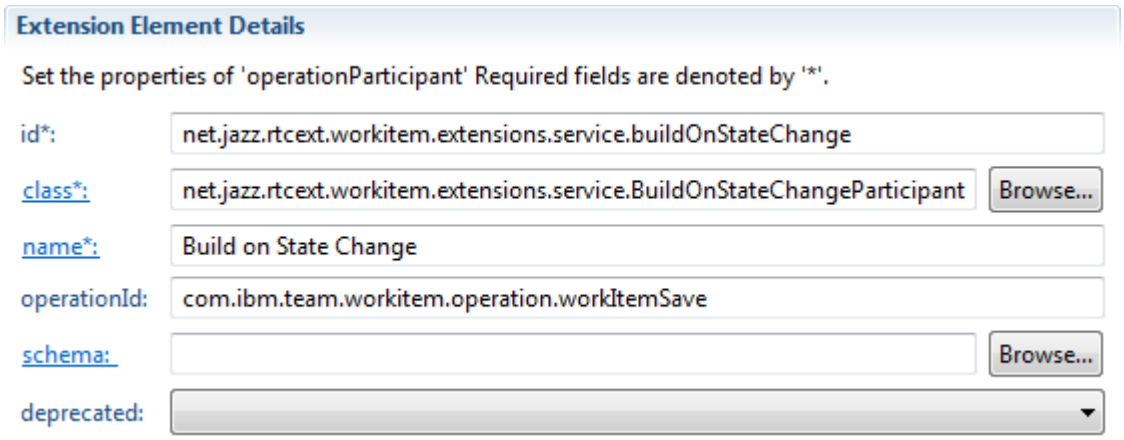
```
/**
 * The extension id is used to identify the operation participant to Jazz.
 * It is also included in instantiations of the participant in process
 * definitions.
 */
public static String EXTENSION_ID =
        "net.jazz.rtcext.workitem.extensions.service.buildOnStateChange";
```

__85. 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.



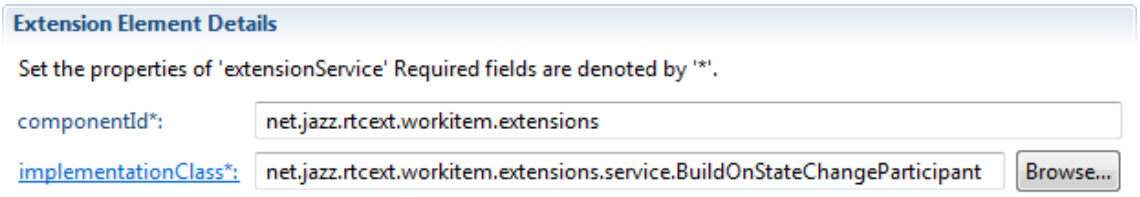
__b. Select the **Build on State Change (operationParticipant)** element on the left then the right side of the editor will look like this. The **class** and **operationId** attributes are the two most critical attributes. The class is the Java code that implements the service (more on that soon) and 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.



Available Extension points and Operation ID's

You can find the available Extension Points and Operation ID's here in the Jazz development wiki.

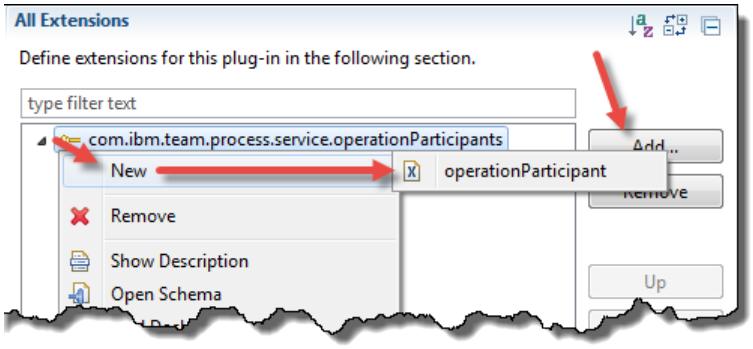
__c. Select the **net.jazz.rtcext.workitem.extensions.service.BuildOnStateChangeParticipant (extensionService)** element on the left and the right side of the editor will look like this. Note that this element is optional. It is only required if the participant will require services from other components.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

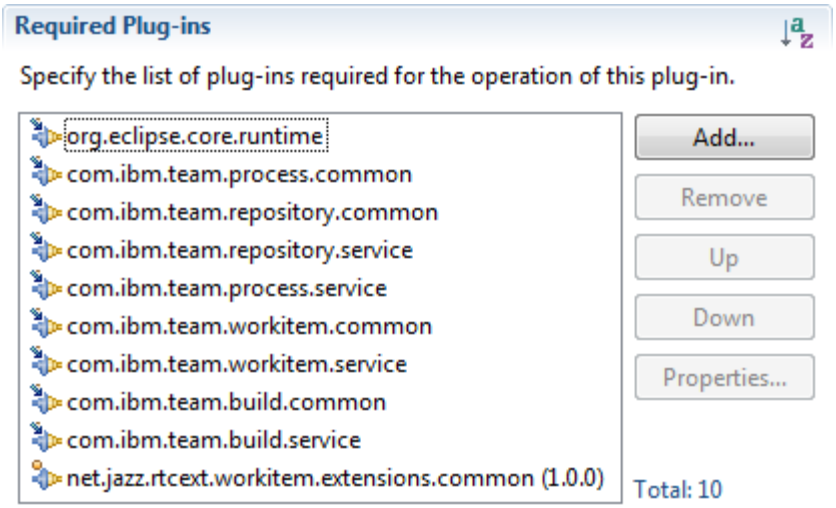
__f. Up to now, all the work you would do to create this definition is possible from this one place using the **Add…** button and the **New >** cascade menu from the various element's pop-up menus.



__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.



__86. 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 inter-dependencies 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))) {
```

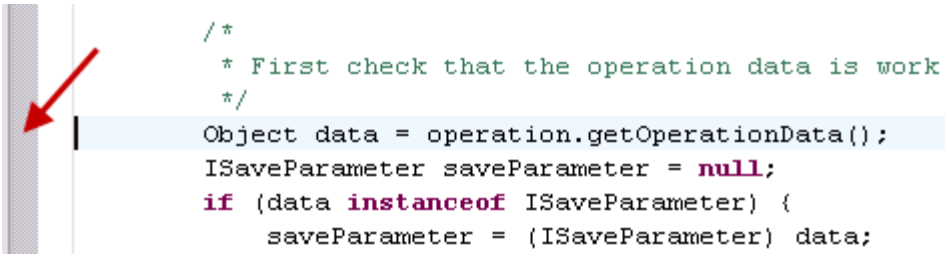
__iii. Is the work item of the type in which the participant is interested? Right now the work item type id is hard coded to the Story type from the Scrum template. You will change that later.

__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.

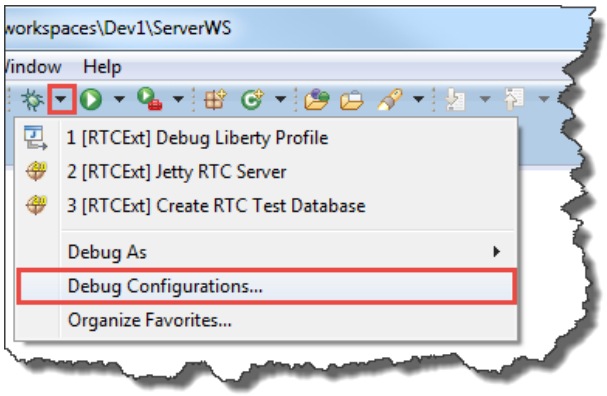
__h. So there you have a pretty simple participant that boils down to a few simple status checks in the run method and four lines of code to request a build. There is one more thing to do before leaving this editor. That is, **set a breakpoint** at the first line of the run method. You will step through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



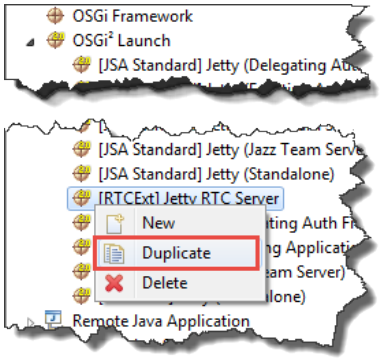
2.2 Launch the Jetty based RTC debug server

__87. Create the launch configuration.

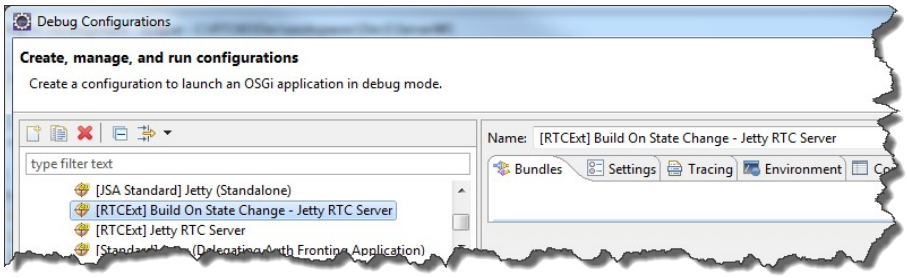
__a. From the **Debug** toolbar dropdown, select **Debug Configurations…**



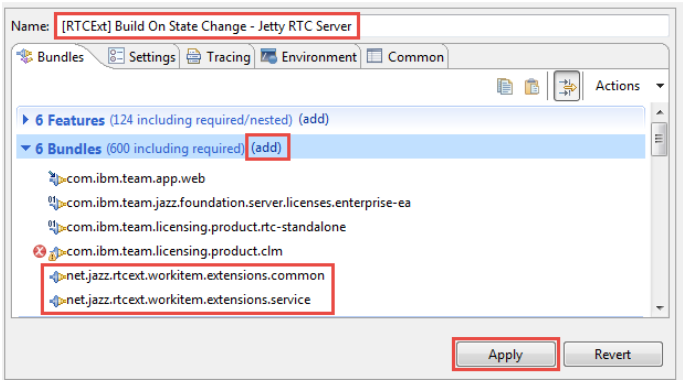
__b. In the **Debug Configurations** dialog, expand the **OSGi² Launch** tree and right click the **[RTCExt] Jetty RTC Server** configuration and then from the popup menu, select **Duplicate**. Note that you are not changing the existing launch but creating a copy of it. You should keep the original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



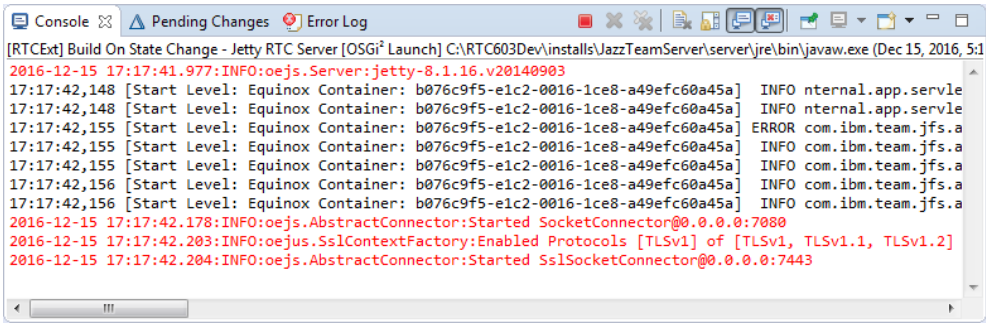
__i. You can ignore the error. The launch is set up to require one of two possible license files and only one of them will be available. As long as not more errors show up, the launch should work.

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

__88. Launch the Jetty based debug 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.

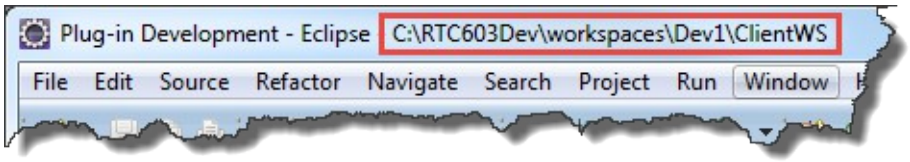
2.3 Launch an RTC client and connect to the server

__89. Return to the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS` from the last lab.

__a. If your RTC Client SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

__i. When prompted, select the Eclipse workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS`. Don't check the "Use as default" check box.

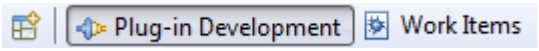
__b. Check the desired Eclipse workspace is used.



__c. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view in the **Work Items** perspective and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.

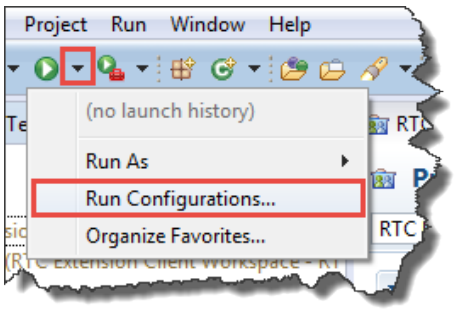
__90. Launch the RTC test client.

__a. In your RTC Eclipse client switch to the **Plug-in Development perspective**. In the toolbar toward the right, click **Plug-in Development** to switch the perspective.



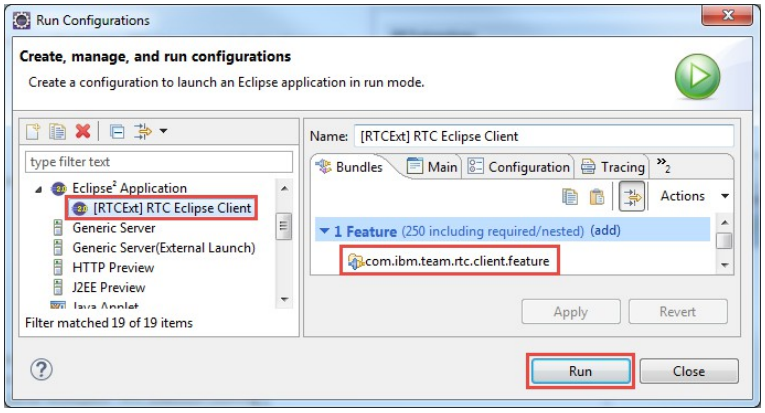
__i. If the **Plug-in Development** perspective is not available, open it now by selecting **Window > Open Perspective > Other… > Plug-in Development** from the menu bar.

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



__c. In the **Run Configurations** dialog, select **Eclipse² Application > [RTCExt] RTC Eclipse Client** and then click **Run**. Note that on this **Bundles** tab, that the feature com.ibm.team.rtc.client.feature you imported earlier (the RTC client feature) is included in this launch.

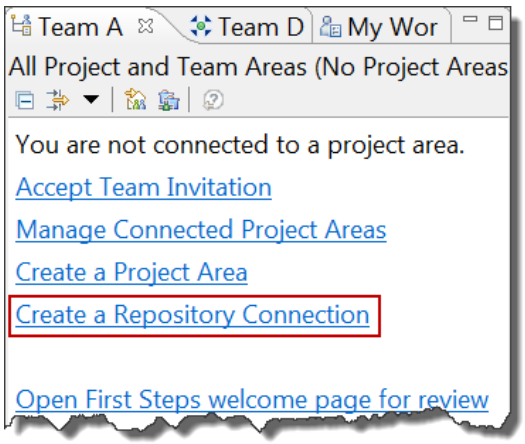
__d. The RTC Eclipse client will start up and should look familiar. If you are prompted to clear the runtime workspace, click **Yes** (you will usually click **No**, but this time start fresh).



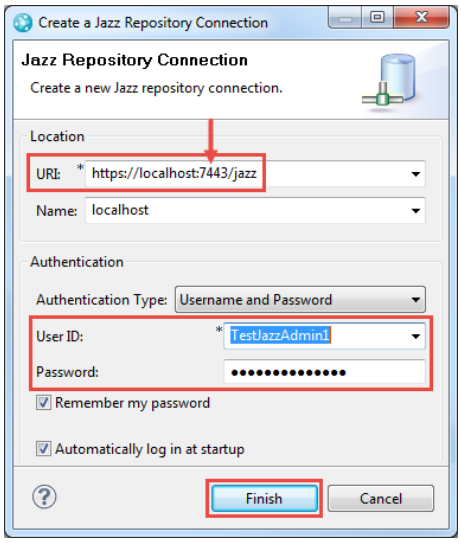
__e. Minimize the **Welcome** screen via this () button near the top or right of the window.

__91. Connect to the Jetty based RTC 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.

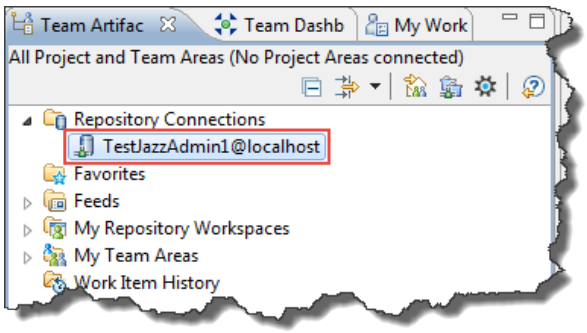


__b. In the **Create a Jazz Repository Connection** wizard, set the **URI** to https://localhost:7443/jazz and the **User ID** and **Password** fields to `TestJazzAdmin1`. Note that it is a '7' and not a '9' in the URI. Then, click **Finish**. Note that "\jazz" is the correct context root and not "\ccm". 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 RTC development server on WAS Liberty with split applications for final testing (a later lab).



__i. If asked for a password for secure storage you might not be able to provide this in some versions. Try providing a password. Using **Cancel** always works but does not allow to store the password and forces to enter the password for each log in.

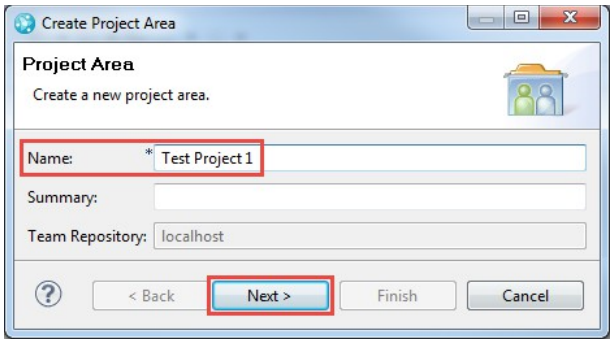
__c. You will now have a repository connection in your **Team Artifacts** view.



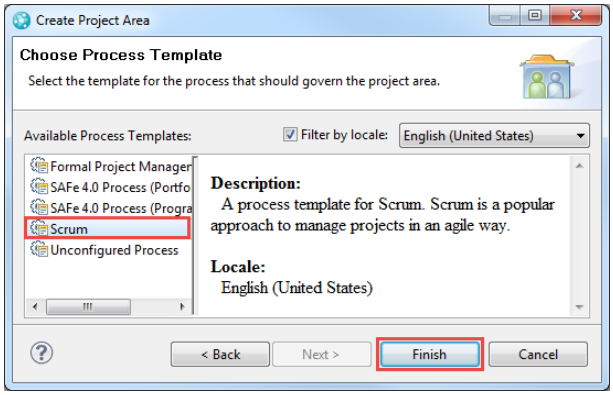
2.4 Edit the Process to Use the Participant

__92. 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**.

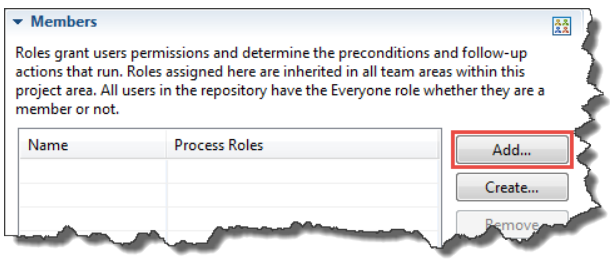


__b. On the second page of the wizard, click the **Deploy Templates** button. This operation may take a bit of time. When it completes, you will be on the next page of the wizard. Select **Scrum** on the left and then click **Finish**. When the operation completes and the project area editor opens, leave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

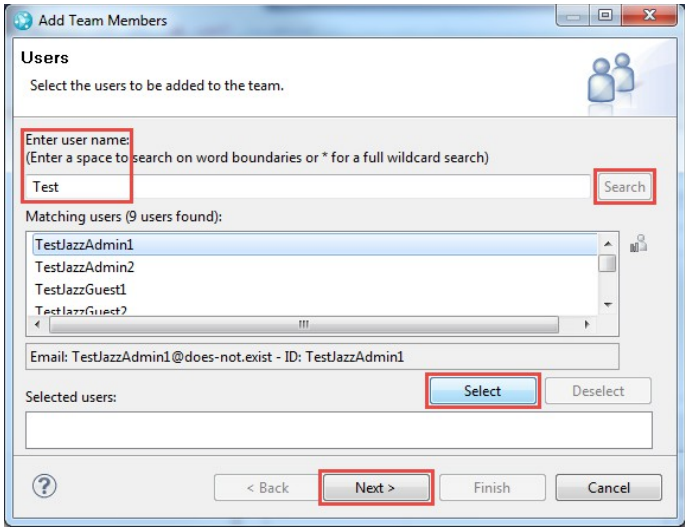


__93. Add `TestJazzAdmin1` as a member of the project area.

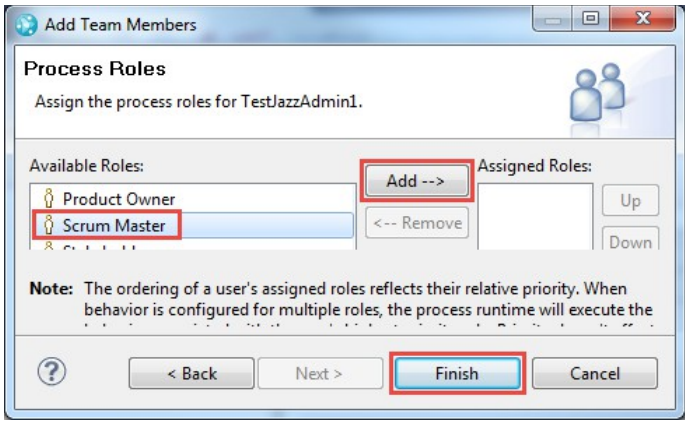
__a. On the **Overview** tab of the project area editor, expand the **Members** section and click **Add…**



__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**.



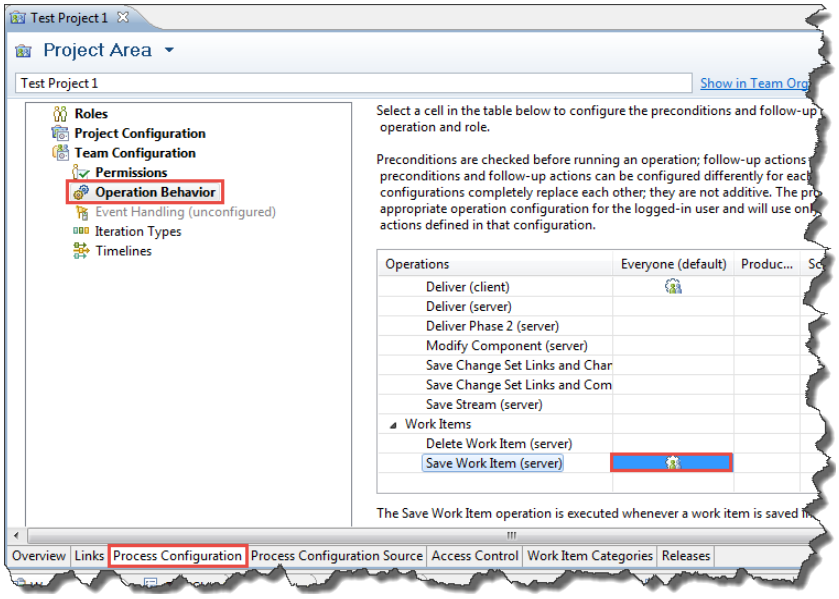
__c. On the second page of the wizard, select **Scrum Master** on the left, click **Add -->** (moves the selection to the right) and then click **Finish**.



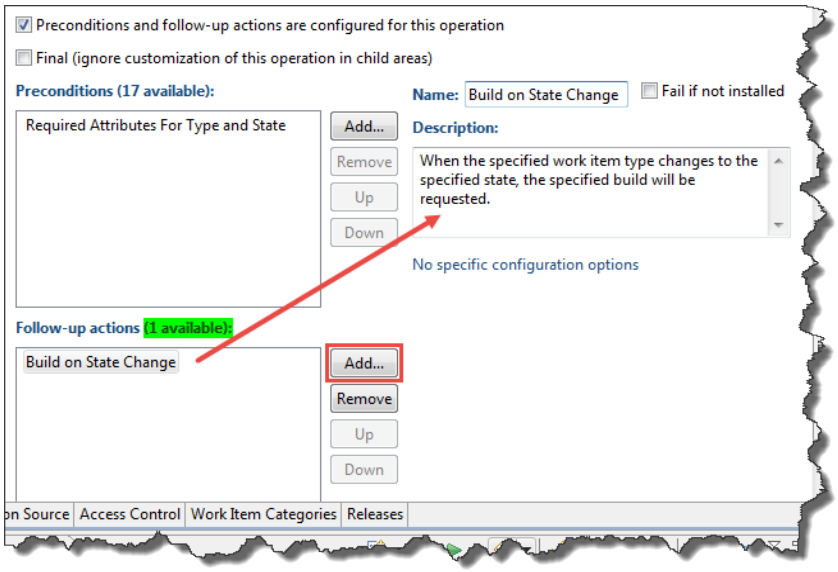
__d. Back on the project area editor's **Overview** page, click **Save** (at the upper right) but leave the editor open for the next step.

__94. 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 corner of the editor.



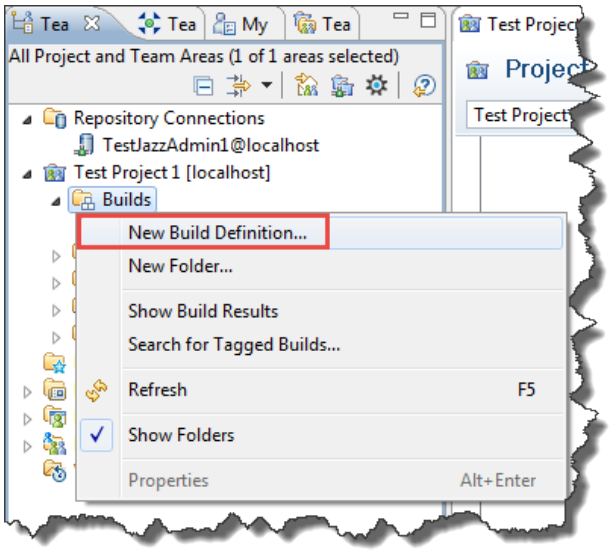
__c. Make sure you have saved your changes, otherwise the next steps will fail.

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

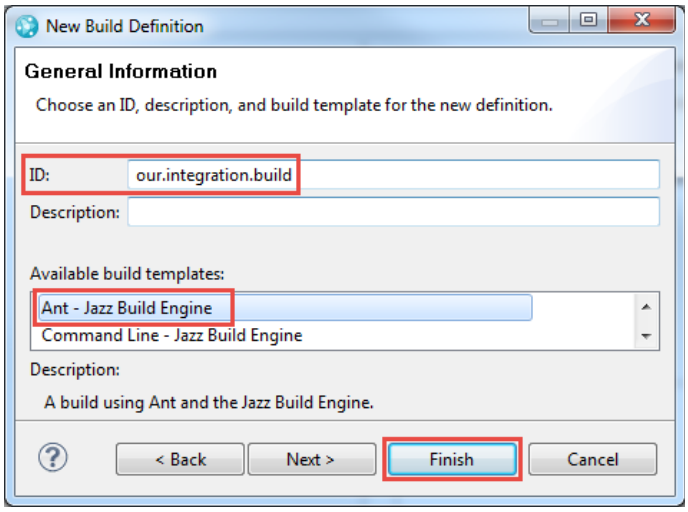
2.5 Trigger the Participant

__95. 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.

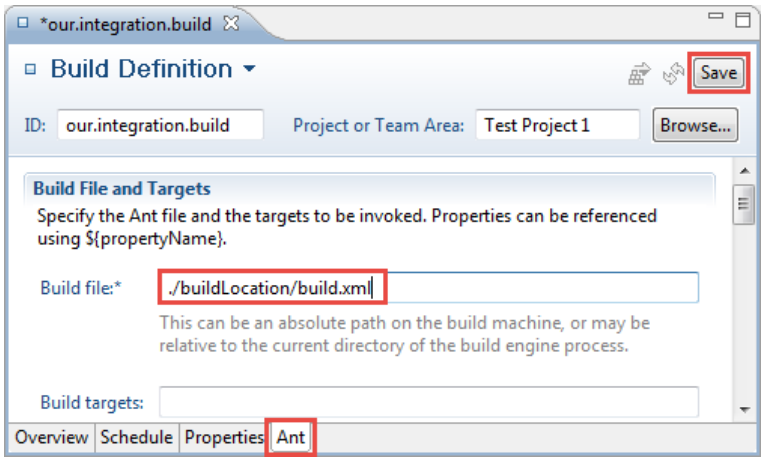
__a. In the **Team Artifacts** view, expand the **Test Project 1** node, right click **Builds** and then click **New Build Definition…**



__b. In the **New Build Definition** wizard, make sure **Create a new build** is selected, 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**.

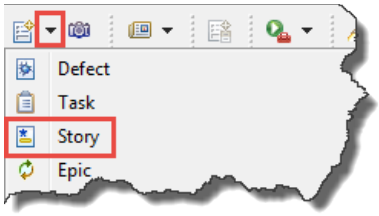


__c. In the build definition editor that opens, switch to the **Ant** tab, and enter a path for the **Build** file and then click **Save**. You may now close the editor. Note that the build file does not exist and any path will work for the current purpose. If you wish, you can use the **Build file**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

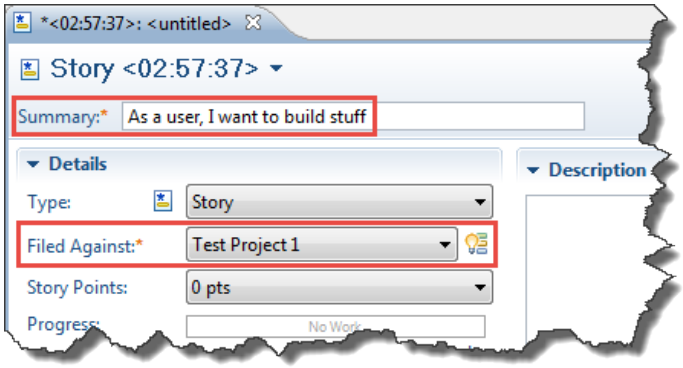


__96. Create a Story work item.

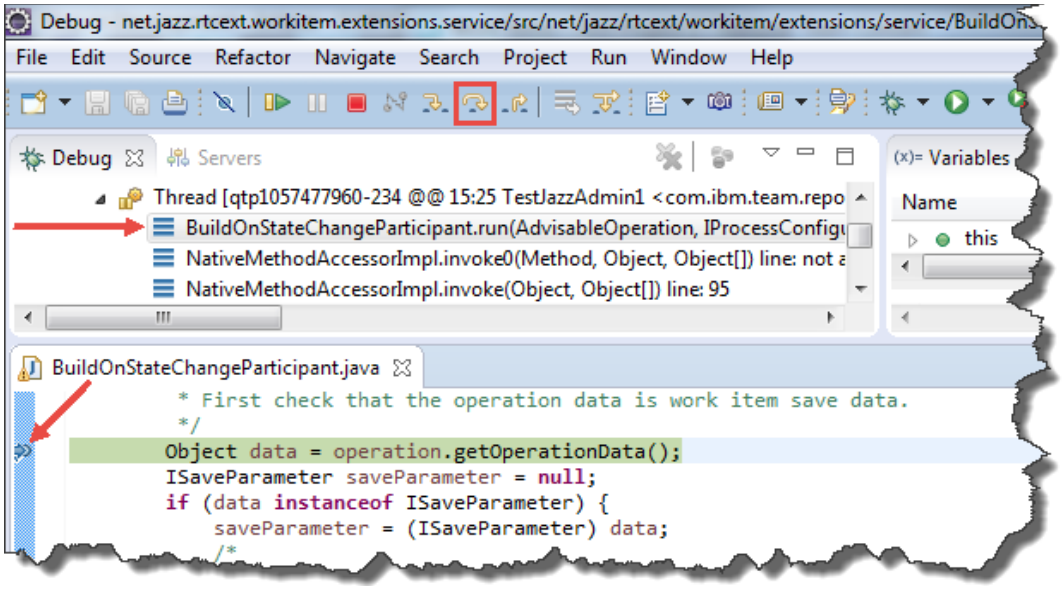
__a. Click the dropdown menu arrow next to the **New Work Item** toolbar icon and then click **Story**.



__b. In the new work item editor that opens, set the required fields as shown here. Set Summary to '`As a user, I want to build stuff`' and Filed Against to **Test Project1** and then click **Save** in the upper right corner.



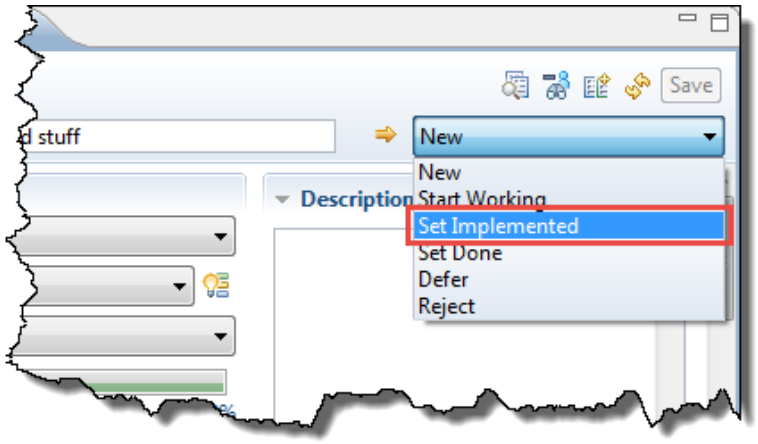
__c. The breakpoint you set earlier is now hit. The RTC Eclipse client in which you were studying the code will now surface (if asked about switching to the debug perspective, click **Yes**). If it does not surface, you probably minimized it earlier. In this case, it will be flashing in the Windows taskbar.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

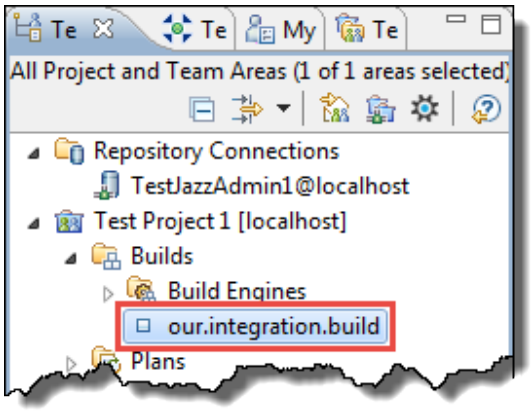
__97. Move the Story to the Implemented state.

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

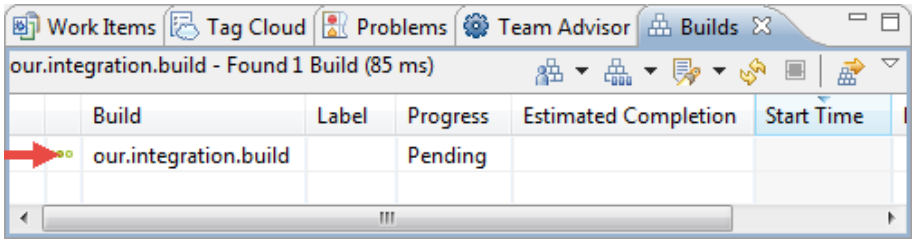


__b. Once again the breakpoint is hit and your debugger surfaces (or you need to click it in the Windows taskbar). Step through the code again. When you get to the call to the build method, use the Step Into button ( ). You can then step through the four lines that request the build and then click the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



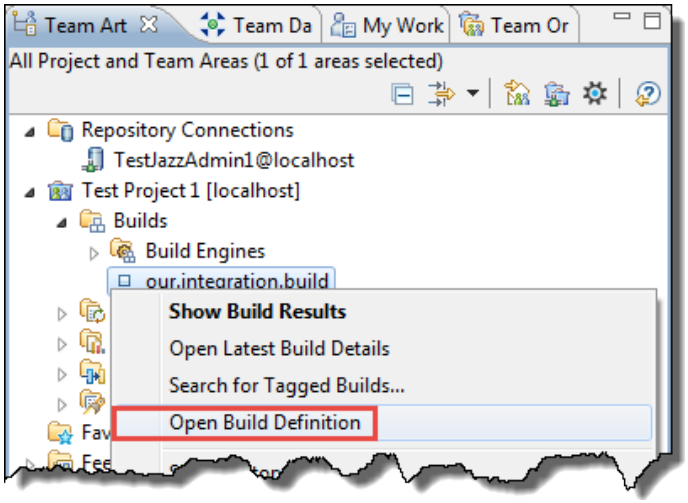
__d. The **Builds** view opens showing the build request the participant just submitted.



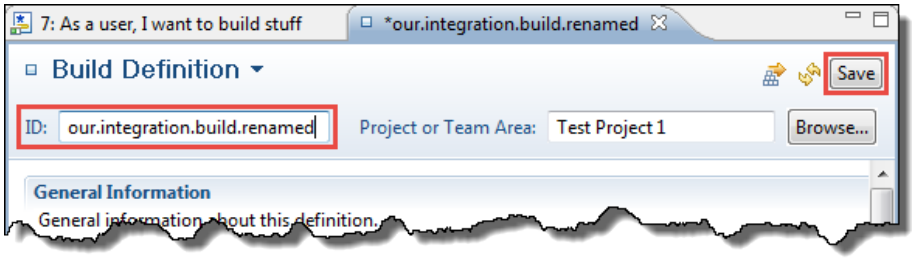
2.6 Rename Build Definition and Try Again

__98. Rename the build definition.

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

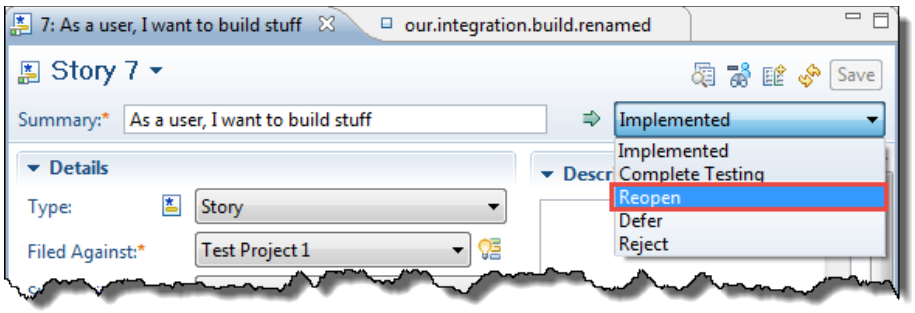


__b. In the build definition editor change the **ID** to `our.integration.build.renamed` and then click **Save**. Do not close the editor as you will want to rename it back soon.

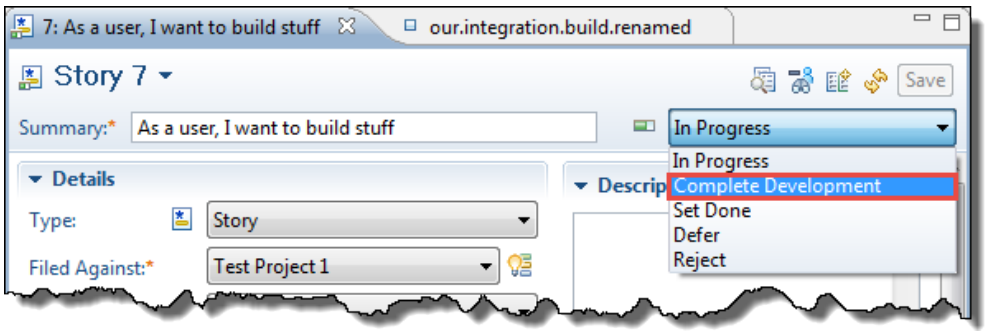


__99. Move the story to the Implemented state again.

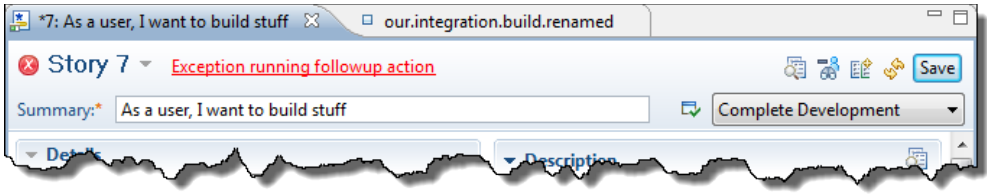
__a. Switch back to the work item editor and select **Reopen** from the state dropdown and then click Save. When the debugger surfaces, just click the resume button ( ). You are not to the interesting bit yet.



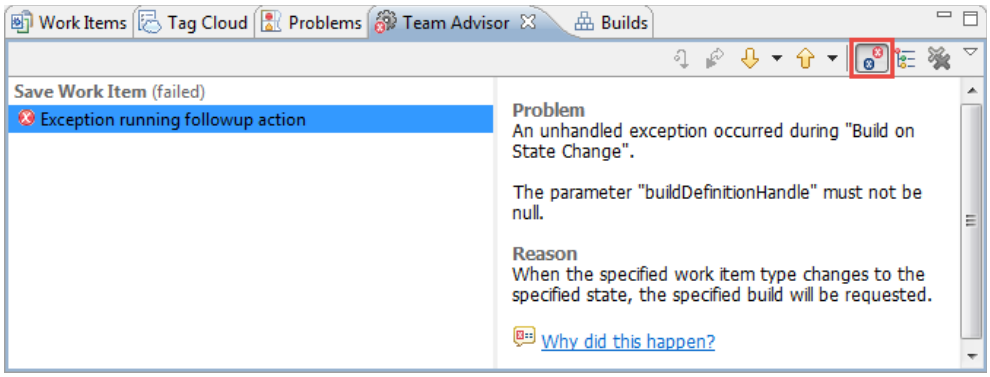
__b. Again in the work item editor, select **Complete Development** from the same dropdown and click **Save** again.



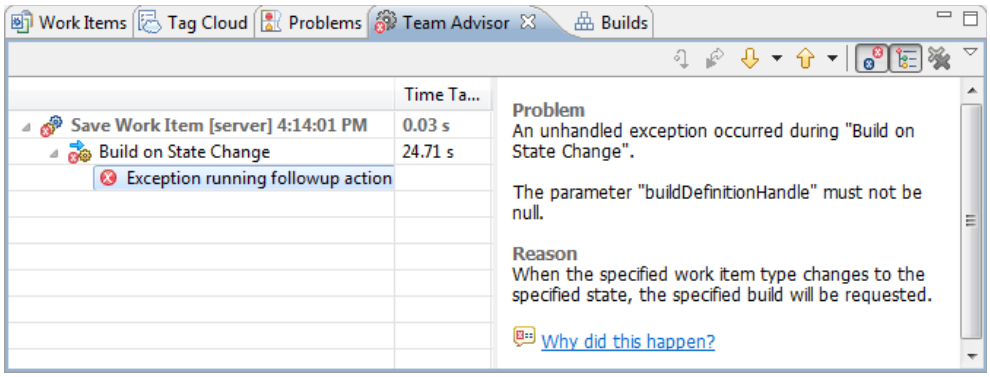
__c. This time, when the debugger surfaces, use the step over button to get to the build method call and then use the step into button. Step through the build method and note the major difference this time. The call to get the build definition returns null and the request of the build throws an exception.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



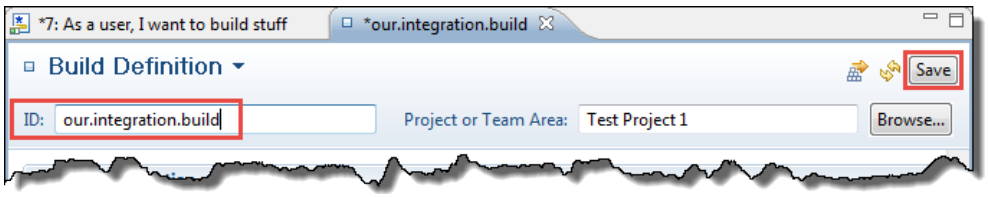
__d. The **Team Advisor** view appears with more information on the error. Click the **Show Detail Tree** button.



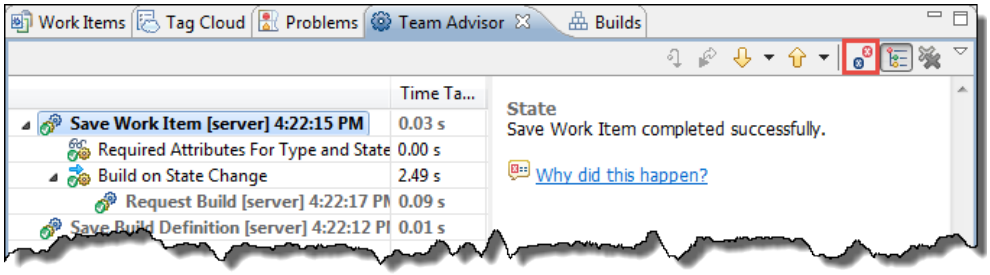
__e. The left side of the view changes to show the structure of the error condition. Click the nodes on the left to see what information is available. It is clear that better information would be helpful. For example, a messages stating that the participant was looking for a particular build definition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

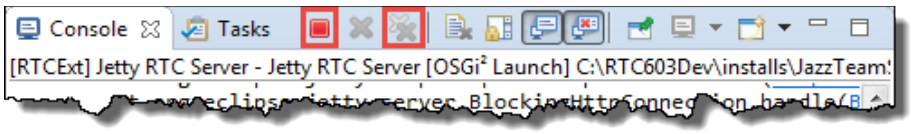


__100. Close down the launched RTC client and Jetty debug server.

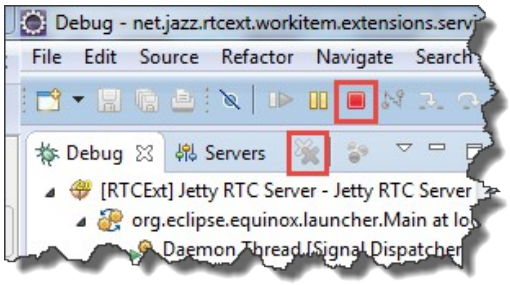
__a. Close the launched RTC Eclipse client where you were working with the Story and build definition (logged in as `TestJazzAdmin1`).

__b. Return to the Eclipse client with the workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS` used to launch the Jetty debug server.

__c. Terminate the Jetty debug server. Go to the **Console** view and click the **Remove All Terminated Launches** icon and then click the **Terminate** icon.



The same options are available around the debug tab as well. Use the **Remove All Terminated Launches** icon and make sure no launch is displayed.



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.

3.1 Understanding Error Handling code

__101. Start the RTC development server, if it is not already started.

__a. Open a Windows Explorer and navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file.

__102. Return to the Eclipse client with the workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS` from the last lab.

__a. If your RTC Server SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

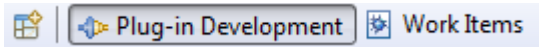
__i. When prompted, select the Eclipse workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS`. Don't check the "Use as default" check box.

__b. Check the desired Eclipse workspace is in use.

__c. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view in the **Work Items** perspective and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.

__103. Browse and load the Lab 3 server extension code.

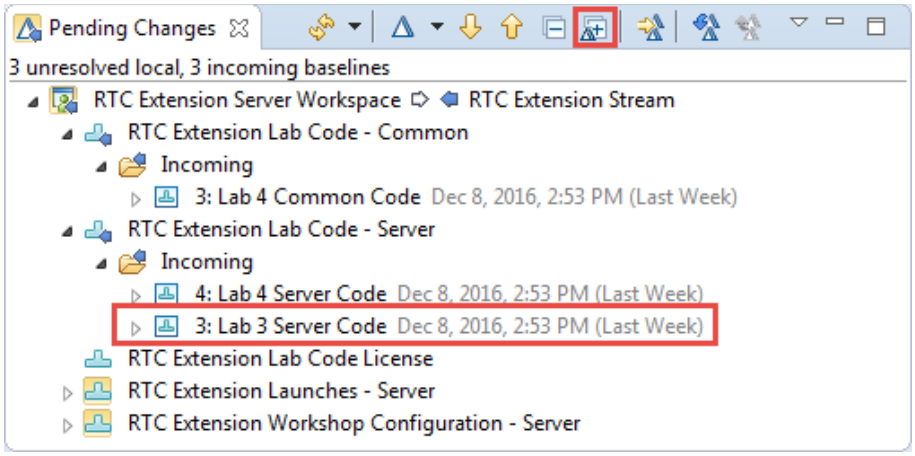
__a. Switch to the **Plug-in Development perspective**. In the toolbar toward the right, click **Plug-in Development** to switch the perspective.



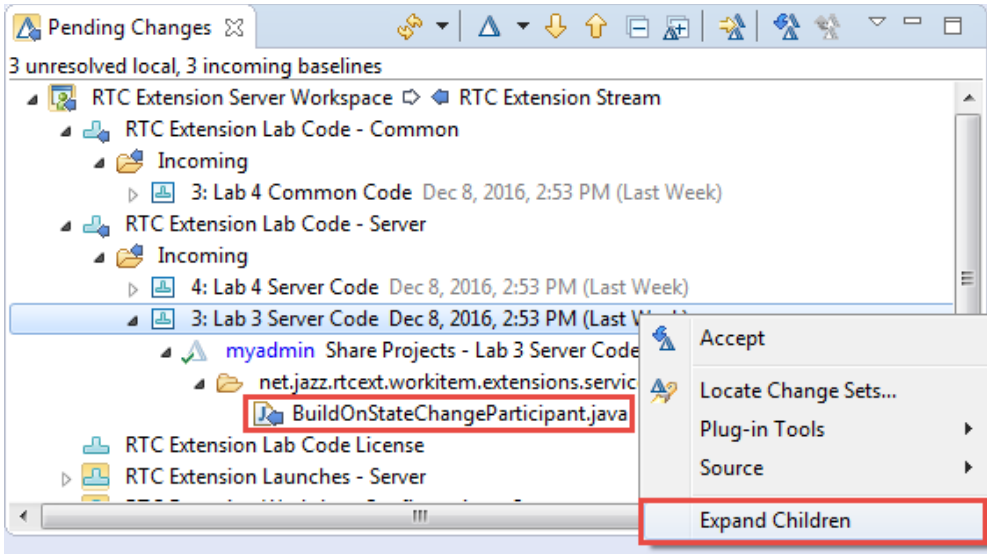
__a. If the **Plug-in Development** perspective is not available, open it now by selecting **Window > Open Perspective > Other… > Plug-in Development** from the menu bar.

__b. Open the **Pending Changes** view, if it is not yet available in the perspective use **Window>Show View>Other...** and select Pending Changes.

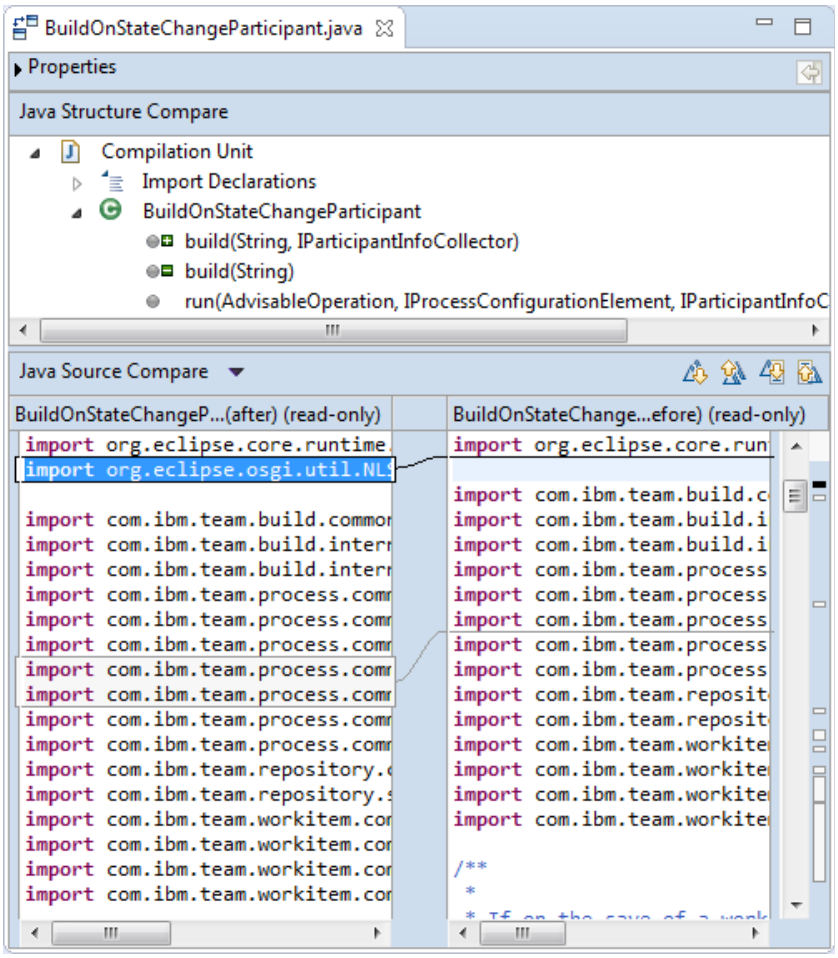
__c. In the **Pending Changes** view, click the **Expand to Change sets** icon. This will show 3 incoming baselines as shown here.



__d. Right click the **Lab 3 Server Code** baseline under the component **RTC Extension Lab Code - Server in** the **RTC Extension Server 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.



__e. 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.



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

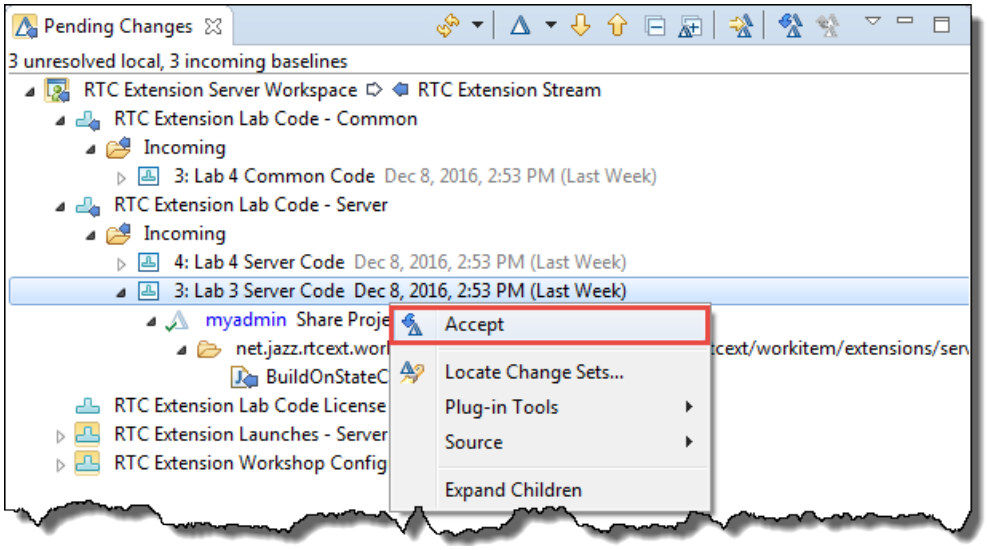
__i. The collector parameter to the run method is now passed through to the build method where it will be used.

__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.

__g. Close the comparison editor and then in the **Pending Changes** view, right click the **Lab 3 Code** baseline under the component **RTC Extension Lab Code – Server** in the **RTC Extension Server 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.



__104. 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.

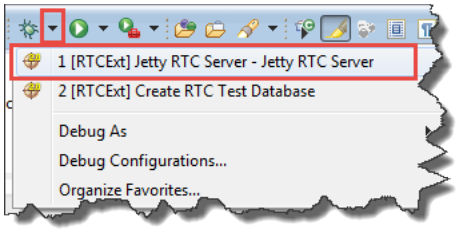
- Finally, the report info item is added to the collector. Note that this is not done automatically by the createInfo factory method.

```
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);
```

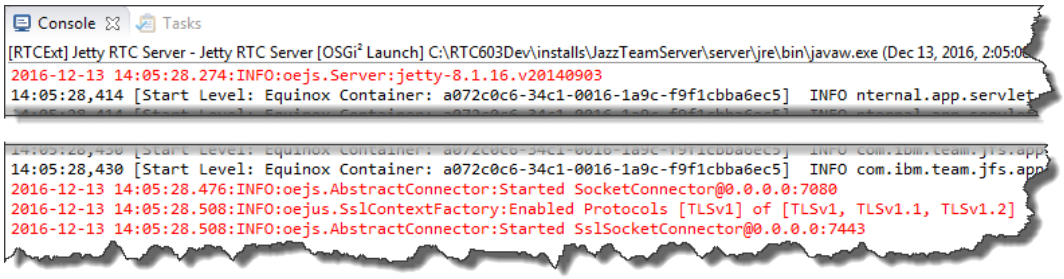
3.2 Launch the Jetty RTC debug server

__105. Use the existing Jetty server launch configuration from lab 2.

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



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



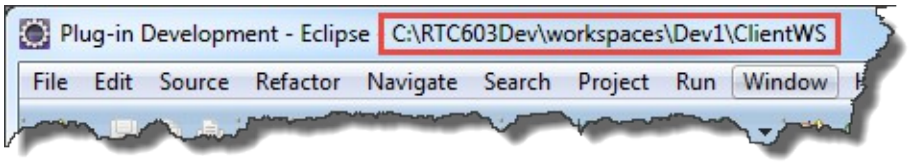
3.3 Launch an RTC test client and connect to the Jetty debug server

__106. Return to the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS` from the last lab.

__a. If your RTC Client SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

__i. When prompted, select the Eclipse workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS`. Don't check the "Use as default" check box.

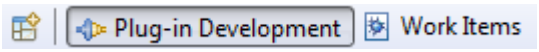
__b. Check the desired Eclipse workspace is used.



__c. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view in the **Work Items** perspective and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.

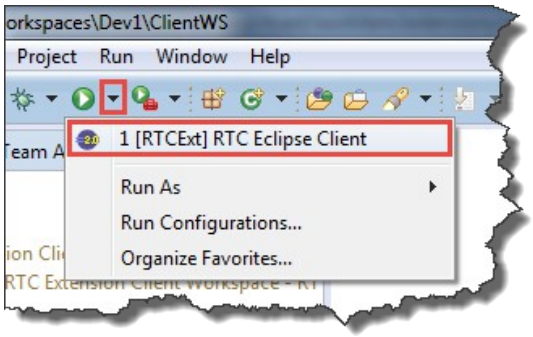
__107. Launch the RTC Eclipse test client from the RTC Client SDK development workspace.

__a. Switch to the **Plug-in Development perspective**. In the toolbar toward the right, click **Plug-in Development** to switch the perspective.

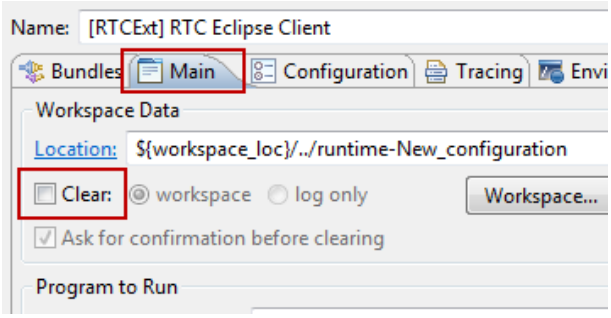


__i. If the **Plug-in Development** perspective is not available, open it now by selecting **Window > Open Perspective > Other… > Plug-in Development** from the menu bar.

__b. 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.



__c. If you are prompted to clear the runtime workspace **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 and removing the option **Clear** in the **Main** tab.



__d. The RTC Eclipse client will start up and will connect automatically to the Jetty debugserver you just launched via the repository connection you created in lab 2.

__e. You might have to provide the password again. If necessary provide the password `TestJazzAdmin1` for the user `TestJazzAdmin1`.

__i. If asked for a password for secure storage you might not be able to provide this in some versions. Try providing a password. Using **Cancel** always works but does not allow to store the password and forces to enter the password for each log in.

__f. 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

__108. 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.

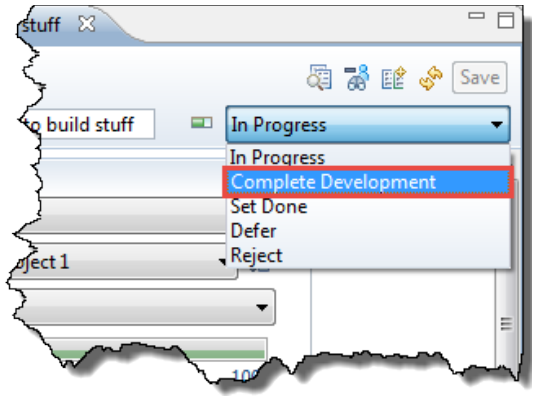
__a. Either of these will cause the breakpoint you set earlier to trigger. If it does not trigger, check if the breakpoint is set to the correct line of code. If necessary remove the old break points and add a valid one. And change the state of the story back. The RTC Eclipse client in which you were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

__109. Move the Story to the Implemented state.

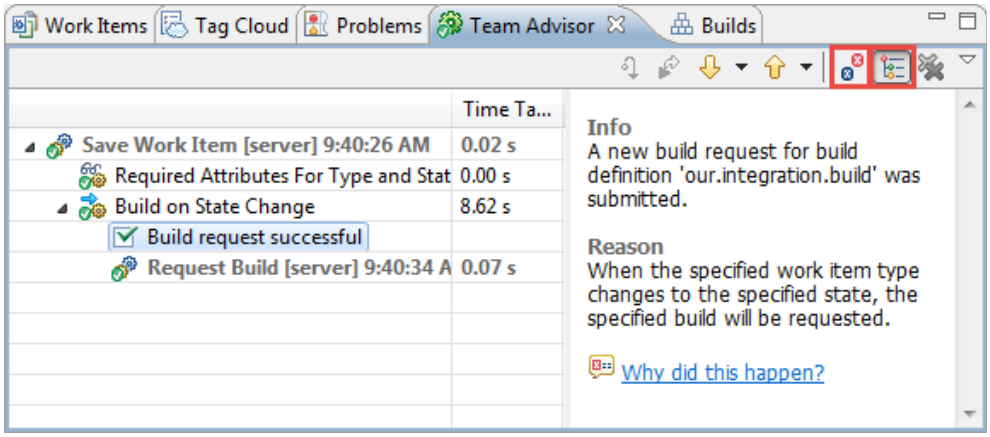
__a. At the upper right portion of the work item editor, select **Set Implemented** or **Complete Development** (depends on which workflow state the story is currently in) and then click **Save**.



__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 ( ).

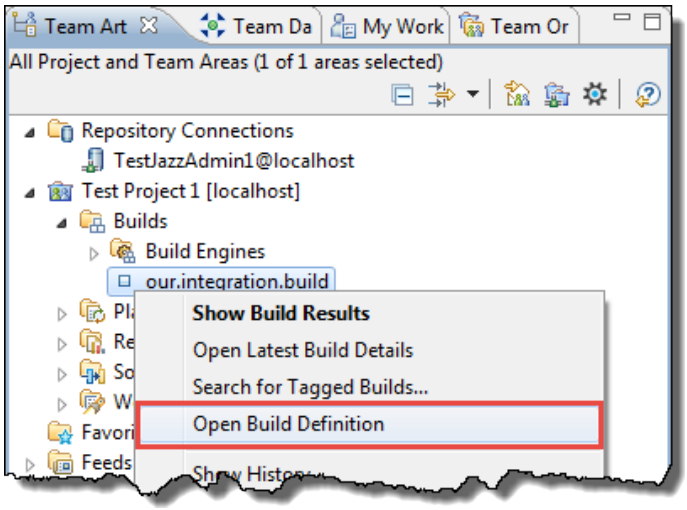
__d. If you go to the **Team Advisor** view and check to make sure the **Show Failures Only** filter is off and **Show Detail Tree** is on (see highlight below), you can browse the results of this successful operation. Also, if you double click **our.integration.build** in the **Team Artifacts** view,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



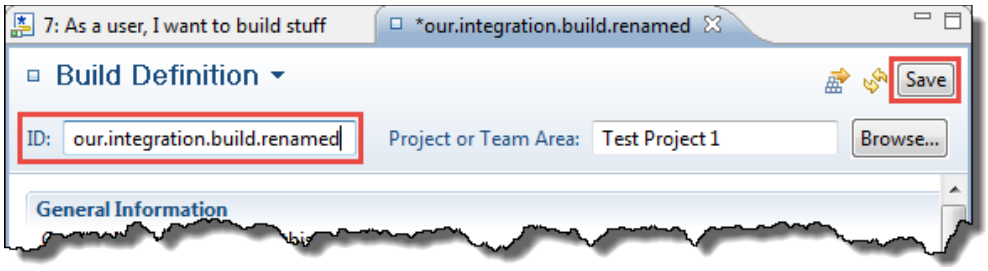
3.5 Rename Build Definition and Try Again

__110. Rename the build definition.

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

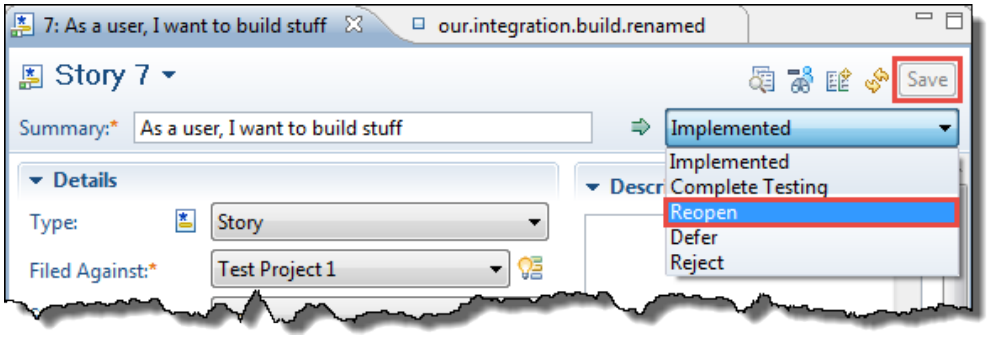


__b. In the build definition editor change the **ID** to `our.integration.build.renamed` and then click **Save**. Do not close the editor as you will want to rename it back soon.

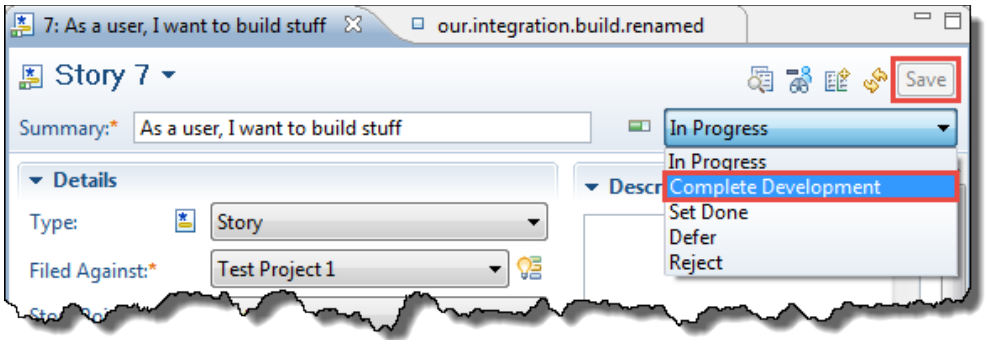


__111. Move the story to the Implemented state again.

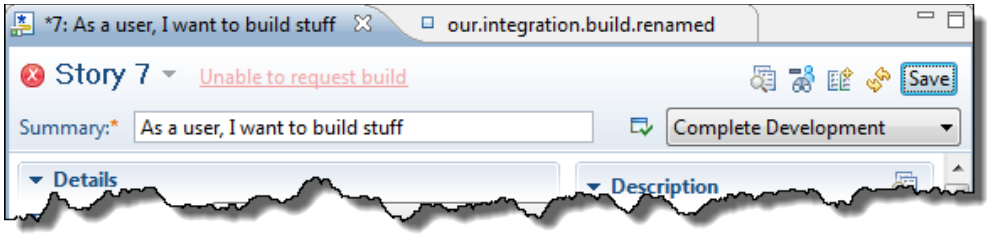
__a. Switch back to the work item editor and select **Reopen** from the state dropdown and then click **Save**. When the debugger surfaces, just click the resume button ( ). You are not to the interesting bit yet.



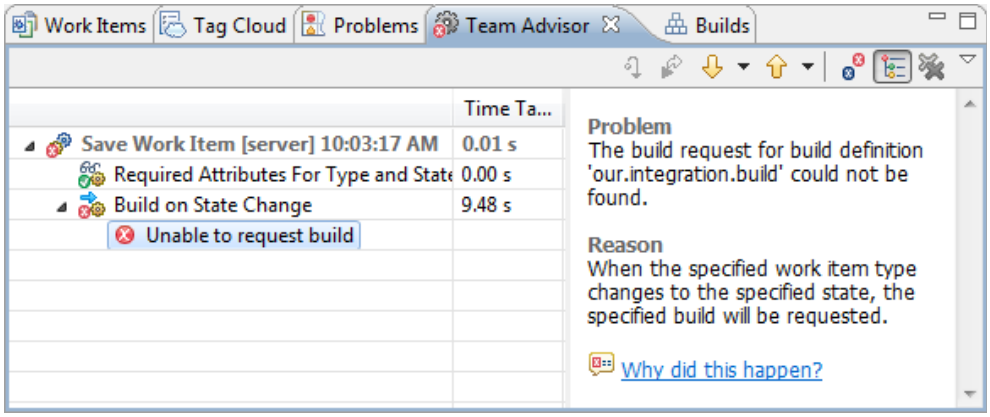
__b. Again in the work item editor, select **Complete Development** from the same dropdown and click **Save** again.



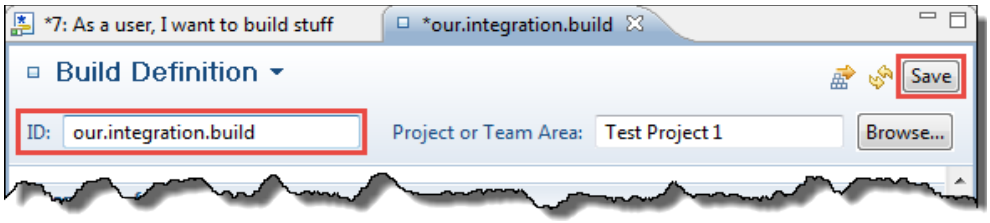
__c. This time, when the debugger surfaces, use the step over button to get to the build method call and then use the step into button. Step through the build method and note the major difference this time. The call to get the build definition returns null, but this time a null pointer exception is not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

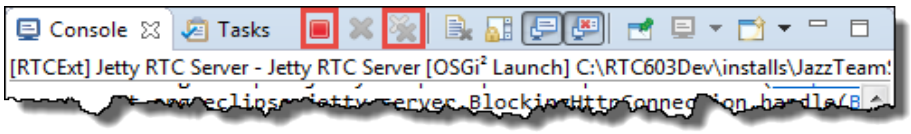


__f. Switch back to the work item editor and click **Save**. When the debugger surfaces, you can step into the build method again or just hit resume. Once you do resume, the work item save should complete okay. Return to the work item editor to confirm this. If you go to the **Team Advisor** view and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

__112. Close down the launched client and server.

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

__b. Return to the Eclipse client with the workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS` used to launch the Jetty server. Terminate the Jetty Server. Go to the **Console** view and click the **Remove All Terminated Launches** icon and then 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. Use the **Remove All Terminated Launches** icon again and make sure all launches are removed.

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 Parametrization

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!

4.1 Understanding Parametrization

__113. Start the RTC development server, if it is not already started.

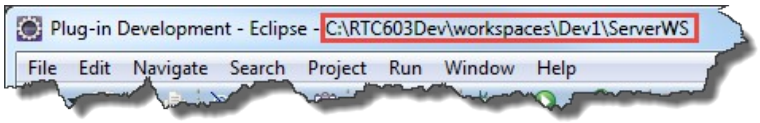
__a. Open a Windows Explorer and navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file.

__114. Return to the Eclipse client with the workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS`.

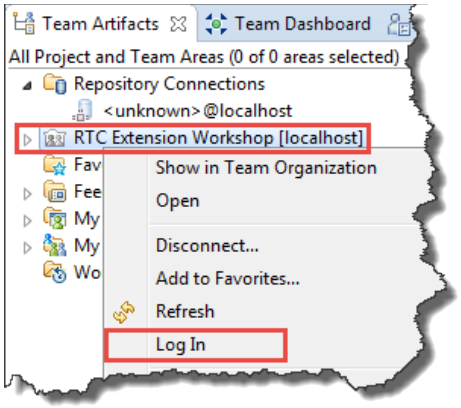
__a. If your RTC Server SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

__i. When prompted, select the Eclipse workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS`. Don't check the “Use as default” check box. Click OK.

__b. Check the desired Eclipse workspace is in use.

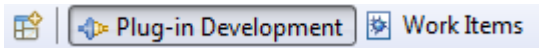


__c. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view in the **Work Items** perspective and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.



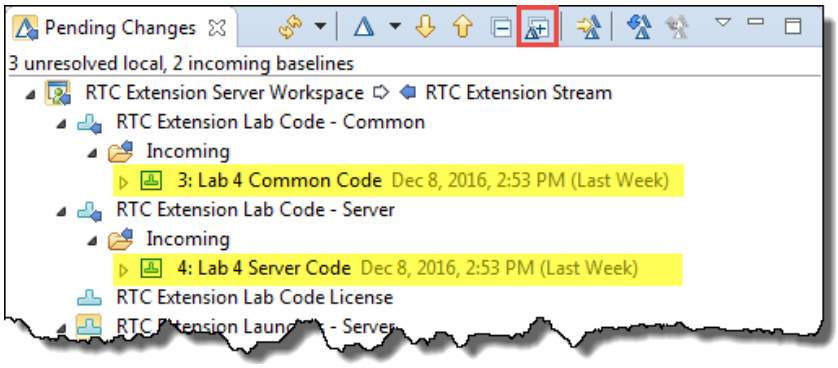
__115. Browse and load the Lab 4 server extension code.

__a. Switch to the **Plug-in Development perspective**. In the toolbar toward the right, click **Plug-in Development** to switch the perspective.

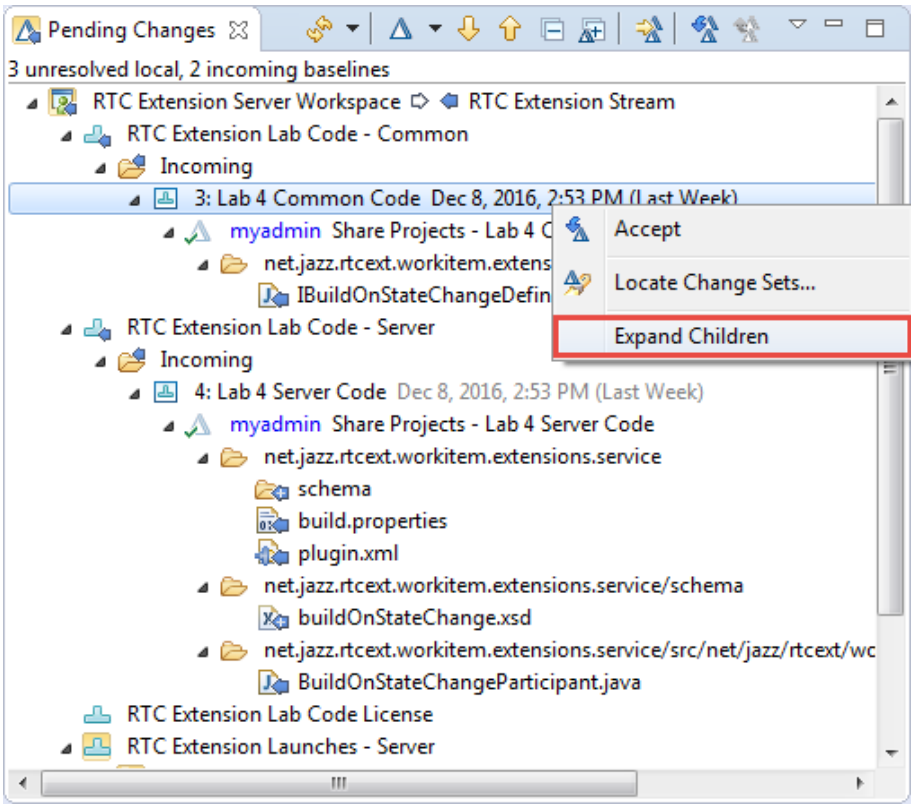


__a. If the **Plug-in Development** perspective is not available, open it now by selecting **Window > Open Perspective > Other… > Plug-in Development** from the menu bar.

__b. in the **Pending Changes** view, click the **Expand to Change sets** icon. This will show 2 incoming baselines in two components as shown here.

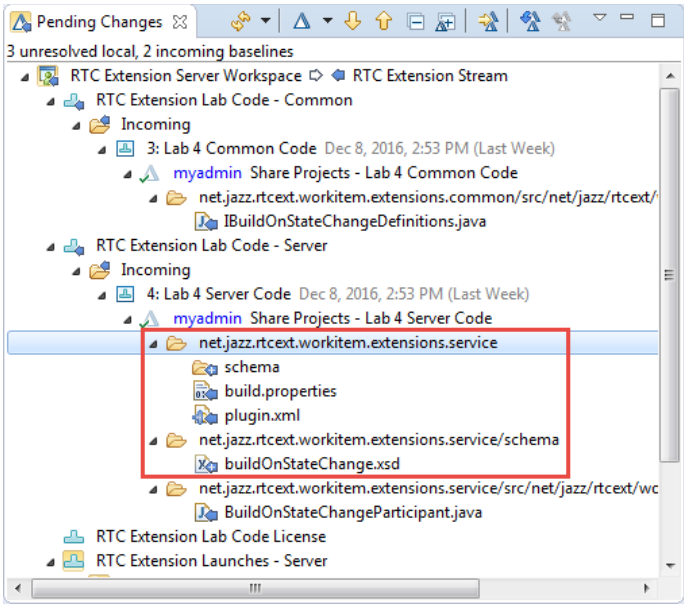


__c. Right click the **Lab 4 Common Code** baseline under the component **RTC Extension Lab Code – Common** in the **RTC Extension Server Workspace** node, and then click the **Expand Children** action. This will reveal all the changes made for lab 4 to this component. Repeat the same for the the **Lab 4 Server Code** baseline for the component RTC Extension Lab Code - Server, and then click the **Expand Children** action. This will reveal all the changes made for lab 4 to this component. As you can see there are quite a few more changes in this lab for several components.

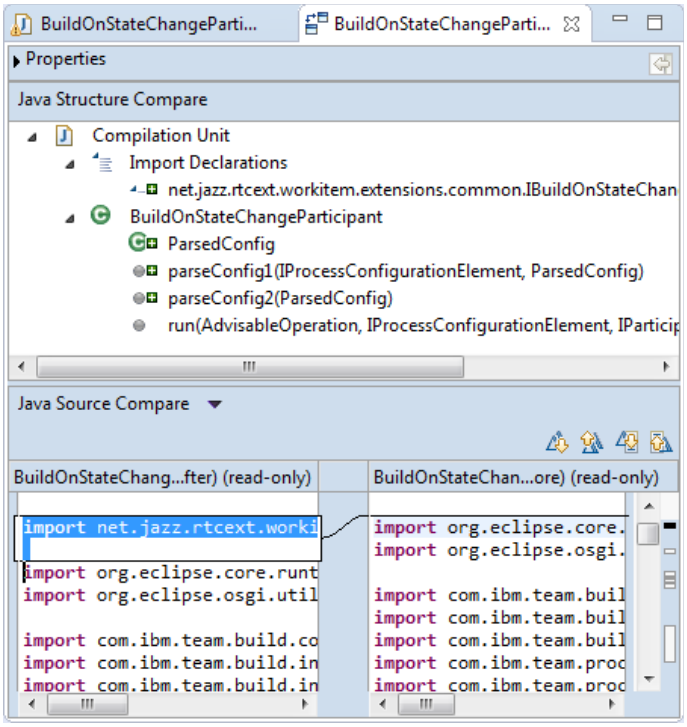


__d. Double click the first changed file, **IBuildOnStateChangeDefinitions.java** to open a comparison editor. You may want to double click the tab of the opened editor to maximize it. A set of constants have been added to this file. Most of them define elements of the XML schema that will be used to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

__e. 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.



__f. 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.



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

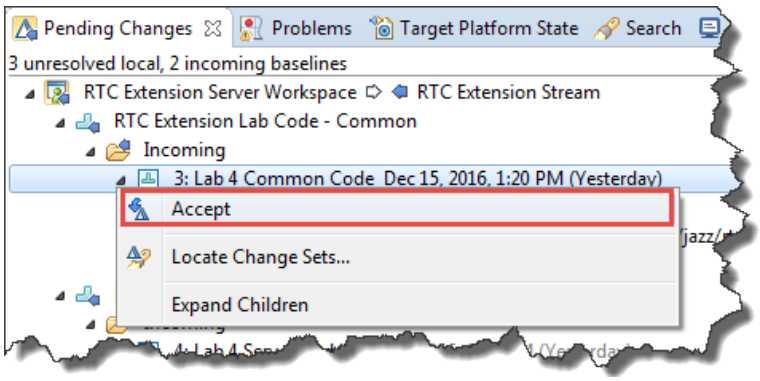
__i. A new nested class has been added, **ParsedConfig**. It is a simple structure used to pass configuration results between the parsing stages. Remember, no instance state variables in an operation participant!

__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.

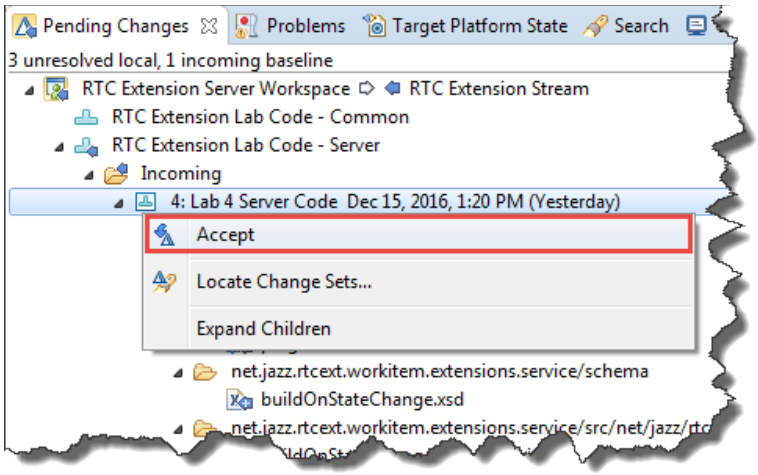
__iii. The run method no longer uses hard coded ids.

__h. Close the comparison editor and then in the **Pending Changes** view.

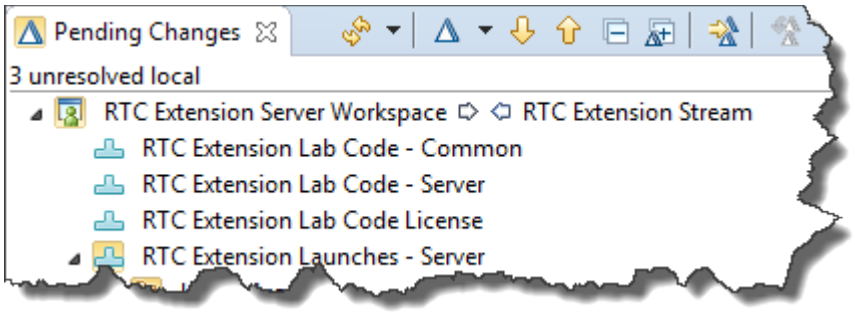
__i. Right click the **Lab 4 Common Code** baseline for the component RTC Extension Lab Code - Common, and then click the **Accept** action.



__ii. Repeat the same for the the **Lab 4 Server Code** baseline for the component RTC Extension Lab Code – Server. This will accept and load the lab 4 delta on top of what you already have loaded from lab 3.



__i. All incoming changes should be accepted.



__116. 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";
```

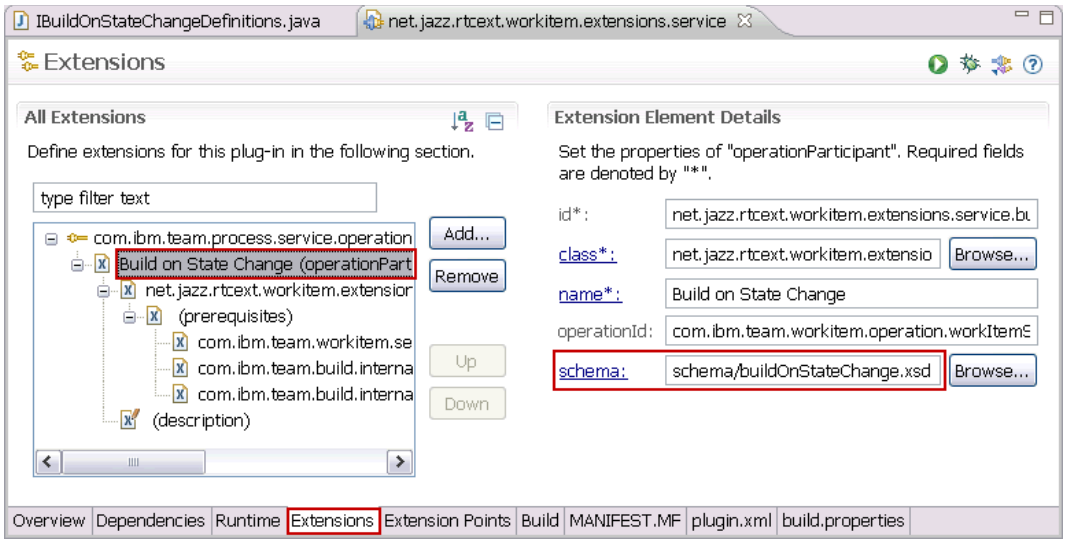
__iii. The second comment and set of constants defines what how the target build is configured.

```
// <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>
```

__117. 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.

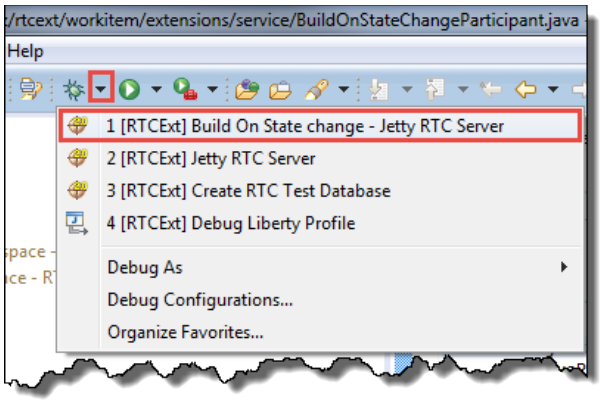
```
/**
 * Second stage of the configuration parsing that handles the build
 * definition.
 *
 * @param parsedConfig
 *            the build definition element is now parsed and the build
 *            definition id is updated. Note that the id is not validated by
 *            this method and may still 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.

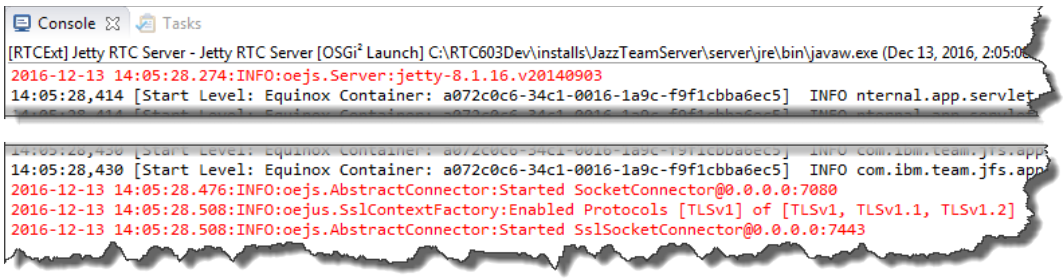
4.2 Launch the Jetty RTC debug server

__118. Use the existing Jetty server launch configuration from lab 2.

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



__b. The **Console** view will show a few log messages indicating that the Jetty RTC debug server is up and running.



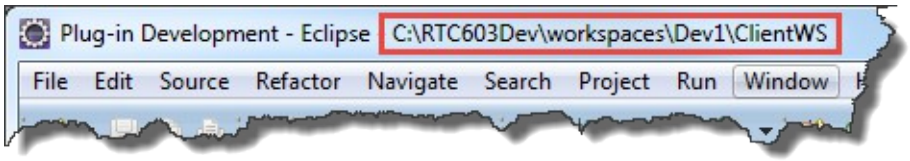
4.3 Launch an RTC test client and connect to the Jetty debug server

__119. Return to the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS`.

__a. If your RTC Client SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

__i. When prompted, select the Eclipse workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS`. Don't check the “Use as default” check box.

__b. Check the desired Eclipse workspace is used.

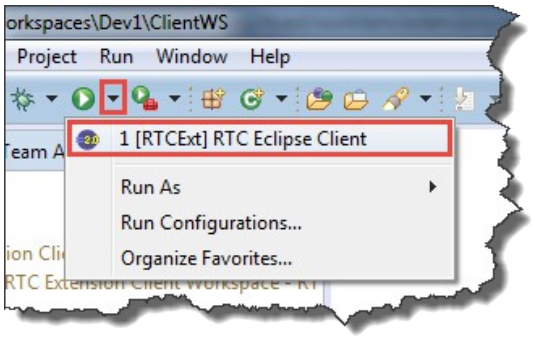


__c. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view in the **Work Items** perspective and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.

__120. Launch the RTC Eclipse Client from the RTC Client SDK development workspace.

__a. 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.

__b. 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.



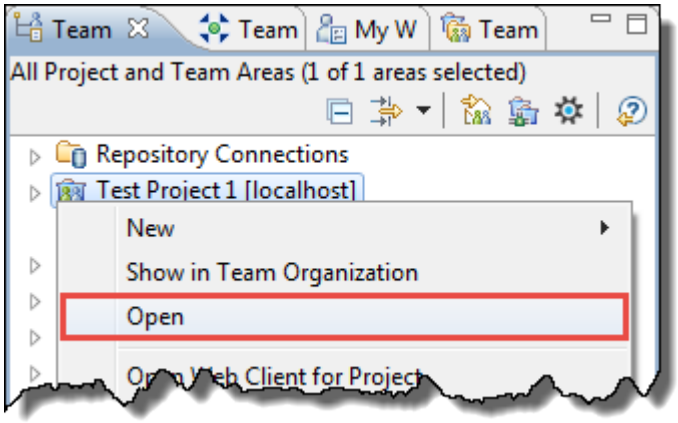
__c. The RTC Eclipse client will start up and will connect automatically to the Jetty debug server you just launched via the repository connection you created in lab 2. You might have to provide the password again. If necessary provide the password `TestJazzAdmin1` for the user `TestJazzAdmin1`.

__i. If asked for a password for secure storage you might not be able to provide this in some versions. Try providing a password. Using **Cancel** always works but does not allow to store the password and forces to enter the password for each log in.

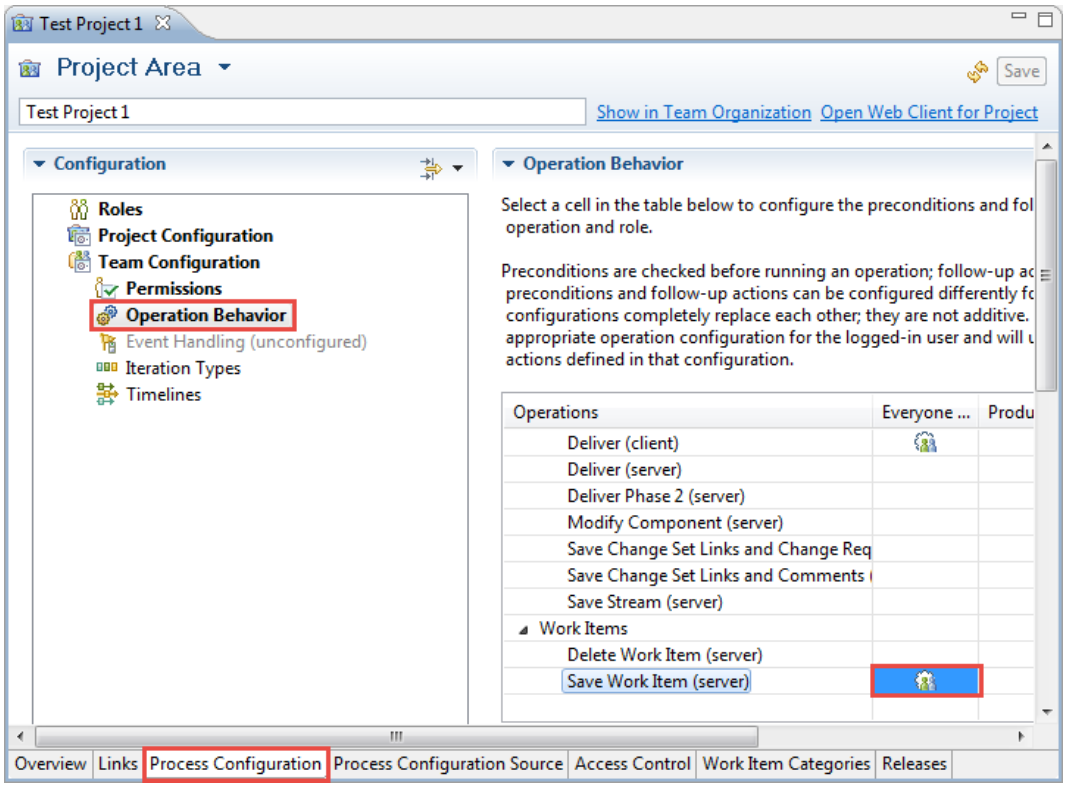
__d. The project area will still be connected; however, you do have some more work to do this time. The participant is still added as a follow-up action on work item save, but it has not been configured with the required work item type, state and build definition ids. You need to fix this.

__e. 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.

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



__g. 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.

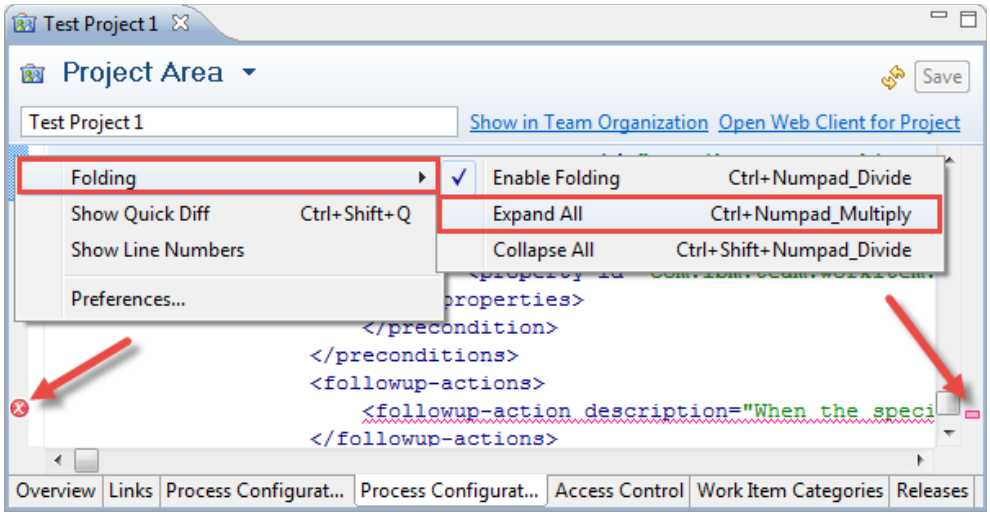


__h. Scroll down to find the **Follow-up actions** section on the right, remove the **Build on State Change** participant that is already in the list and save the change. Add the participant back in and save again. This may seem unusual, but there is a good reason for it. If you looked at the XML for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

__i. Press **Save** in the upper right corner of the editor to save this change.

__121. 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`

The configuration should look like this:

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

__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

__122. 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.

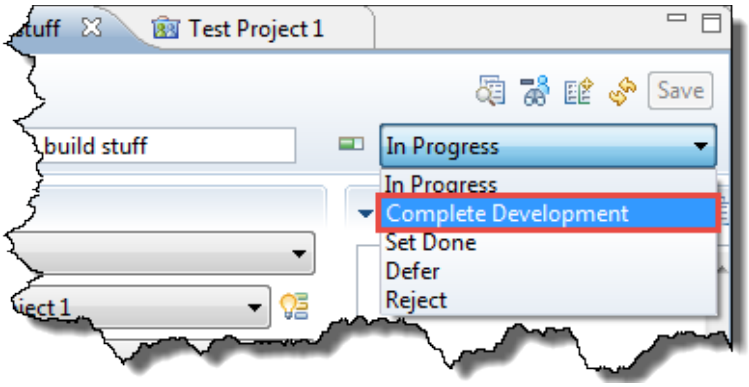
__a. Either of these will cause the breakpoint you set earlier to trigger. If not, re-check the breakpoint is at a valid source code line. The RTC Eclipse client for developing the server extension in which you were studying the code will now surface (if asked about switching to the debug perspective,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

__123. Move the Story to the Implemented state.

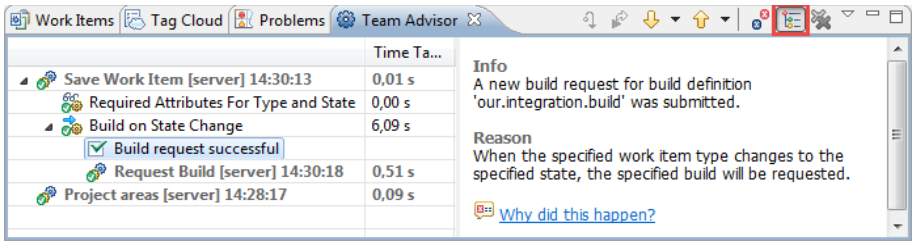
__a. At the upper right portion of the work item editor, select **Complete Development** or **Set Implemented** (depends on which workflow state the story is currently in, if in doubt Reopen the work item) and then click **Save**.



__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

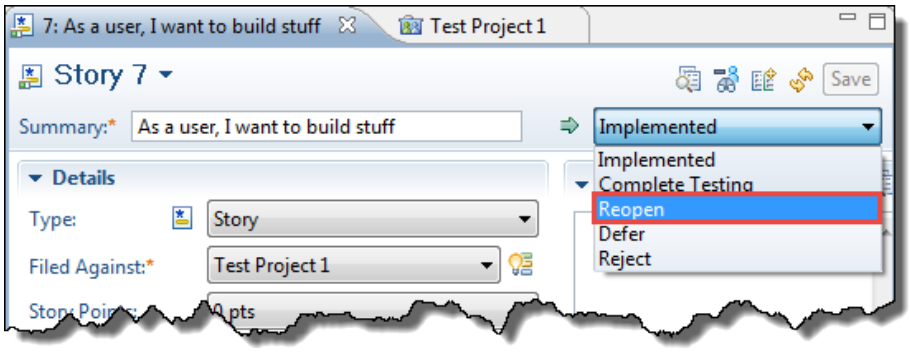
__124. Return to the Test Project 1 project area editor and change the build id.

__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.

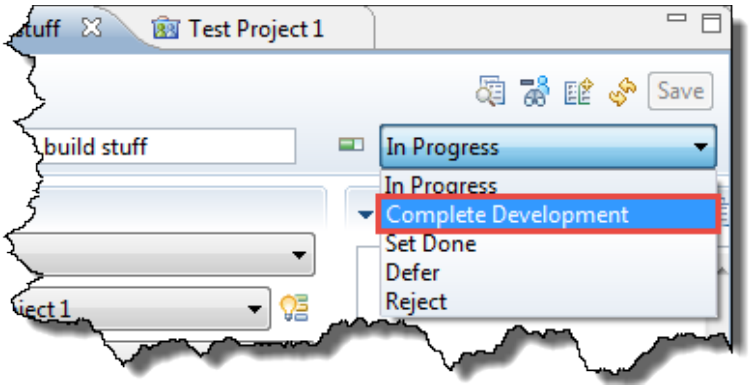
```
<followup-action
  xmlns="http://net.jazz.rtcext.workitem.extensions.service/server/buildOnStateChange"
  description="When the specified work item type changes to the specified state, the
   specified build will be requested."
  id="net.jazz.rtcext.workitem.extensions.service.buildOnStateChange"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

__125. 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.

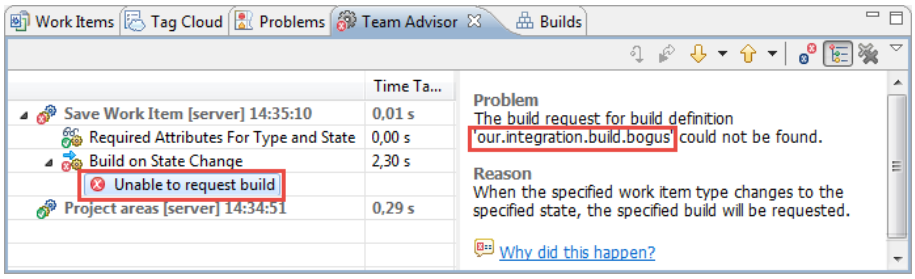


__b. Again in the work item editor, select **Complete Development** from the same dropdown and click **Save** again.



__c. This time, when the debugger surfaces, you can step into the **parseConfig2** 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.

__126. 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 RTC Eclipse Server Development client used to launch the Jetty server select the **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).

Please note that the Aspect Editor is an Eclipse client extension plug-in. The Aspect Editor is only available in RTC Eclipse clients where it is deployed.

5.1 Understanding the Aspect Editor

__127. Start the RTC development server, if it is not already started.

__a. Open a Windows Explorer and navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file.

__128. Return to the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS` from the last lab.

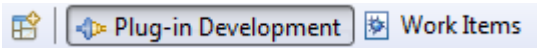
__a. If your RTC Client SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

__i. When prompted, select the Eclipse workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS`. Don't check the "Use as default" check box.

__b. Check the desired Eclipse workspace is used.

__c. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view in the **Work Items** perspective and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.

__d. Switch to the **Plug-in Development perspective**. In the toolbar toward the right, click **Plug-in Development** to switch the perspective.

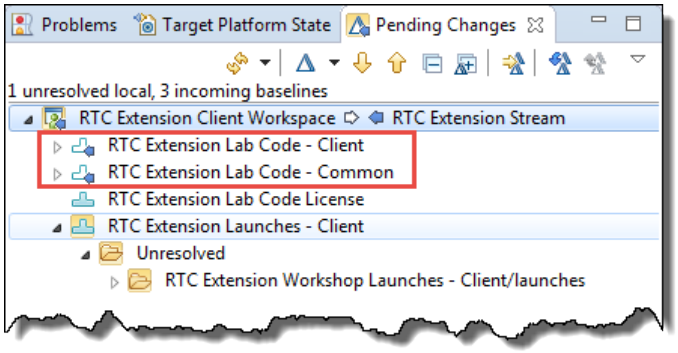


__i. If the **Plug-in Development** perspective is not available, open it now by selecting **Window > Open Perspective > Other… > Plug-in Development** from the menu bar.

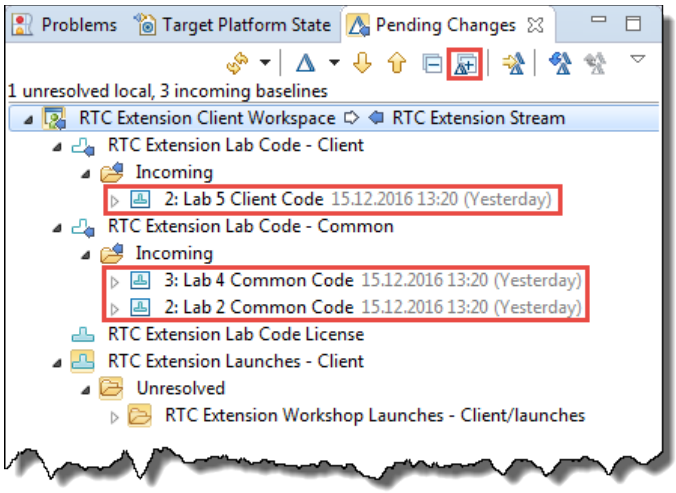
__129. Browse and load the Lab 5 code into the RTC Eclipse client used for Client SDK plug-in development.

__a. Open the **Pending Changes** view, if it is not yet available in the perspective use **Window>Show View>Other...** and select Pending Changes.

__b. Note the unloaded components **RTC Extension Lab Code – Client** and **RTC Extension Lab Code – Common**. So far, there was no extension development for the Eclipse client. The code that was created only works in the RTC server. The Aspect Editor will now be a RTC Eclipse client extension. The code for it is in the component **RTC Extension Lab Code – Client**. The client code shares some code with the server extension especially related to the schema and configuration elements which is in the component **RTC Extension Lab Code – Common**. This is a typical pattern. The next steps prepare the client development workshop.

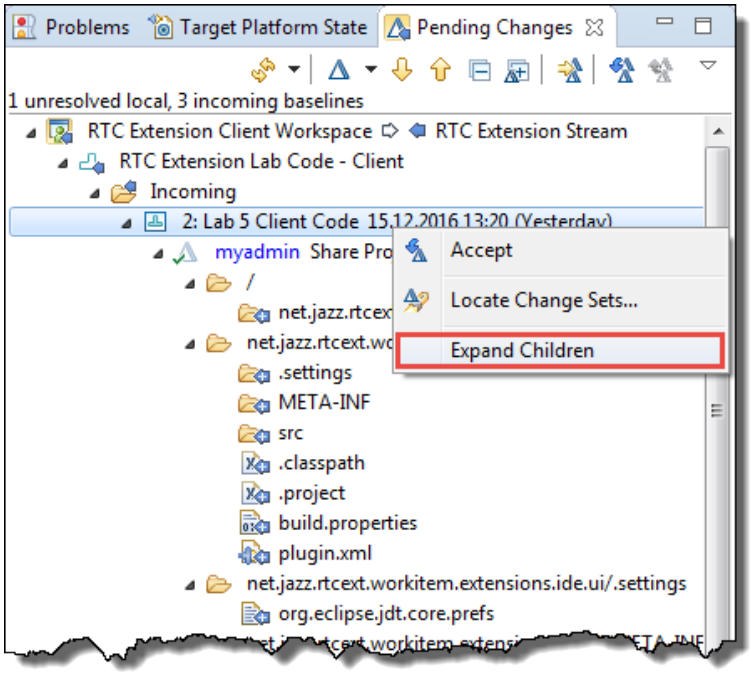


__c. In the **Pending Changes** view, click the **Expand to Change sets** icon. This will show the incoming baselines. One incoming baseline for the Lab 5 Client Code is shown at the component **RTC Extension Lab Code – Client** as shown here. Baselines for shared common code developed for the server extension are incoming at the component **RTC Extension Lab Code – Common**.

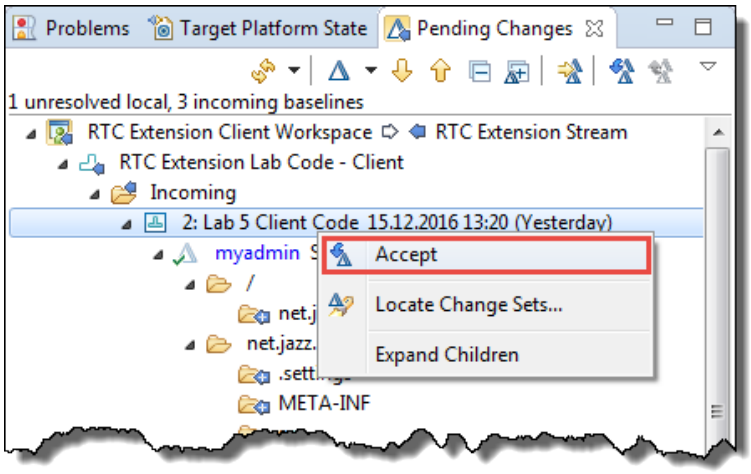


__d. Browse the Client extension code for the aspect editor.

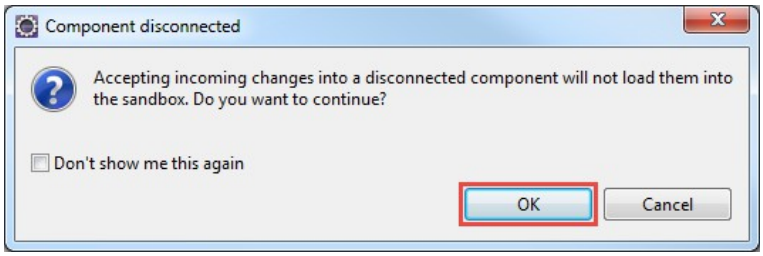
__i. Right click the **Lab 5 Code** baseline under the component component **RTC Extension Lab Code – Client**., and then click the **Expand Children** action. This will reveal all the changes made for lab 5. As you can see the full change is the addition of a new plug-in project. The first entry shows 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.



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

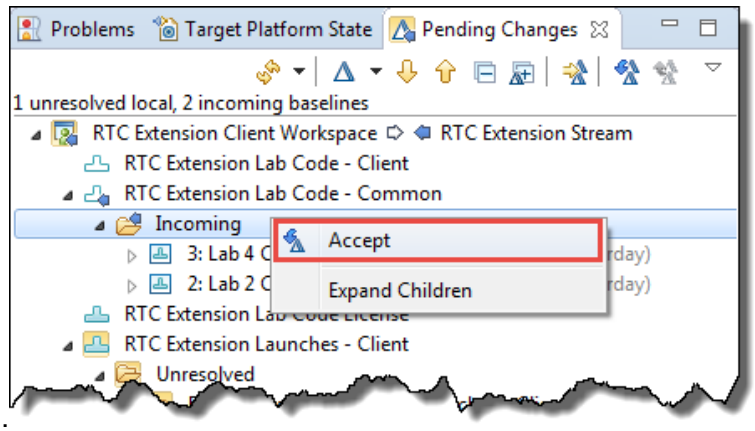


__iii. A message comes up explaining that the component is not yet loaded in this Eclipse workspace and the accept does not load it automatically. Press OK to dismiss the dialog and continue. You will load the component later.



__e. Accept the incoming baselines for the previews labs in the common code component.

__i. In the **Pending Changes** view, right click the **incoming** node under under the **RTC Extension Lab Code – Common** component node, and then click the **Accept** action. This will accept the common code that is shared with the server part of the extension.

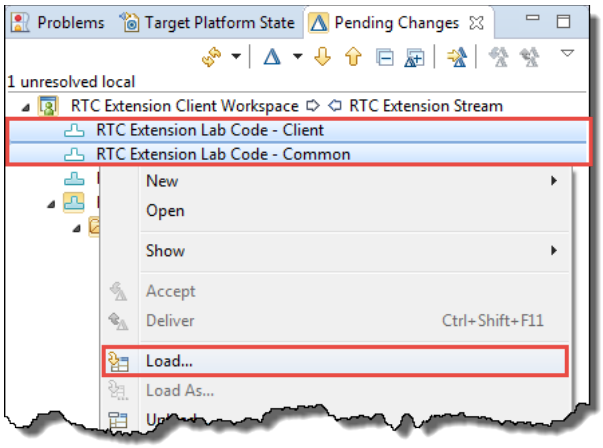


.

__ii. The message explaining the component is not yet loaded in this Eclipse workspace comes up again. Press OK to dismiss the dialog and continue. You will load the components in the next steps.

__f. Load the components to the local workspace.

__i. Select the components **RTC Extension Lab Code – Client** and **RTC Extension Lab Code – Common** then **right click** the selection and click **Load...** to load the content into the Eclipse workspace.

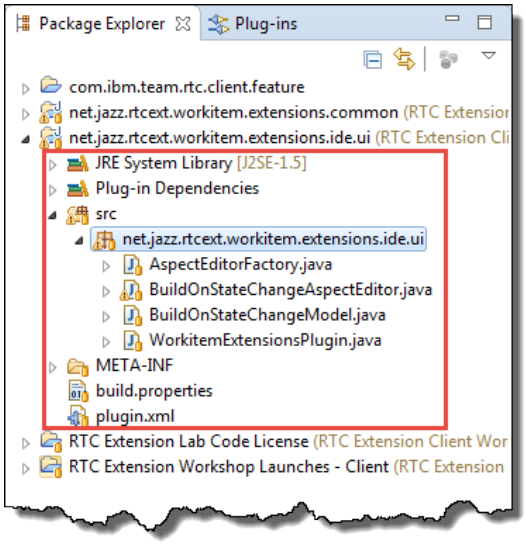


__ii. In the **Load Repository Workspace** wizard, make sure **Find and load Eclipse projects** is selected and then click **Finish**.

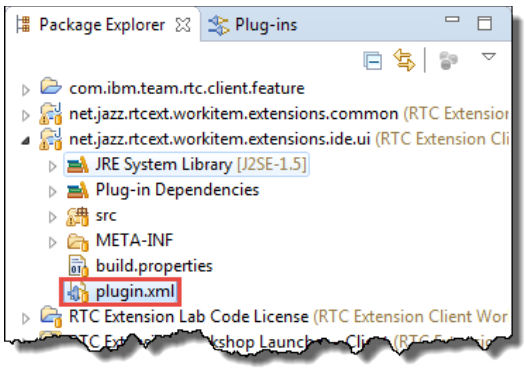
__iii. You might see a warning for the common component that the components are already loaded in other workspaces. This is true. It requires careful handling of changes to these components. Best is to unload the common component in one of the workspaces and only do changes in the other. After the changes are done, load the component again, to pick up the changes. However, we are experts and ignore this advice for the workshop.

__130. Understanding the aspect editor plug-in.

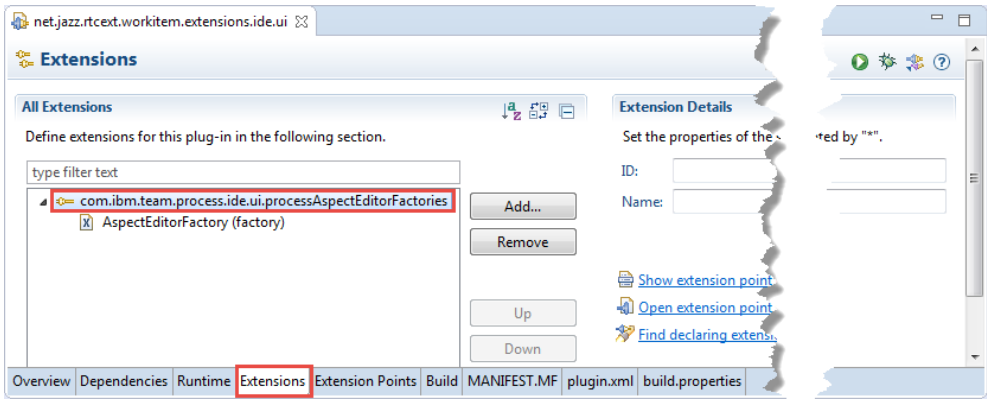
__a. In the **Package Explorer** view, expand the tree for the new user interface project (net.jazz.rtcext.workitem.extensions.ide.ui) and review the files.



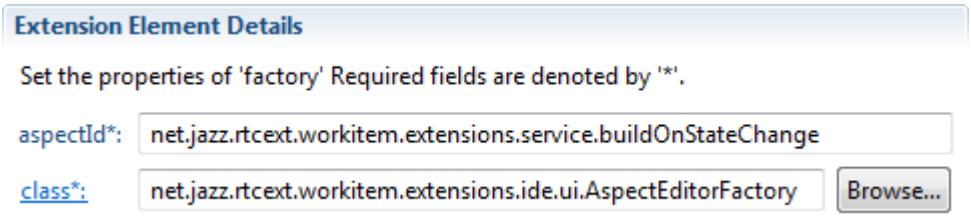
__b. Double click the **plugin.xml** file. The editor that opens presents information from not only the plugin.xml file but also the build.properties and META-INF/MANIFEST.MF files. As before, the content reflects standard Eclipse plug-in practices. Note on the **Overview** page that there is one significant 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.



__c. 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.



__d. 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.



__131. 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.

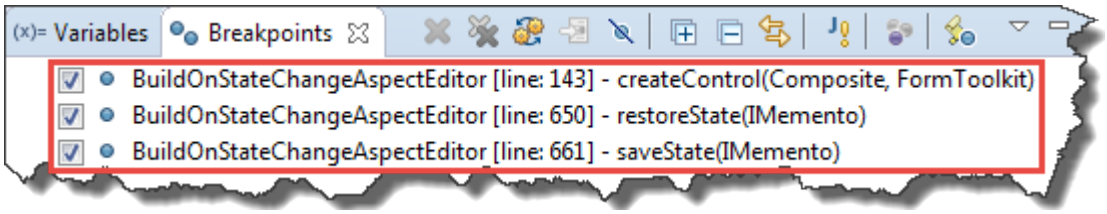
- The getWorkItemTypes method returns an array of WorkItemType instances that are valid for the project area. It only calculates the list once per aspect editor instance. It also contains the optimization around lists of states for work item types in the same type group. It only calls getStatesForTypeCategory 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. That code has not changed at all for this lab. The image below shows the breakpoints in the Debug Perspective.



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

__h. Keep the Eclipse client open.

5.2 Launch the Jetty RTC debug server

__132. Return to the Eclipse client with the workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS` from the last lab.

__a. If your RTC Server SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

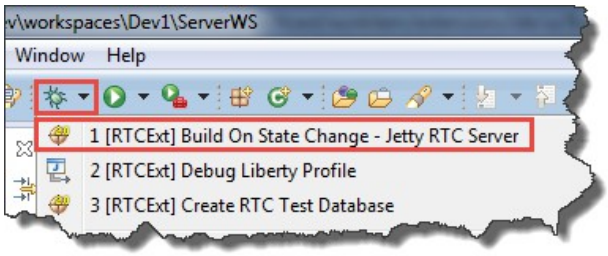
__i. When prompted, select the Eclipse workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS`. Don't check the "Use as default" check box.

__b. Check the desired Eclipse workspace is in use.

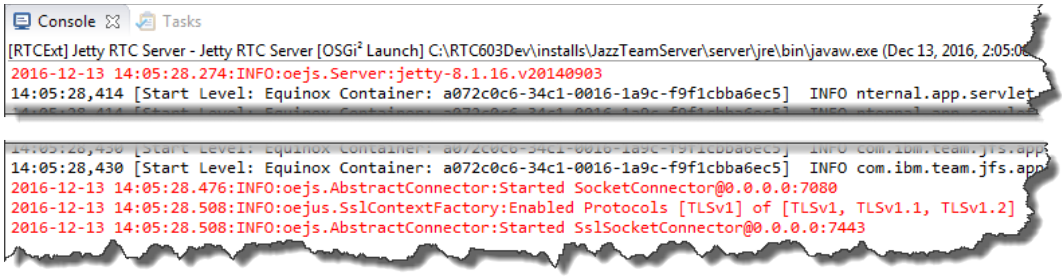
__c. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view in the **Work Items** perspective and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.

__133. Use the existing Jetty server launch configuration from the prior labs.

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



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



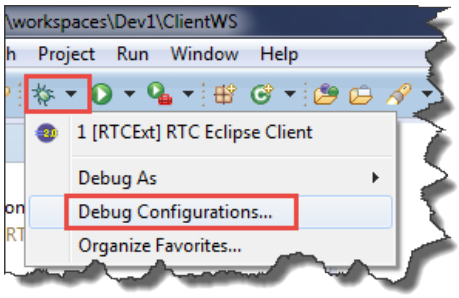
5.3 Launch an RTC Client and Configure the Participant

__134. Return to the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS` in use before the last section..

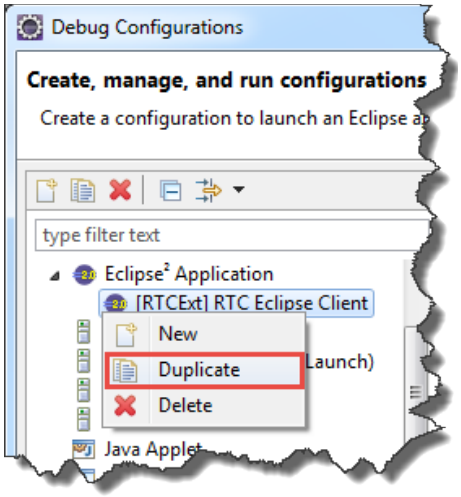
__a. Check the desired Eclipse workspace is used.

__135. Create a new launch configuration for the RTC Client plus your aspect editor.

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

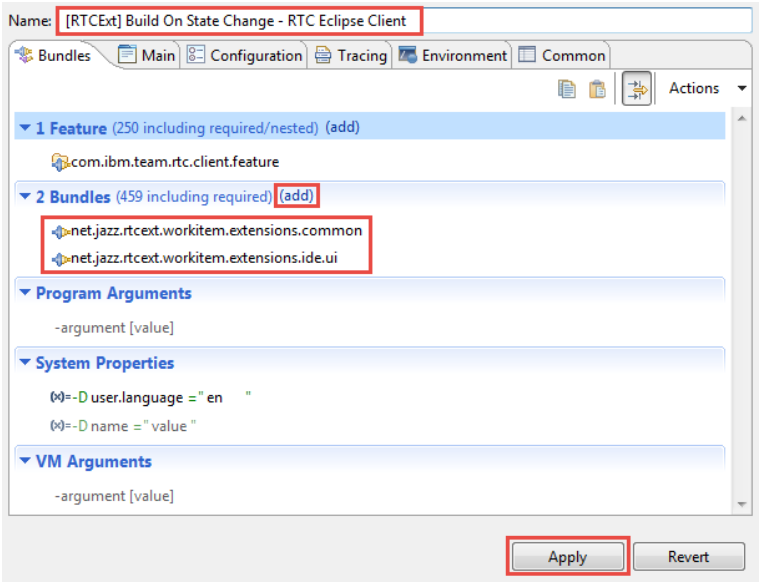


__b. In the **Debug Configurations** dialog, expand the **Eclipse² Application** tree and right click the **[RTCExt] RTC Eclipse Client** configuration and then from the popup menu, select **Duplicate**. Note that you are not changing the existing launch but creating a copy of it. You should keep the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

__136. 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.

__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 the participant is fully configured from lab 4

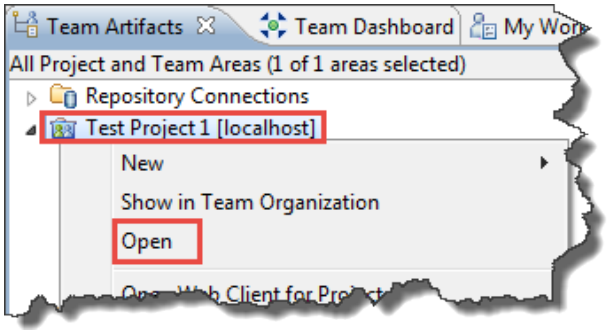
__i. You might have to provide the password again. If necessary provide the password `TestJazzAdmin1` for the user `TestJazzAdmin1`.

__ii. If asked for a password for secure storage you might not be able to provide this in some versions. Try providing a password. Using **Cancel** always works but does not allow to store the password and forces to enter the password for each log in.

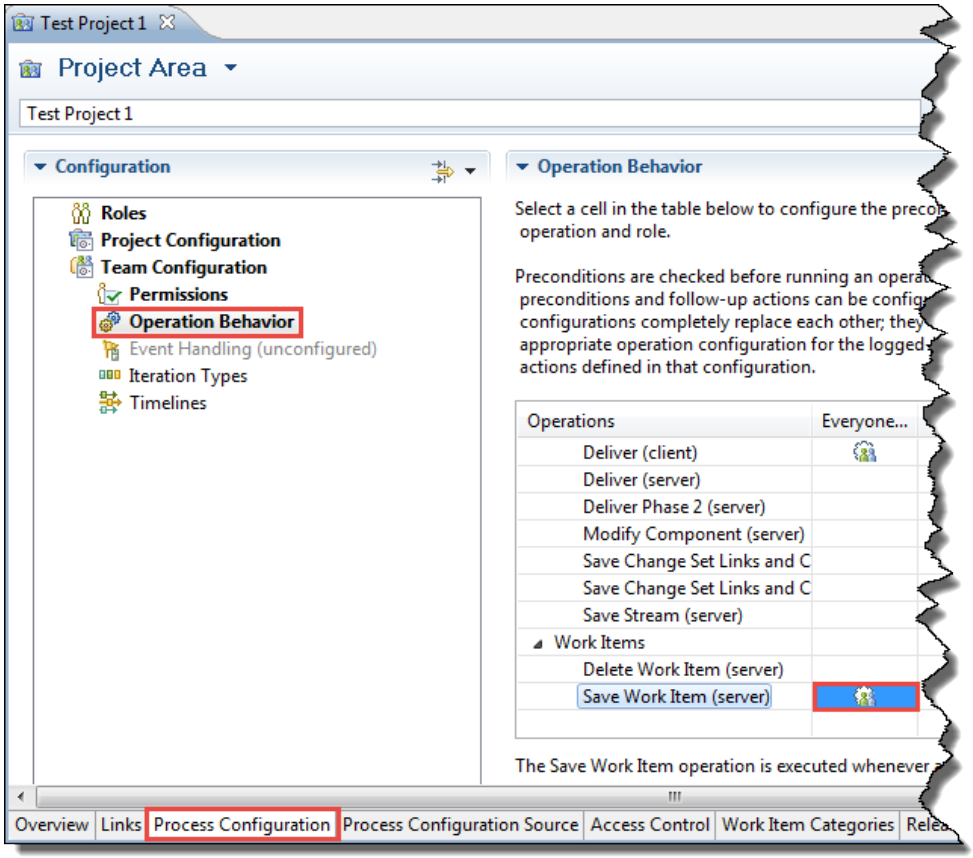
__d. The next time you want to debug this server configuration, you will be able to click a shortcut to it on the dropdown of the **Debug** toolbar icon. You will not need to open the **Debug Configurations** dialog.

__137. Try out the new aspect editor.

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



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



__c. Scroll down to find the **Follow-up actions** section on the right and select the **Build on State Change** entry.

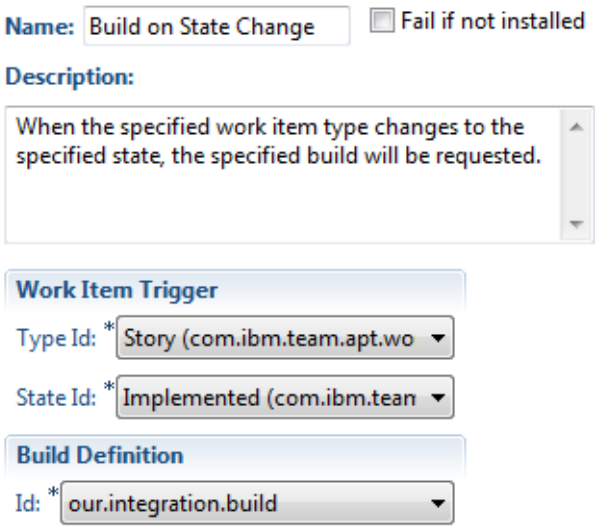
__i. If you set it, your breakpoint in the restoreState method will trigger. Step into and through the two methods called from here.

__ii. Hit the debugger's resume button and your breakpoint in createControl will trigger. Step into and through the methods called from here.

__iii. After you hit resume from createControl or one of its called methods, the breakpoints in the selection changes listeners will start to trigger because of the initial setting of the combobox selected element during initialization.

__iv. Once you have hit resume after all the selection change listener breakpoints (each may trigger twice), switch back to the launched RTC Eclipse client and see the aspect editor in action.

__d. The selected values in the comboboxes should look familiar. In fact, even better since the actual work item type and state names and not just the ids are shown. Note that the id is all that is put into the process XML.



__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.

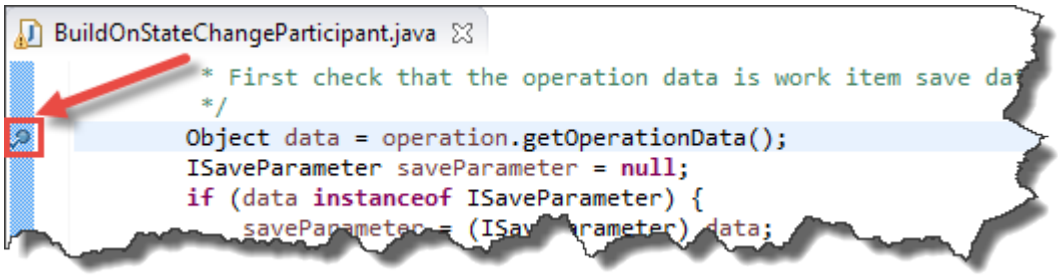
__g. **Leave the editor open at this point.** You will soon come back here and make a change.

5.4 Trigger the Participant

__138. 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.

__139. 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 in the Eclipse client with the workspace used for RTC Server SDK API development to trigger (unless you cleared it). That 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.

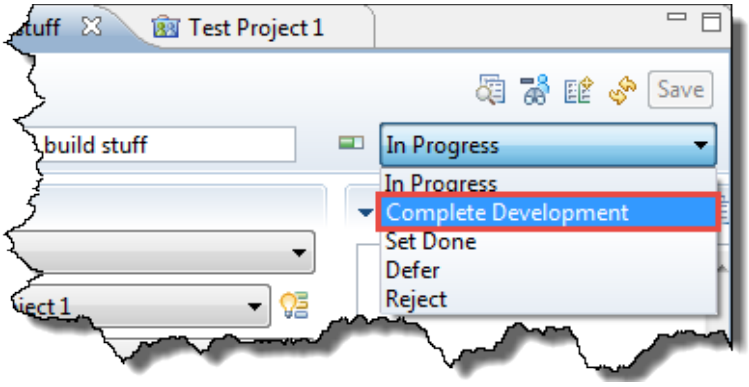


__b. Simply resume execution since this code has not changed ( ).

__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 ( ).

__140. Move the Story to the Implemented state (or your different type to the trigger state).

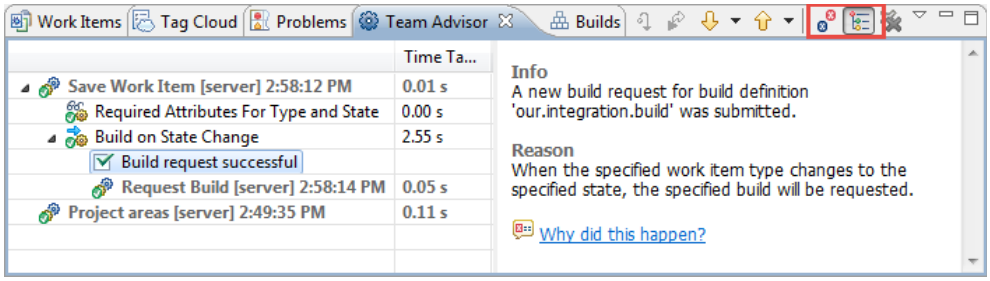
__a. At the upper right portion of the work item editor, select **Set Implemented** or **Complete Development** (depends on which workflow state the story is currently in) and then click **Save**.



__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 ( ).

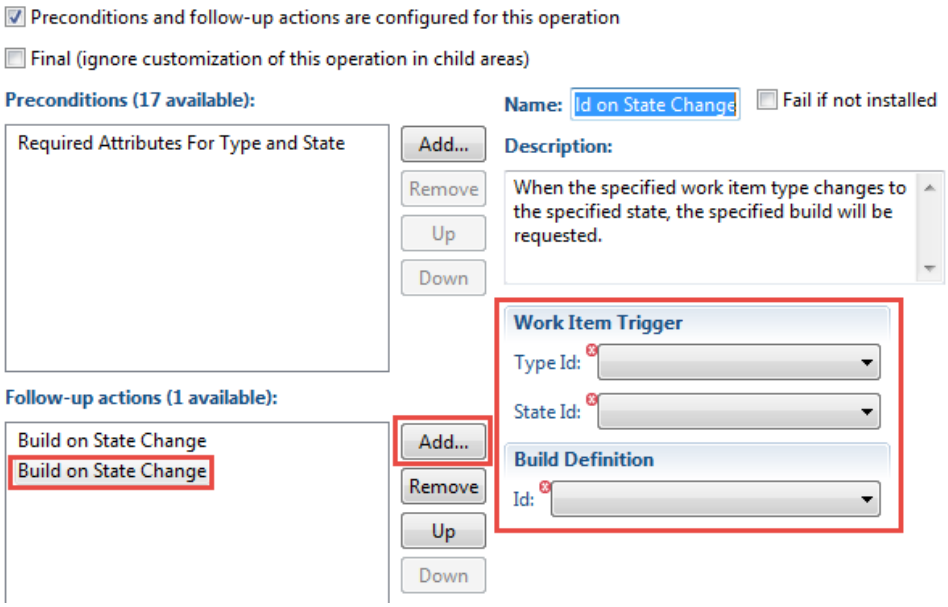
__d. If you go to the **Team Advisor** view and check to make sure the **Show Failures Only** filter is off and **Show Detail Tree** is on (see highlight below), you can browse the results of this successful operation. Also, if you double click **our.integration.build** in the **Team Artifacts** view,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



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

__141. 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.



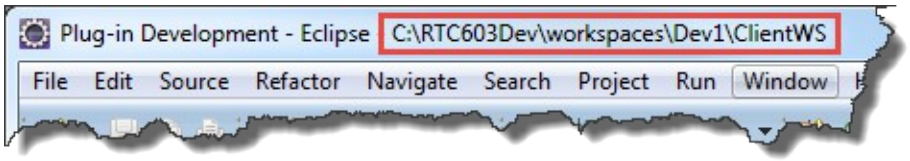
__b. Select a work item and state that are different from the ones configured for the first instance. Select the one and only build definition. If you wish, you can create a new build definition. If you do create a new build definition, you will not see it until a new instance of the aspect editor is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

__142. 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.

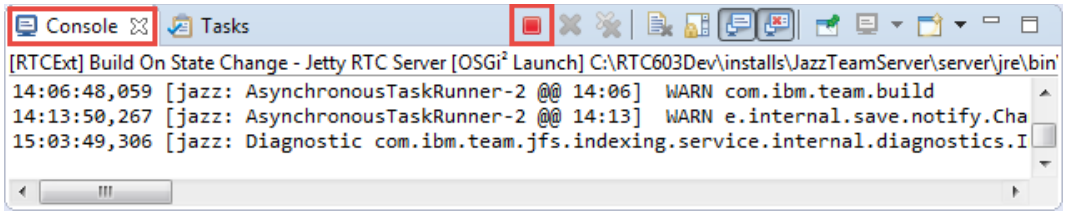
__143. Close down the launched RTC Eclipse debug client and the Jetty server.

__a. Close the RTC Eclipse client where you were working with the work items and project area.

__b. Close the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS`.



__c. Back in the Eclipse client with the workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS`, 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 extension

Lab Scenario

Now the code is really complete. Only the deployment to the production environment is left to do. This lab will show these steps for deploying on the server and on the client. The process for both tasks is very similar with some small differences.

6.1 Create a Feature for the RTC server extension

__144. Start the RTC development server, if it is not already started.

__a. Open a Windows explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file.

__145. Return to the Eclipse client with the workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS`.

__a. If your RTC Server SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

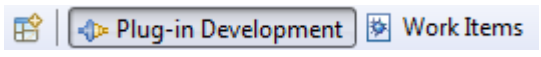
__i. When prompted, select the Eclipse workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS`. Don't check the "Use as default" check box.

__b. Check the desired Eclipse workspace is in use.

__c. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view in the **Work Items** perspective and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.

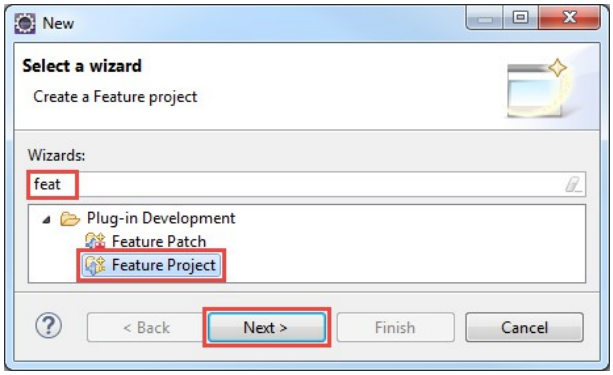
__146. Create the server side feature. The feature is used to collect all required plug-ins for the Build On State change server extension.

__a. Switch to the **Plug-in Development perspective**. In the toolbar toward the right, click **Plug-in Development** to switch the perspective.

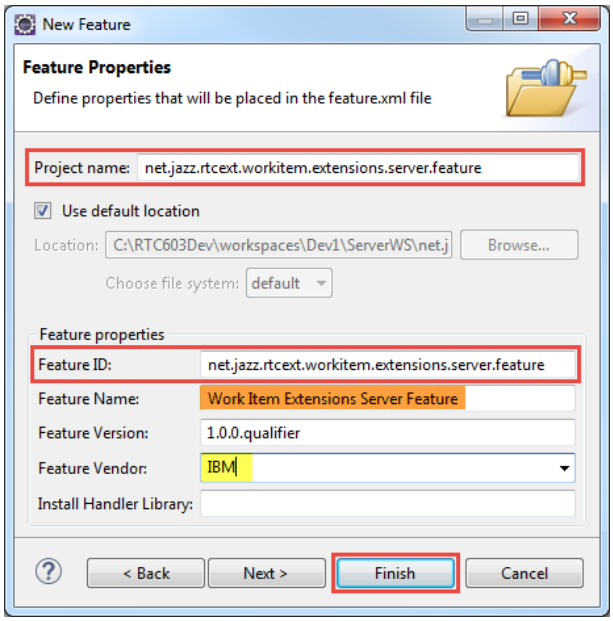


__a. If the **Plug-in Development** perspective is not available, open it by selecting **Window > Open Perspective > Other… > Plug-in Development** from the menu bar.

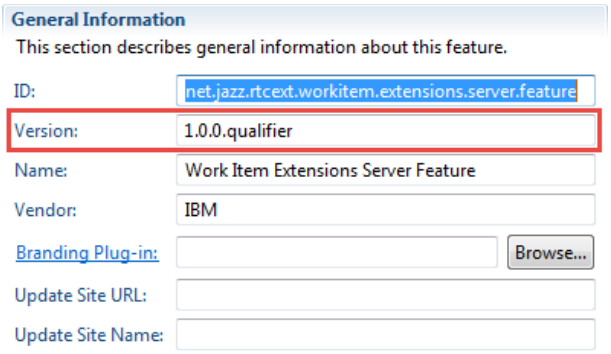
__b. From the menu bar, select **File > New > Project…** then in the **New Project** wizard, type `feature` in the filter field, select **Feature Project** from the list and then click **Next**.



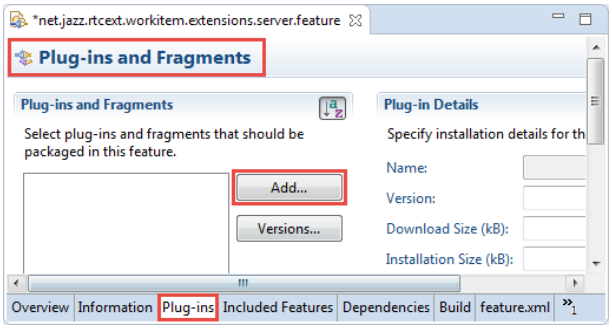
__c. On the second page of the wizard type `net.jazz.rtcext.workitem.extensions.server.feature` into the **Project name** field. As you type, the **Feature ID** is set to a reasonable value but the **Feature Name** should be set to a more descriptive name: `Work Item Extensions Server Feature`. You can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



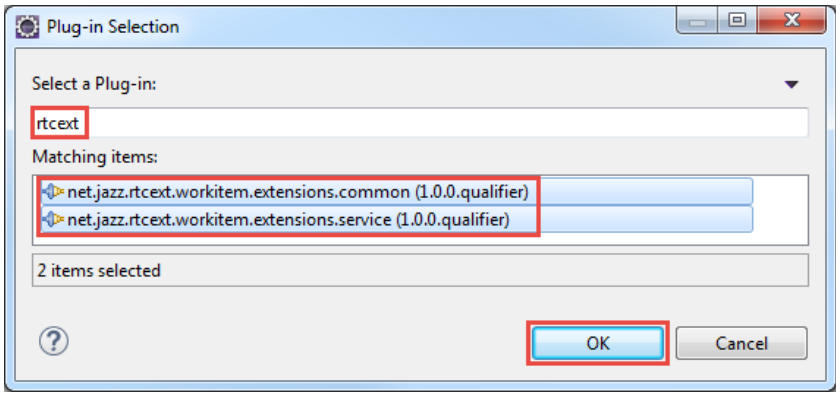
__d. Your new feature project appears in the **Package Explorer** view and an editor opens on the feature.xml file. On the **Overview** tab, make sure the **Version** is set to `1.0.0.qualifier`. This is the same Eclipse best practice you used for the plug-ins.



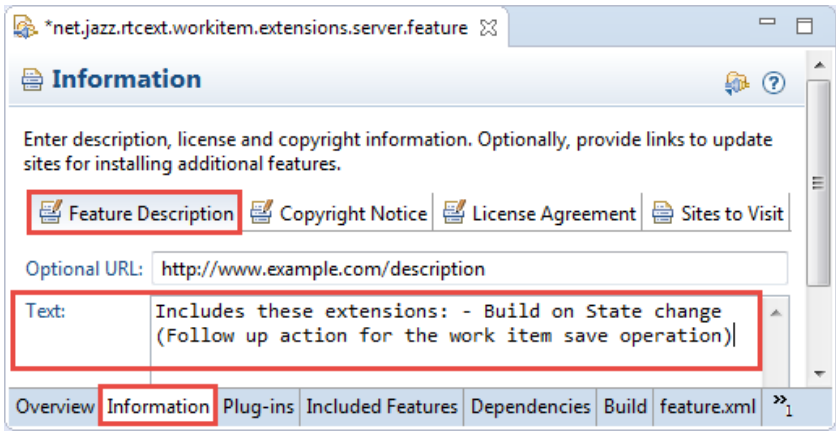
__e. The feature currently does not contain any plug-ins, you will add them now. In the editor, switch to the **Plug-ins** tab. In the section **Plug-ins and Fragments** click the **Add...** button.



__f. In the Plug-in Selection dialog type"rtcrext" to filter for your plug-ins. Select the plug-ins ***net.jazz.rtcext.workitem.extensions.common*** and ***net.jazz.rtcext.workitem.extensions.service*** and then press **OK**.

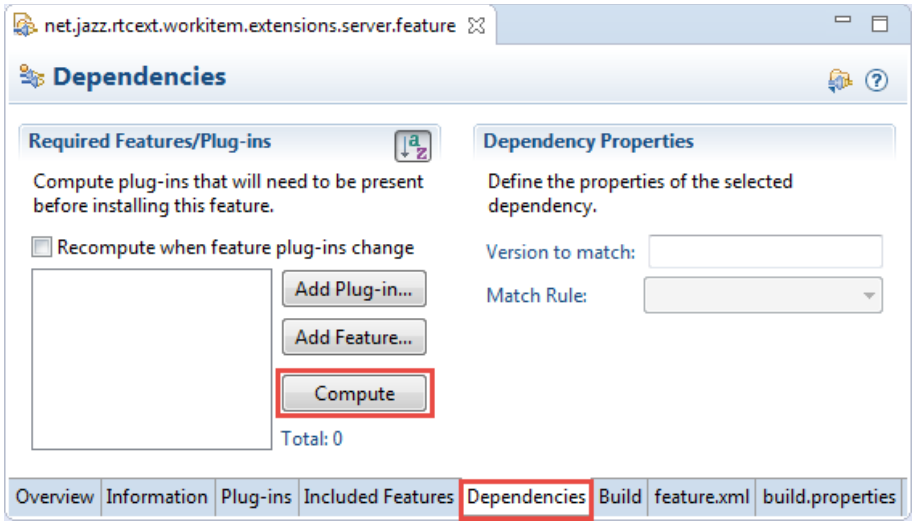


__g. Still in the editor, switch to the **Information** tab, select the **Feature Description** sub-tab and enter a **Text** description as shown here. If you wish you can look at other information that can be added, such as a copyright and license information.



__h. Type **Ctrl+S** to save the `feature.xml` file.

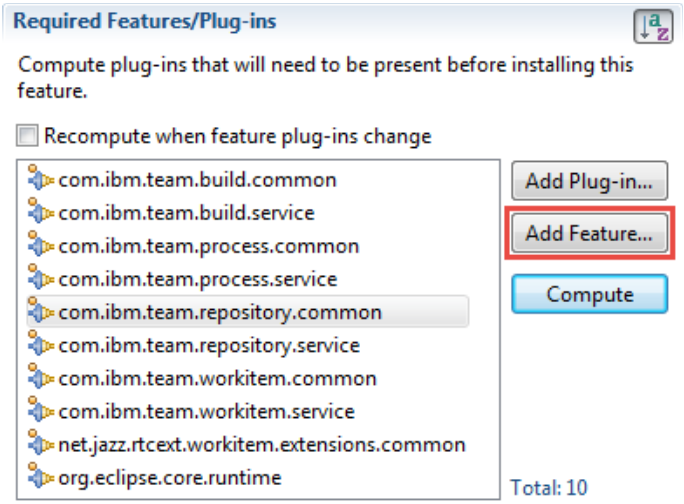
__i. It is important to add the dependencies that are needed to the feature and you will do this in the next steps. Switch to the **Dependencies** tab and click **Compute**.



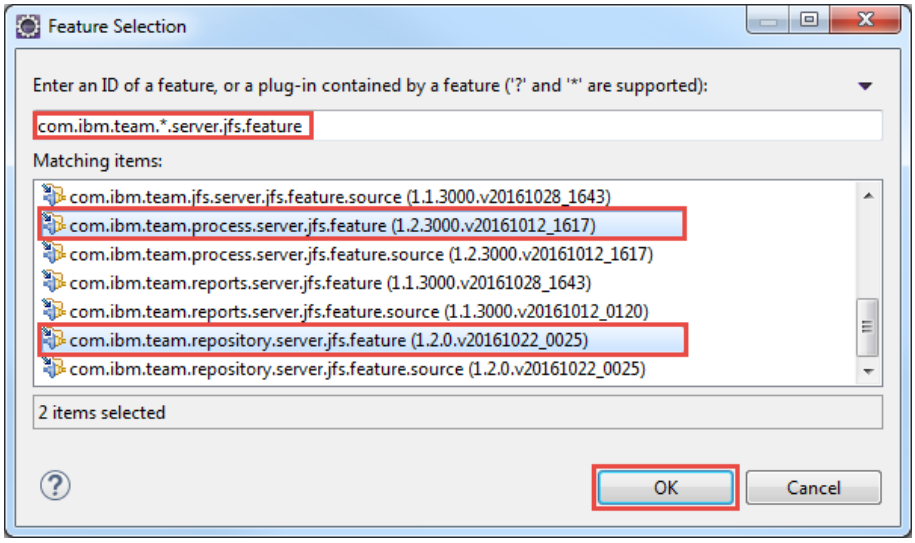
__j. The dependencies list is computed as shown here. The dependencies are expressed in terms of plug-ins; however, for Jazz server side provisioning, you need to use features.

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…**

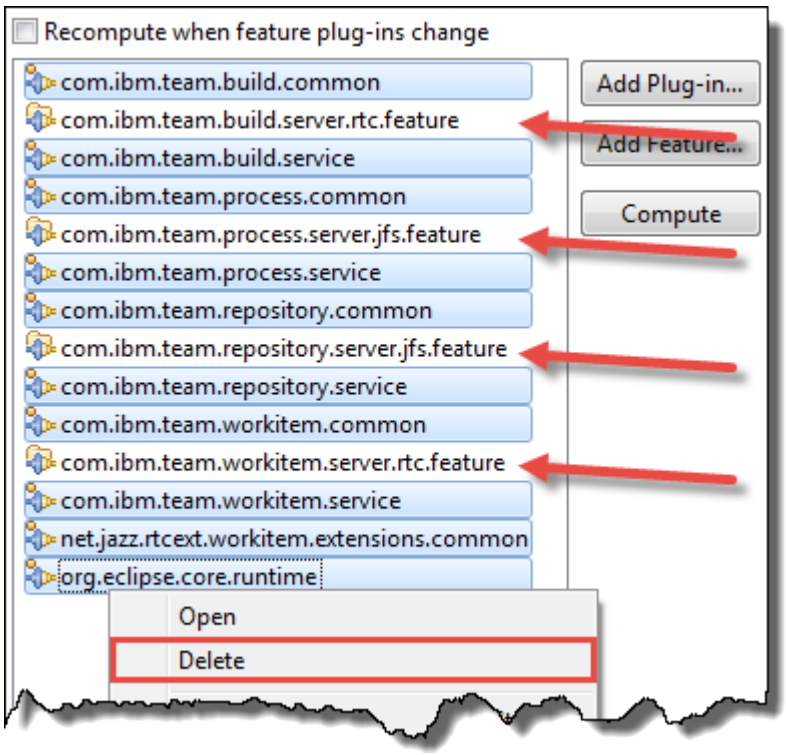


__k. In the **Feature Selection** dialog type `com.ibm.team.*.server.jfs.feature` into the filter field, select the two features shown here (**com.ibm.team.process.server.jfs.feature** and **com.ibm.team.repository.server.jfs.feature**) and then click **OK**.

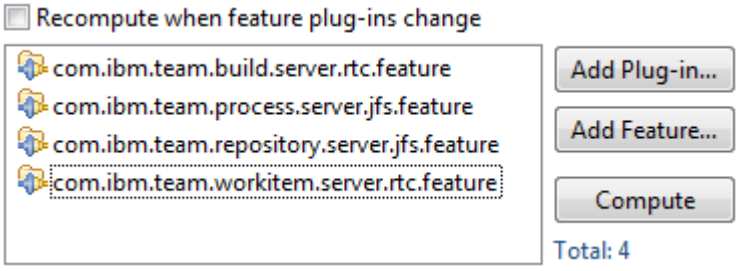


__l. Click **Add Feature...** again, but this time in the **Feature Selection** dialog type `com.ibm.team.*.server.rtc.feature` into the filter field, select these two features (**com.ibm.team.build.server.rtc.feature** and **com.ibm.team.workitem.server.rtc.feature**) and then click **OK**.

__m. 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.



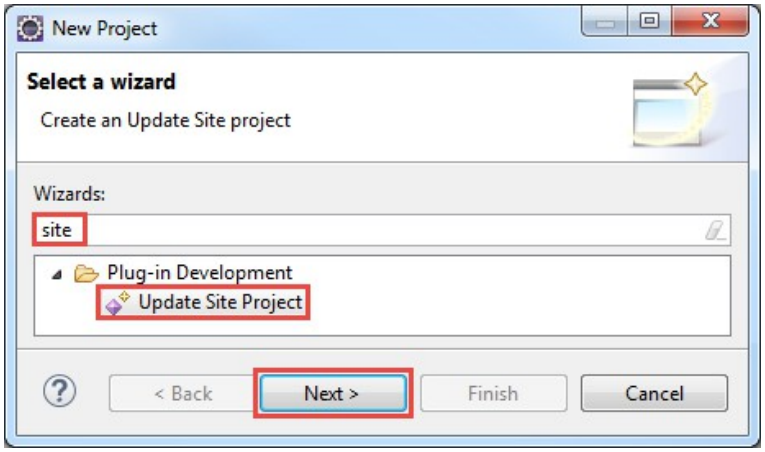
__n. The list will now look like this. Type **Ctrl+S** to save the feature.xml file. You can now close the editor.



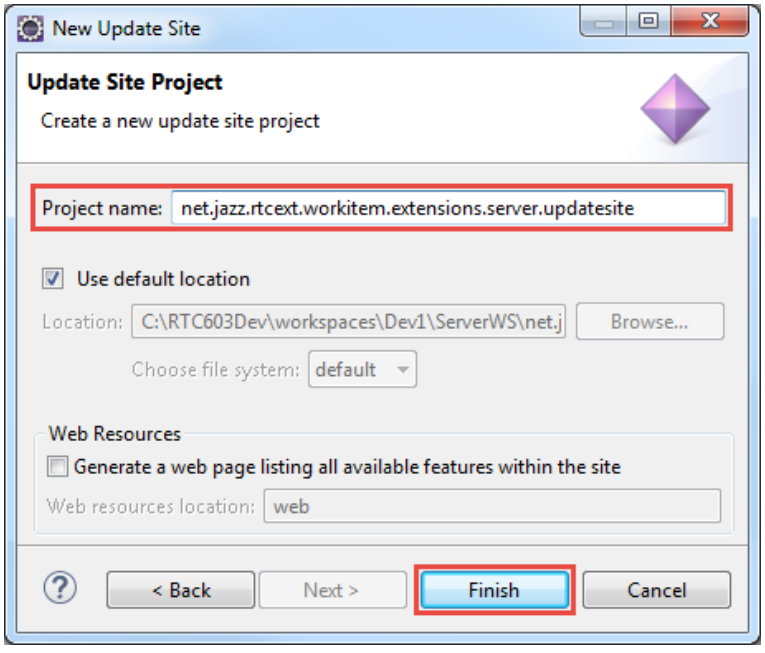
6.2 Create an Update Site for the server extension

__147. Create the update site used to generate the server deployable binaries.

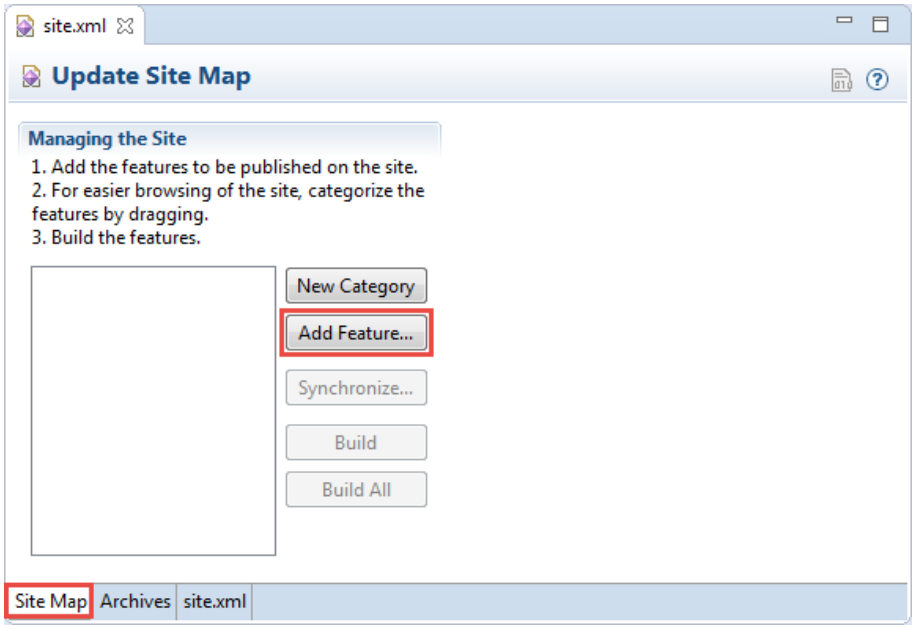
__a. From the menu bar, select **File > New > Project…** then in the **New Project** wizard, type `site` in the filter field, select **Update Site Project** from the list and then click **Next**



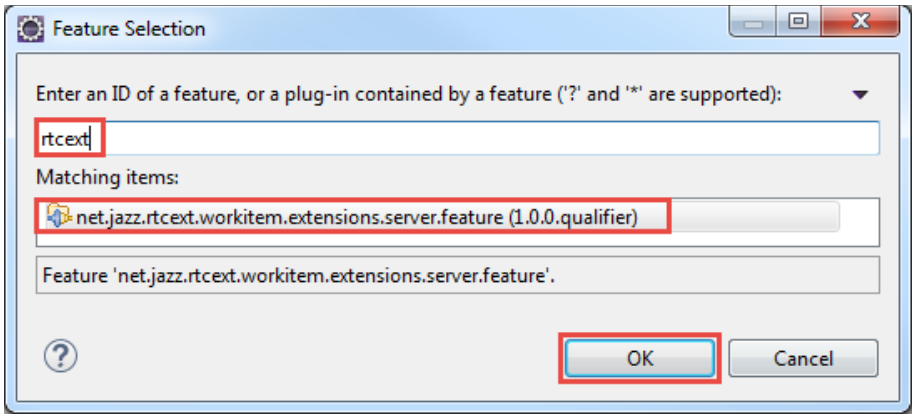
__b. On the second page of the wizard type `net.jazz.rtcext.workitem.extensions.server.updatesite` into the **Project name** field. Click **Finish**.



__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**.

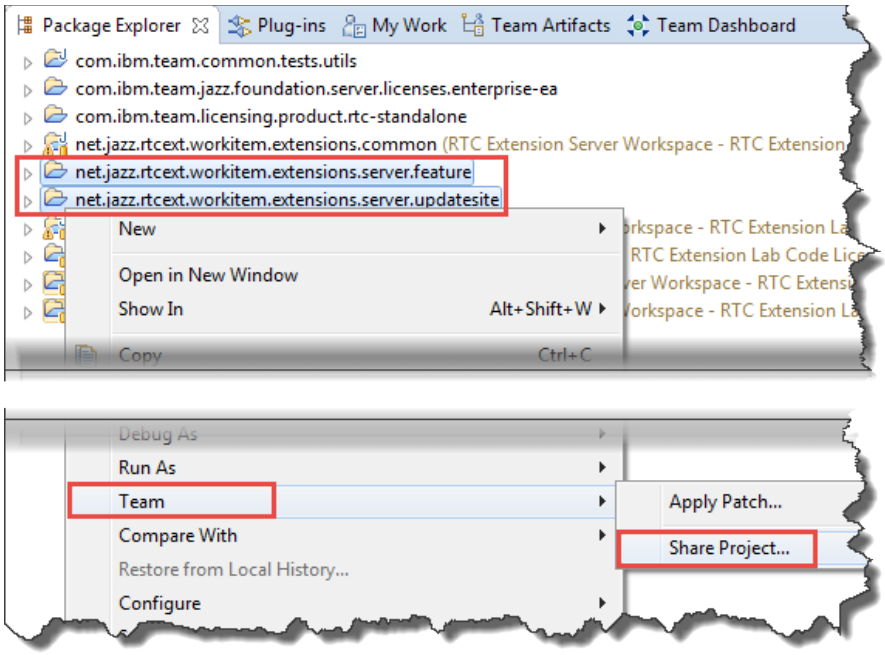


__d. In the **Feature Selection** dialog, type `rtcext` into the filter, select the feature you created in the last section and then click **OK**. Back on the site.xml editor type **Ctrl+S** to save the file.

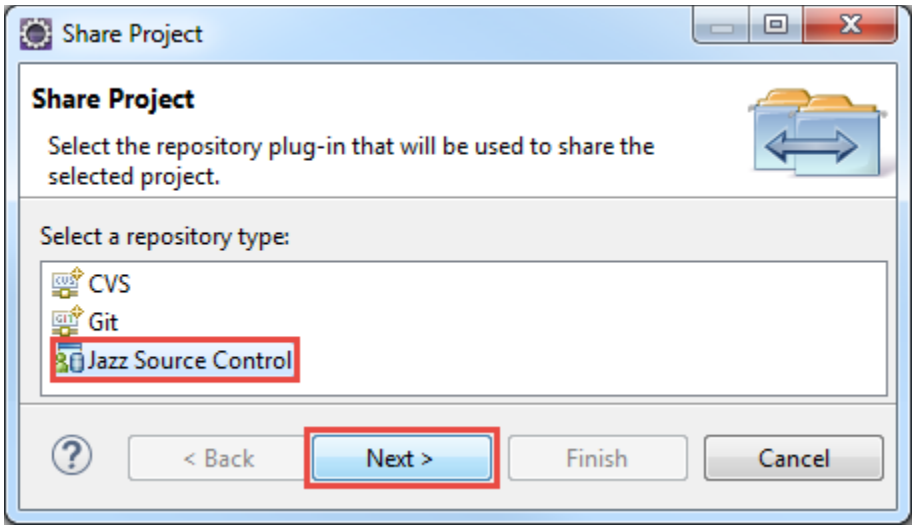


__148. Share the new projects to your repository workspace.

__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…**

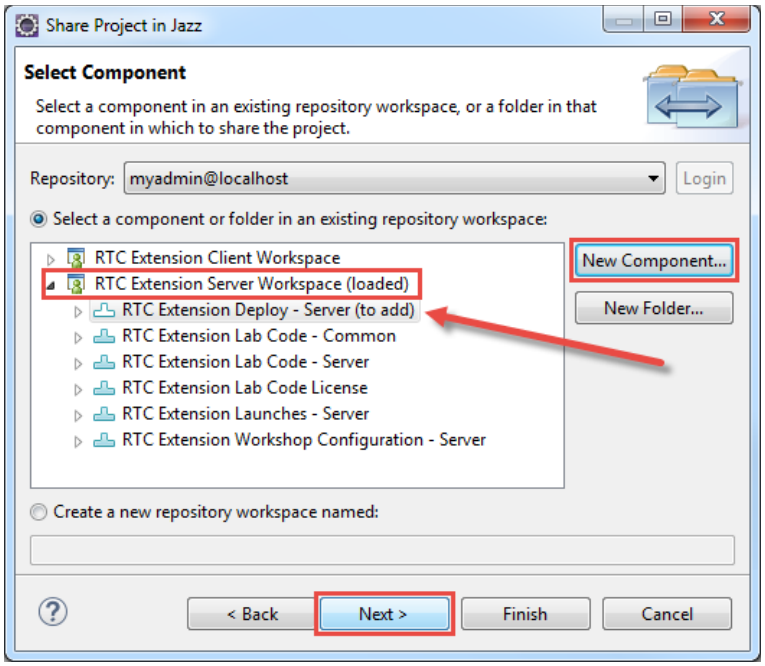


__b. In the **Share Project** wizard, select **Jazz Source Control** then click **Next**.

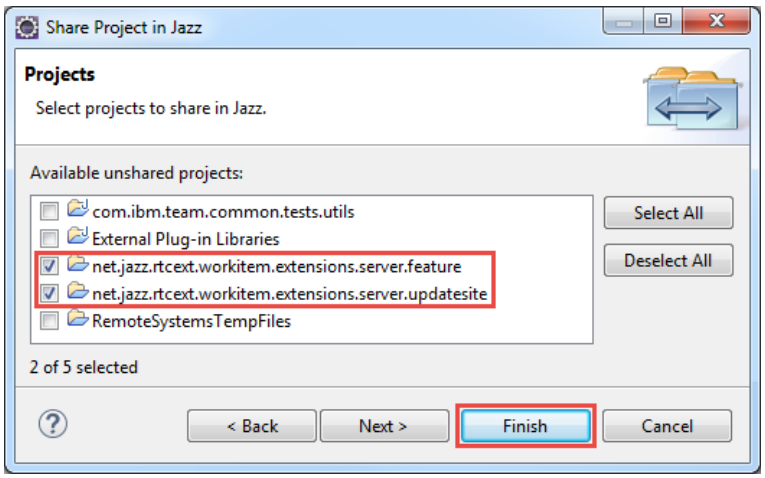


__c. On the second page of the wizard, select the **RTC Extension Server Workspace** (as highlighted with a red box) and click **New Component**.

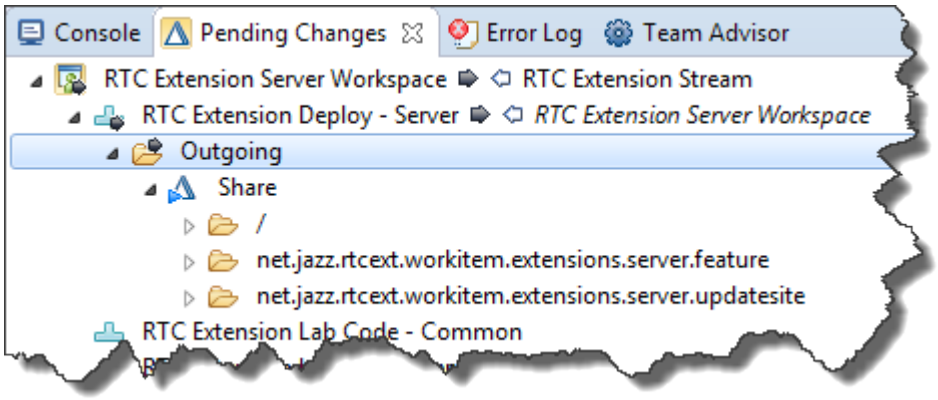
In the **New Component** dialog, enter `RTC Extension Deploy - Server` as the component name and click **OK**. Finally, back to the wizard, make sure the new component is selected (red arrow) and then click **Next**.



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

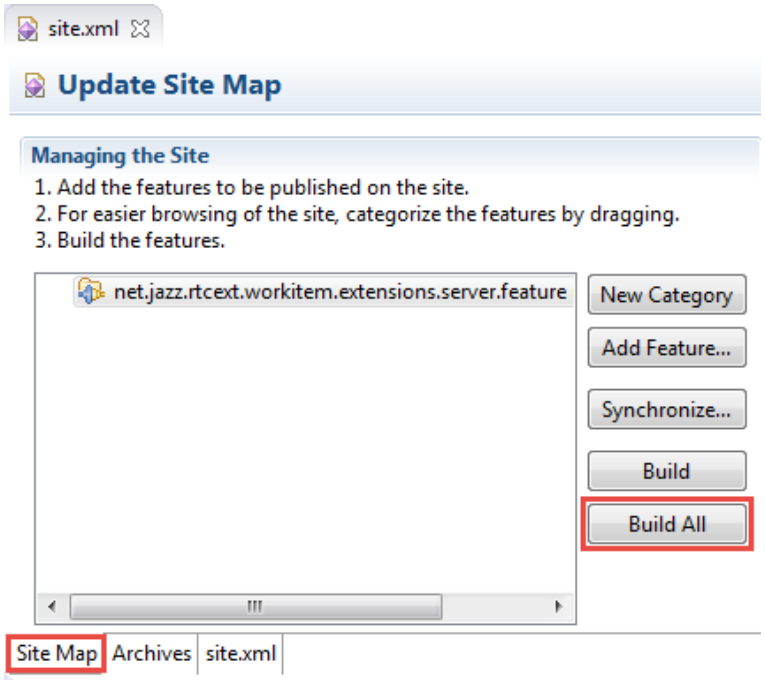


__e. The **Pending Changes** view will show your outgoing component addition with its newly shared projects. You will deliver them later.



__149. Build the update site.

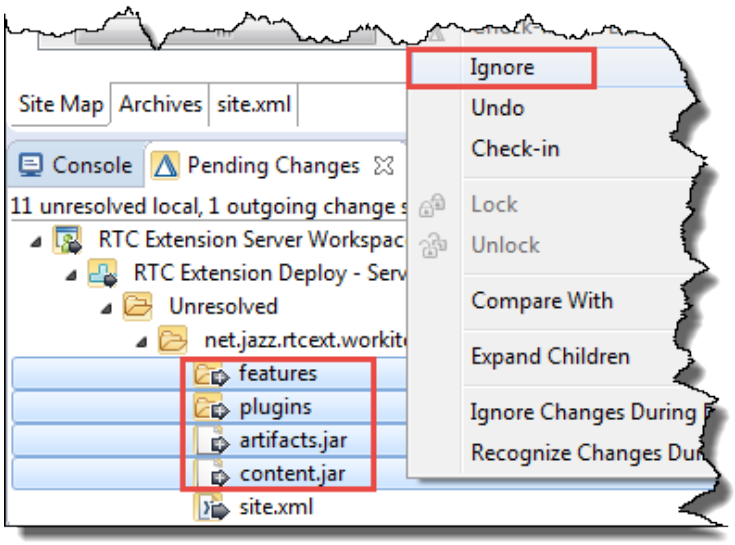
__a. Return to the site.xml editor and click **Build All**.



__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**.

__i. Select the four entries in the root of the update site as shown here (note that site.xml is not selected). 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.

Now right click one of the selected files 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.

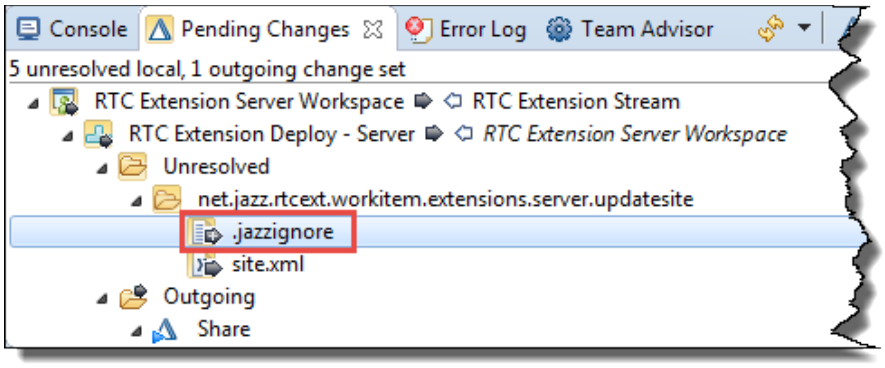


__ii. Note that the ***artifacts.xml*** and ***content.xml*** files are used for the new P2 style update sites. The jazz server side provisioning does not use them at this time. However, if you create an update site for the client side plug-ins, you can create a P2 enabled update site.

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.

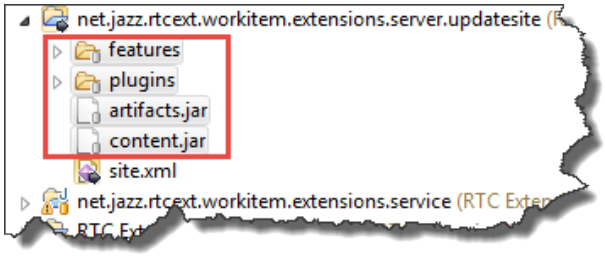
The site.xml file is both a build input and output when an update site is built. You want to build input version under source control, not the build output version.

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



__d. If you now dig into the site.xml file and into the JAR files 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 generated from the update site project's features and plugins folders. The build will generate new with different date and time stamps and leave the old ones there too. You can also just delete the files and folders you just ignored.

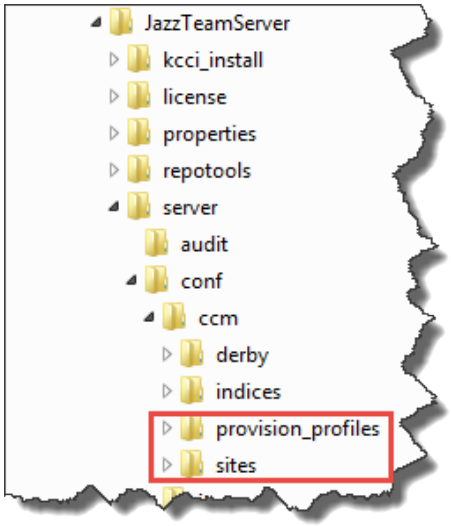


__e. You can close the editor open for the site.xml file.

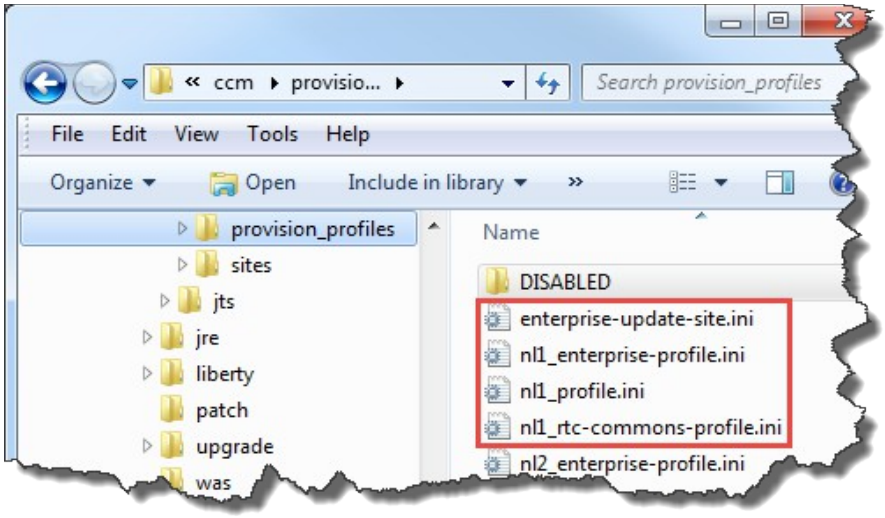
6.3 Create a Server Deploy project

__150. Review the structure of the deployment related files in the application server.

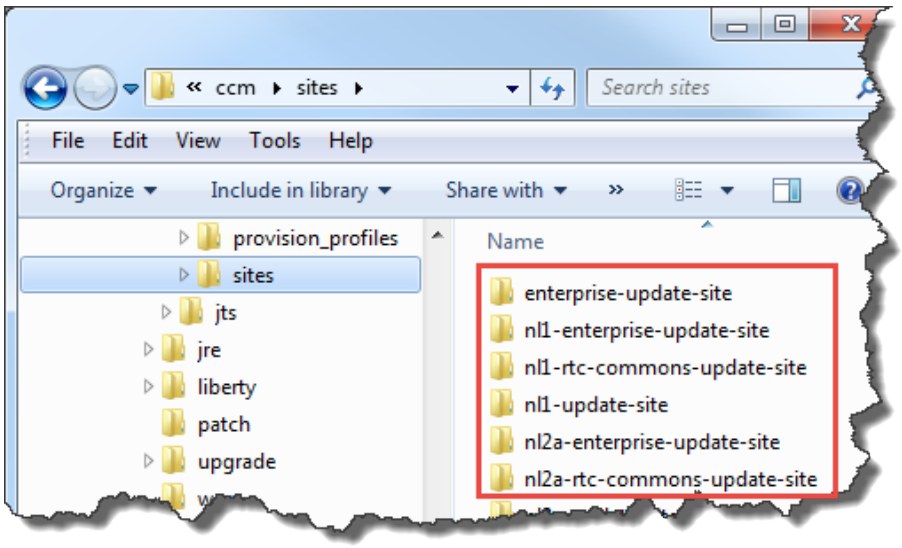
__a. Browse to the configuration folder of the CCM application representing RTC in the folder `C:\RTC603Dev\installs\JazzTeamServer\server\conf\ccm`. Browse the folders `provision_profiles` and the folder `sites`.



__b. The folder `provision_profiles` contains several provision profiles – files that describe how to provision features to the CCM server.



__i. The folder `sites` contains update site folders containing the `site.xml` and the update site feature and plug-in binaries that will be deployed.

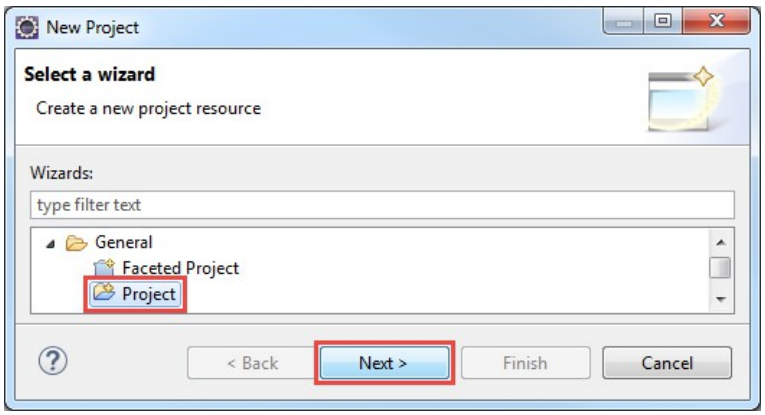


__151. Create the Server Deploy Project.

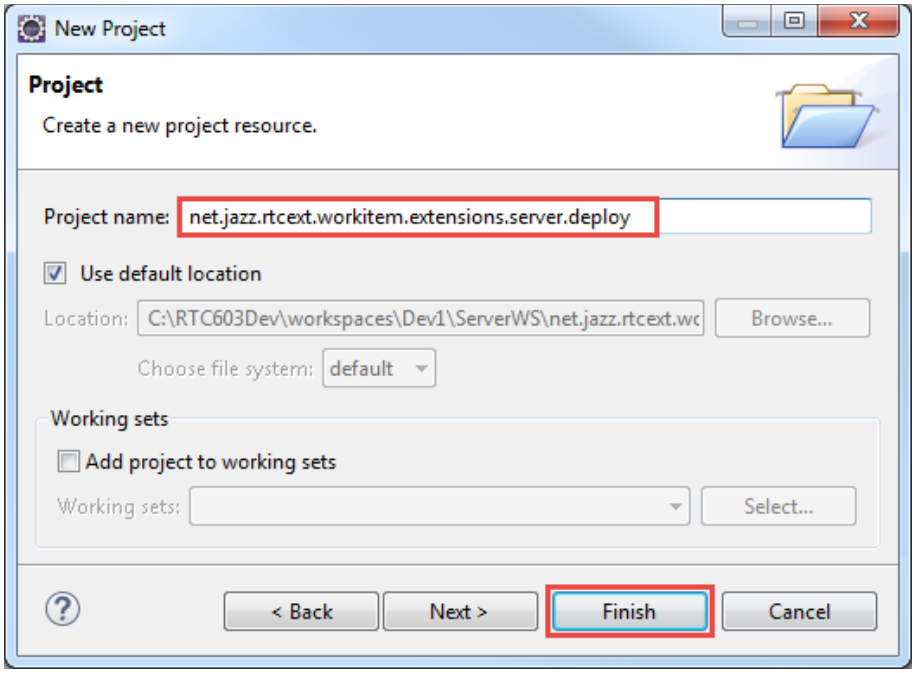
Deployment requires an update site folder that contains the site.xml as well as the feature and plug-in binaries you just created. It also requires a special provision profile, a file that describes the extension and points to the update site folder. The update site folder and the provision profile have related content and assume a certain structure. It is a good practice to manage this information under source control to make deployment repeatable. To be able to do this in Eclipse, a normal Eclipse project can be used.

Please note, packaging for deployment is done manually in this lab, it could be automated using ANT or other build tools in a production environment.

__a. From the menu bar, select **File > New > Project…** then in the **New Project** wizard, browse to General and select **Project** then click **Next.**

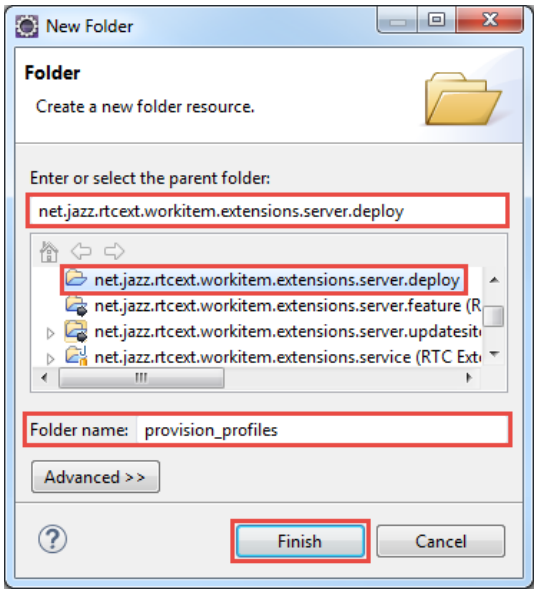


__b. On the second page of the wizard type `net.jazz.rtcext.workitem.extensions.server.deploy` into the **Project name** field. Click **Finish**.



__152. Create the provision profile and update site folders.

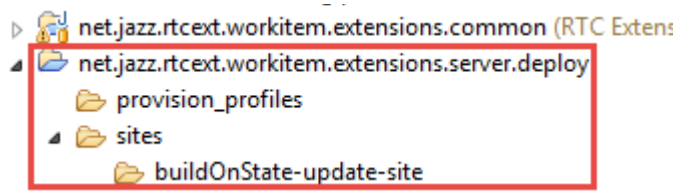
__a. From the menu bar, select **File > New > Folder…** then in the **New Folder** wizard, browse to the project `net.jazz.rtcext.workitem.extensions.server.deploy` (enter `net.jazz.rtcext.workitem.extensions.server.deploy` as parent folder), as folder name enter `provision_profiles` then click **Finish**.



__b. Repeat the steps above and create a folder named `sites` directly in the parent folder `net.jazz.rtcext.workitem.extensions.server.deploy`.

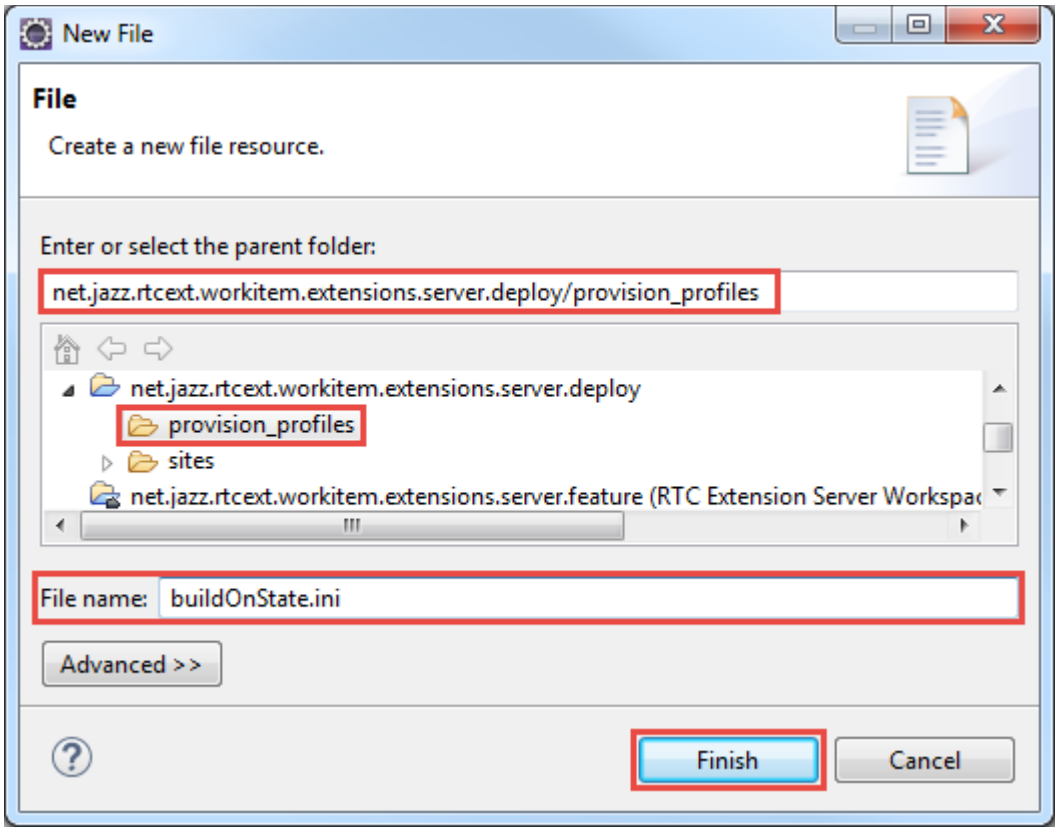
__c. Repeat the steps above and create a folder named `buildOnState-update-site` in the parent folder `net.jazz.rtcext.workitem.extensions.server.deploy/sites` you just created. This folder will later contain the features and plug-ins deployed on the server.

The project should now look like below.



__153. Create the provision profile

__a. From the menu bar, select **File > New > File…** then in the **New File** wizard, browse to the project `net.jazz.rtcext.workitem.extensions.server.deploy`, select `provision_profiles` as parent folder and enter `buildOnState.ini` as file name then click **Finish**.



__b. The new ini file should open in an editor, if not, open the new file and enter these two lines:

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

Update Site URL

The url property points to a relative path from the configuration folder e.g. JazzTeamServer\server\conf to the site folder for the new extension of the CCM server. The folder with its content will be created during the deployment later. The update site must contain the site.xml for the extension.

If the update site is missing or can not be found, e.g. due to a typo in the URL or missing files such as site.xml, deployment will fail and can cause crashes preventing the server to start.

Feature ID

The featureid property has to match the feature Id provided in the server feature project. A mismatch can cause deployment problems and crashes preventing the server to start.

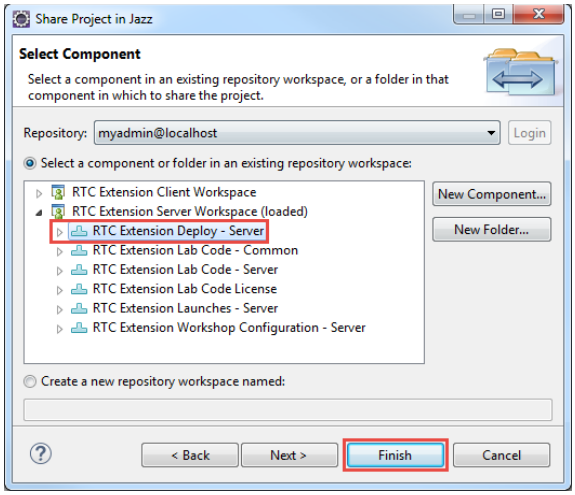
__c. Save the file and close the editor.

__154. Share the new project to your repository workspace.

__a. In the **Package Explorer** view, right click one the project `net.jazz.rtcext.workitem.extensions.server.deploy` and from the menu, select **Team > Share Project…**

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

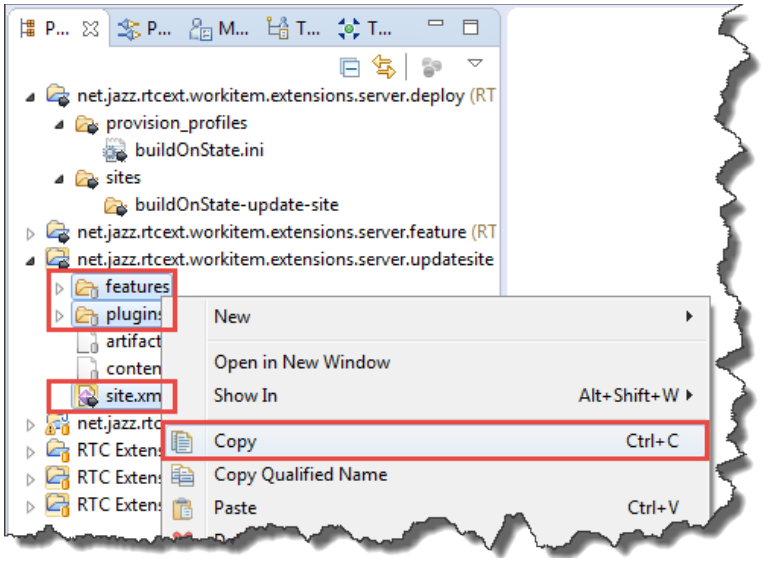
__c. On the second page of the wizard, select the component `RTC Extension Deploy - Server` in the **RTC Extension Server Workspace** and then click **Finish**.



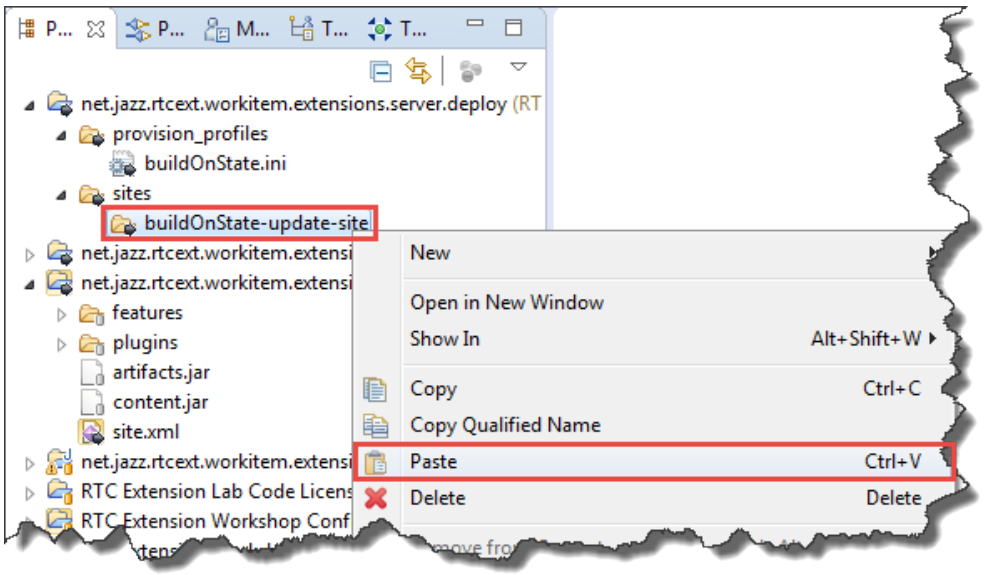
__155. Copy the server side update site files into the server deploy project.

The last step in creating the file structure needed to deploy the server side feature on the application server is to copy the `site.xml` and the required binaries into the server deploy project.

__a. In the Package Explorer expand the project `net.jazz.rtcext.workitem.extensions.server.updatesite`. Select the file `site.xml` and the two folders `plugins` and `features`. Right click on one of the selections and from the menu select **Copy**.



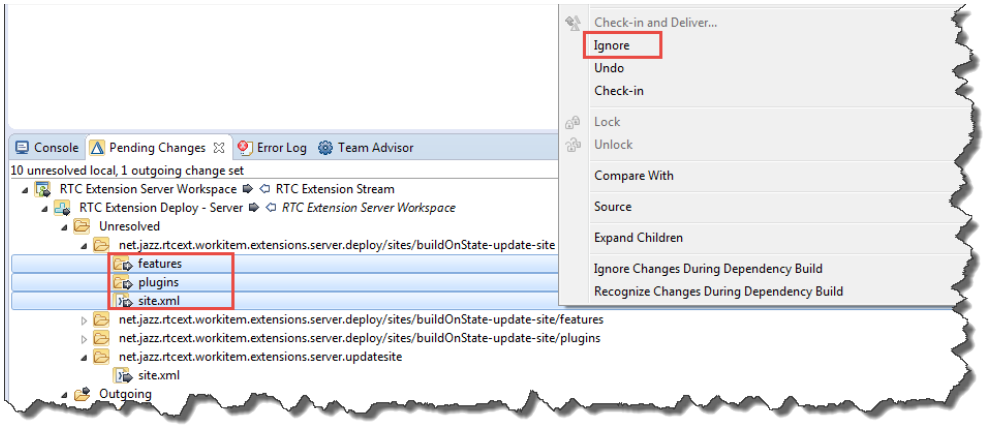
__b. In the Package Explorer expand the project `net.jazz.rtcext.workitem.extensions.server.deploy`. Right Click on the folder `/sites/buildOnState-update-site` and from the menu select **Paste**.



__c. The **Package Explorer** and **Pending Changes** views will show several new files in your server deploy project. In the Pending Changes view they will show up as **Unresolved**.

__i. Select the three entries in the folder buildOnState-update-site of the project as shown here. These files are used for deployment and have to be replaced with newer built versions if the source code changes. They do not need to be versioned under source control. This action along with the next sub-step will make sure you do not accidentally check them at another time.

Right click one of the selected files and select **Ignore** from the menu. 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.



__ii. The **Pending Changes** view will now shows a new **.jazzignore** file as **Unresolved**. Go ahead and check it in now by right clicking the file and then selecting **Check-in > Share** from the menu ("Share" is the name of the change set created when you shared the two projects into the RTC Extension Deploy component). The ignored files should have disappeared from the unresolved changes folder.

6.4 Deploy the Server extension

You are now ready to deploy the server side extension feature. This will be done in these major steps. You will request a server reset, to enforce a fresh re-read of the provision profiles, shut down the server, deploy the extension binaries, then restart the server and check the extension is deployed.

The request reset is technically only required when deploying a new version of an extension that has already been deployed, but it is a good practice to do it anyway.

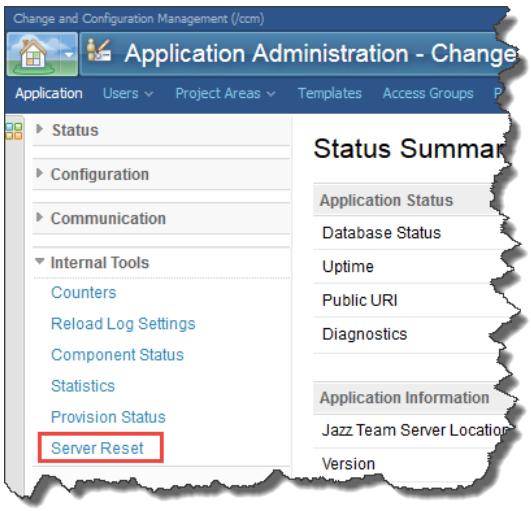
__156. Request a Server Reset

__a. If the development RTC server is not started start the server. Open a Windows Explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file.

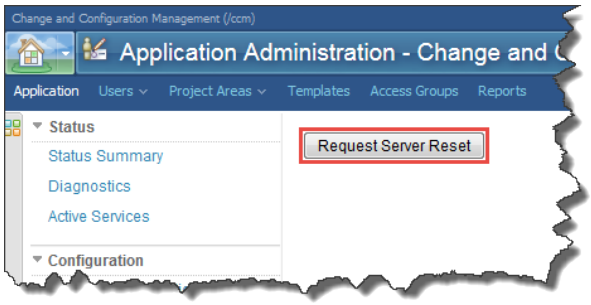
__b. Open a web browser and browser to the RTC Server admin page https://localhost:9443/ccm/admin.

__i. Log in, using myadmin as user id and password.

__ii. Append **?internal=true** to the admin page URL and open the resulting url https://localhost:9443/ccm/admin?internal=true. Shis shows the Internal Tools. Click on the server reset menu item to open the page https://localhost:9443/ccm/admin?internal=true#action=com.ibm.team.repository.admin.serverReset.



__iii. Click the Request Server Reset button. The server will be reset during the next server start.



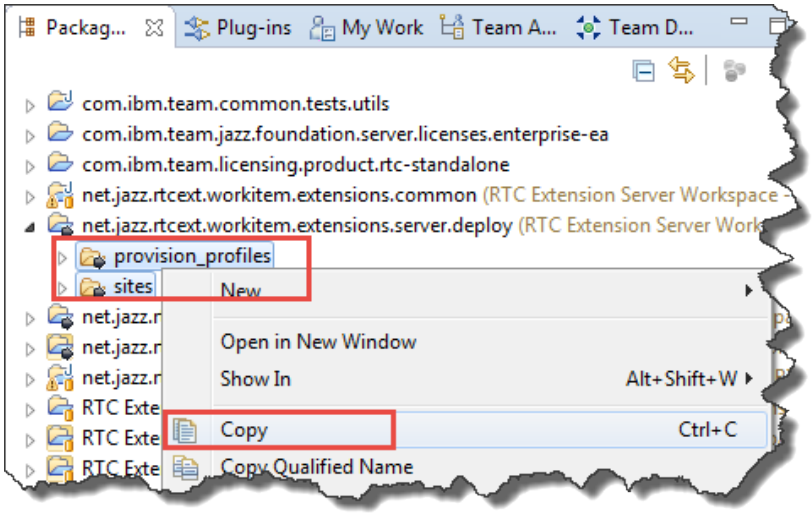
__157. Shutdown the RTC server to prepare deploying the server extension.

__a. Open a Windows Explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.shutdown.bat** file.

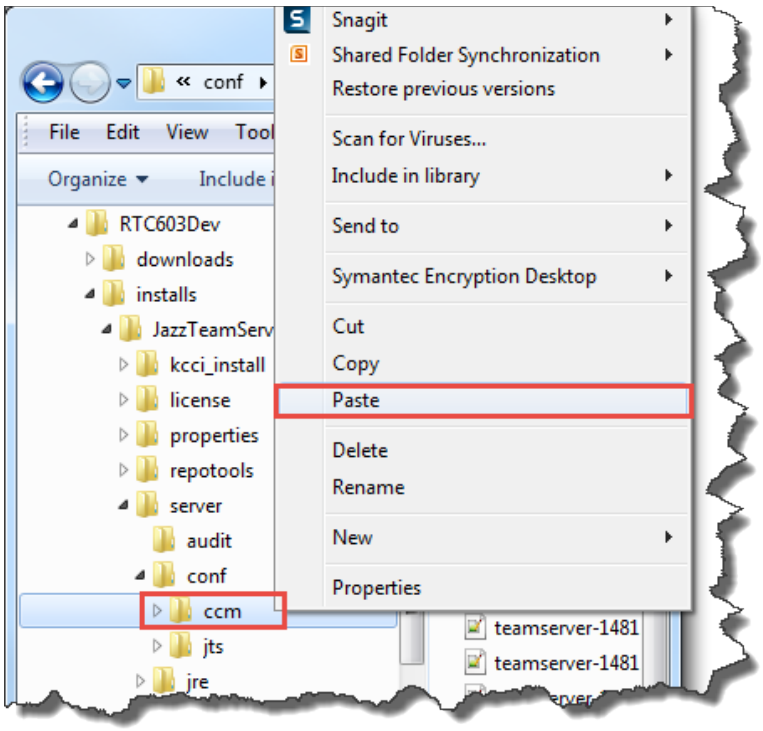
__158. Copy and paste the extension files to deploy the extension

__a. Return to the Eclipse client with the workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS`.

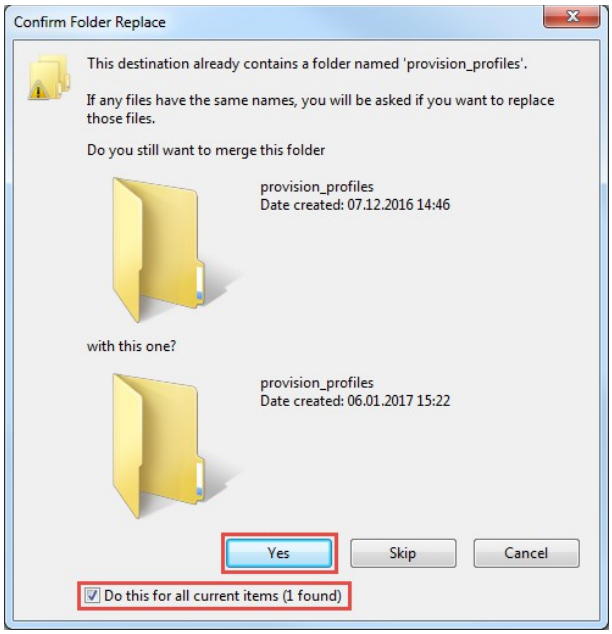
__b. In the package explorer browse to the server deploy project `net.jazz.rtcext.workitem.extensions.server.deploy`. Select the folders `provision_profiles` and `sites`, then right click on the selection and use the menu item **Copy** to copy the folders and contained files.



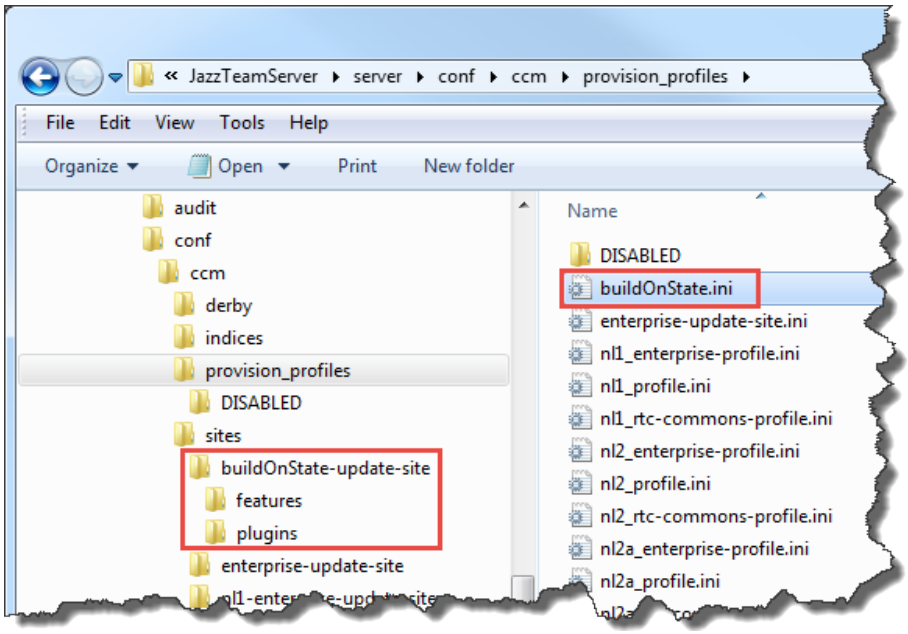
__c. Open a Windows Explorer, navigate to the configuration folder of the CCM application RTC `C:\RTC603Dev\installs\JazzTeamServer\server\conf\ccm`. Right click on the ccm folder and use the menu item **Paste**.



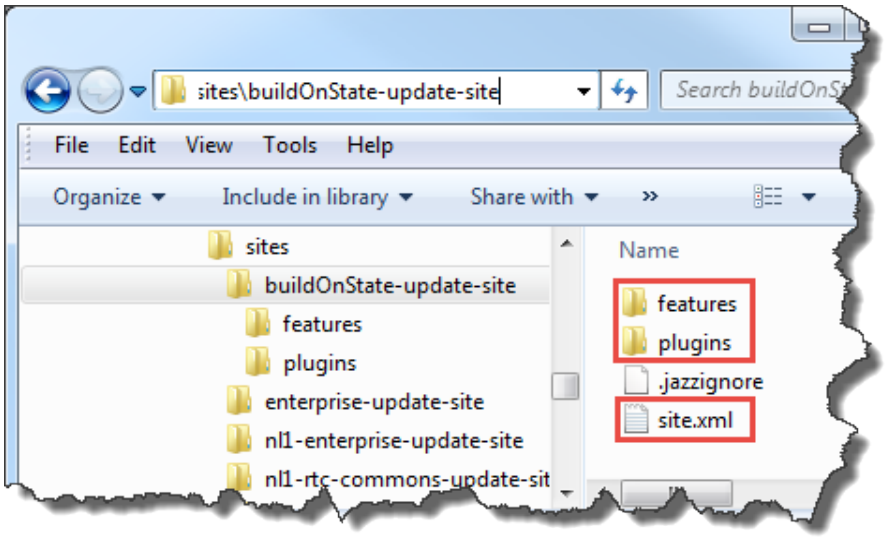
__i. The folders already exist. Confirm overwriting and continue pasting the folders and files into the existing structure.



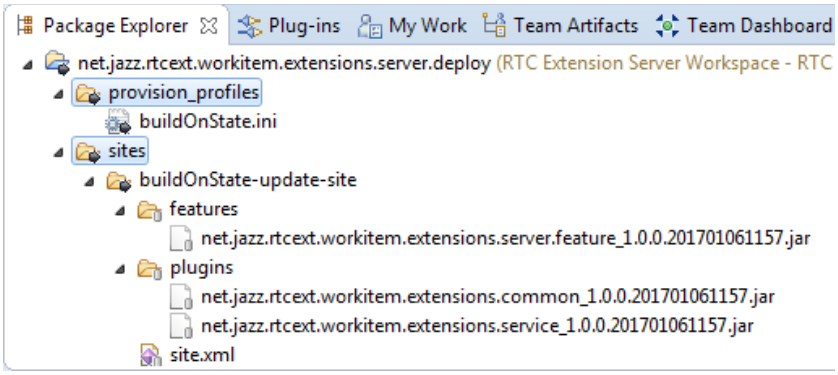
__ii. Verify that the provision profile buildOnState.ini is in the provision profiles folder along with the other provision profiles.



__iii. Verify that the buildOnState-update-site is located in the site folder along with the other update sites. Check that the site.xml and the folders feature and plugins are provided and contain the feature and plugin JAR files.



__a. The files copied must be the ones from the deploy project in Eclipse.

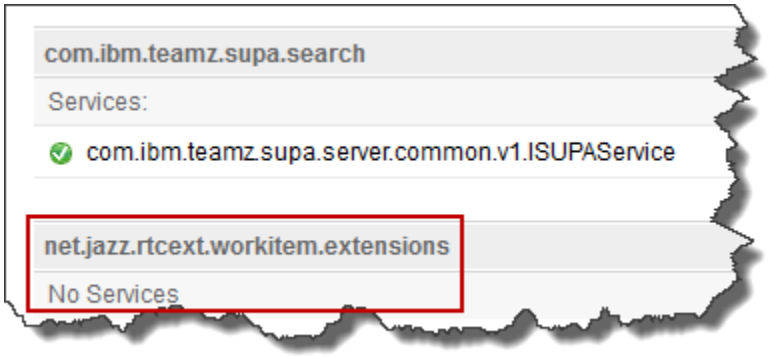


__159. Start the RTC server.

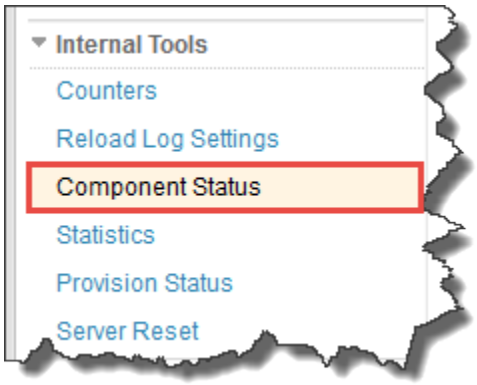
__a. Open a Windows Explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file. The startup can take considerably longer because the server has to redeploy all extensions.

__b. When the server is started browse to this URL https://localhost:9443/ccm/admin#action=com.ibm.team.repository.admin.componentStatus, login as `myadmin / myadmin`.

__i. 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.



__ii. Note that the Component Status can also be found from the internal tools that were used to request a server reset: https://localhost:9443/ccm/admin?internal=true



6.5 Understanding Deployment Problems

During deployment of Extensions a lot of things can, and very often do go wrong. This can lead to the extension not working or to the RTC server not starting at all.

Finding reasons for deployment problems

It is not easy to find reasons for deployment problems and how to fix them. This is mainly due to the fact that there is a declarative part using XML which can not be debugged. The problem only surfaces on run time, when the extension gets deployed. This section should give you some basic understanding where to look and what to look for.

Some typical issues that come up often:

- Typos in the provision profile leading to not being able to find the update site
- Incorrect data in the update site
  - Missing `site.xml` file
  - Missing feature and plug-in JAR files
  - Corrupted feature and plug-in JAR files e.g. built on Windows and extraction fails on other operating system
- Wrong or missing dependencies in the feature and plug-ins such as dependencies to a client API plug-in or to other plug-ins not being available in the server API or mismatches of the required versions

__160. You will now create a deployment problem, observe the effects and fix the problem.

__a. If the development RTC server is not started, start the server. Open a Windows Explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file.

__b. Request a server reset

__i. Open a web browser and browser to https://localhost:9443/ccm/admin?internal=true#action=com.ibm.team.repository.admin.serverReset.

__a. Log in, using `myadmin` as user id and password.

__b. Click the **Request Server Reset** button.

__c. Stop the server

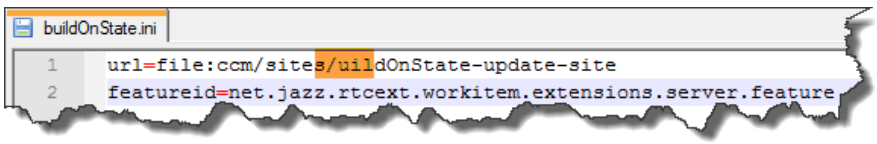
__i. Open a Windows Explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.shutdown.bat** file.

__ii. Wait for the server to stop.

__d. Introduce a defect in the provision profile

__i. Open a Windows explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server\conf\ccm\provision_profiles` and open the file `buildOnState.ini` with a text editor.

__ii. Introduce a typo in the URL. E.g. add a letter or remove a letter like below where the leading “b” was removed from `buildOnState-update-site` and save the change.



__e. Start the server

__i. Open a Windows explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file.

__ii. Wait for the server to start.

__f. Connect to the server

__i. Try to check the component status of the extension by browsing to this URL https://localhost:9443/ccm/admin#action=com.ibm.team.repository.admin.componentStatus or try any other CCM related URL.

__ii. The server does respond with an error or does not respond at all and the browser times out.

__g. Check the log files.

__i. Open a Windows explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server\liberty\servers\clm\logs` and open the file ccm.log.

__ii. Search for the string “ProvisionService” and any other hints for a problem.

__iii. In the case of the incorrect update site location an error like the below should come up indicating the update site can not be found.

```
2017-01-09 11:59:34,242 [       Launch callback handler] ERROR
eam.repository.provision.internal.ProvisionService  - CRJAZ0288E
The "net.jazz.rtcext.workitem.extensions.server.feature" profile
feature could not be installed from the
"file:ccm/sites/buildOnState-update-site_" update site that is
referenced in the
"C:\RTC603~2\installs\JAZZTE~1\server\conf\ccm\provision_profiles\b
uildOnState.ini" profile file.
2017-01-09 12:06:37,429 [       Launch callback handler] ERROR
eam.repository.provision.internal.ProvisionService  - CRJAZ0285I
Failed to connect to "file:ccm/sites/uildOnState-update-site".
java.io.FileNotFoundException:
C:\RTC603~2\installs\JAZZTE~1\server\conf\ccm\sites\uildOnState-
update-site\site.xml (The system cannot find the path specified)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:158)
        at java.io.FileInputStream.<init>(FileInputStream.java:113)
        at
sun.net.www.protocol.file.FileURLConnection.connect(FileURLConnecti
on.java:104)
        at
sun.net.www.protocol.file.FileURLConnection.getInputStream(FileURLC
onnection.java:202)
        at java.net.URL.openStream(URL.java:1061)
```

There are several clues for example C:\RTC603~2\installs\JAZZTE~1\server\conf\ccm\sites\uildOnState-update-site\site.xml (***The system cannot find the path specified***) and others like highlighted above.

__161. Fix the deployment

__a. Stop the server

__i. Open a Windows explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.shutdown.bat** file.

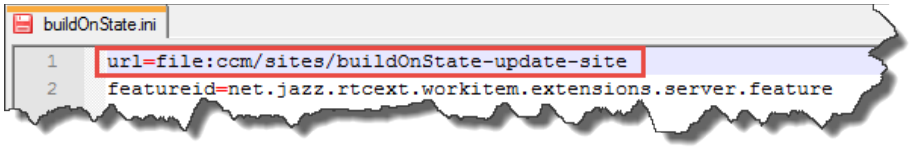
__ii. Wait for the server to stop.

__b. Fix the typo

__i. Open a Windows explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server\conf\ccm\provision_profiles` and open the file `buildOnState.ini` with an editor.

__ii. Fix the first line. The content should again be as below:

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



Save the change.

__c. Start the server. Since the server did not start at all, a request reset is not necessary.

__i. Open a Windows explorer, navigate to `C:\RTC603Dev\installs\JazzTeamServer\server` and run the **server.startup.bat** file.

__ii. Wait for the server to start.

__d. Log into the CCM server.

__i. Check for the component status and browser to this URL https://localhost:9443/ccm/admin#action=com.ibm.team.repository.admin.componentStatus. Login using `myadmin` as user id and password.

__ii. You should now be able to log in successfully.

Reset the server if server does not start?

If the server does not start anymore e.g. due to a faulty extension deployment a request reset using the Web UI is impossible.

Another mechanism available to enforce a server reset is to manually delete the file **`built-on.txt`**. This file is automatically generated when the server loads and initially reads and deploys the provision profiles. The file contains the time stamp of the last provisioning.

The file **`built-on.txt`** can be found in temporary folders of the application server. Where the file is located depends on the application server it is however easy enough to find it **using file search**.

For the workshop the file can be found in a location like this:
`C:\RTC603Dev\installs\JazzTeamServer\server\liberty\servers\clm\workarea\org.eclipse.osgi\65\data\temp\default_node\SMF_WebContainer\ccm\ccm\built-on.txt`

Note, the file will not be available after a server reset request as it gets deleted due to the reset request.

Also see this post for more information.

6.6 Create a Feature for the RTC Eclipse client extension

In the past it was possible to basically just export the client plug-in into the dropins folder to test it. This does not work with Eclipse Luna (4.4.2). It is at least necessary to create a feature similar to the steps for the server deployment above.

__162. Return to the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS` from the last lab.

__a. If your RTC Client SDK development environment is not open, start Eclipse. Navigate to `C:\RTC603Dev\installs\TeamConcert\eclipse` in the Windows explorer and double click **eclipse.exe**.

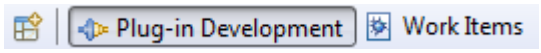
__i. When prompted, select the Eclipse workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS`. Don't check the “Use as default” check box.

__b. Check the desired Eclipse workspace is used.

__c. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view in the **Work Items** perspective and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.

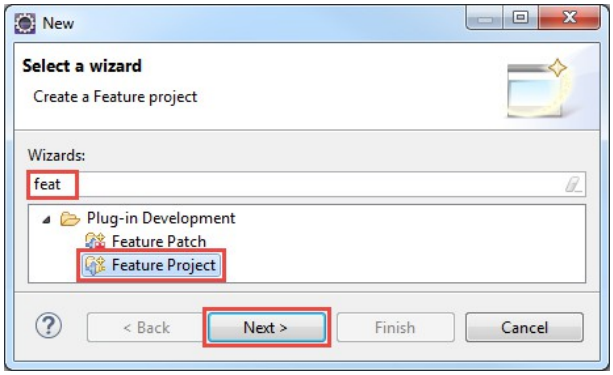
__163. Create a Feature Project for the RTC Eclipse client extension. The feature is used to collect all required plug-ins for the Build On State change Eclipse client Aspect Editor extension.

__a. Switch to the **Plug-in Development perspective**. In the toolbar toward the right, click **Plug-in Development** to switch the perspective.

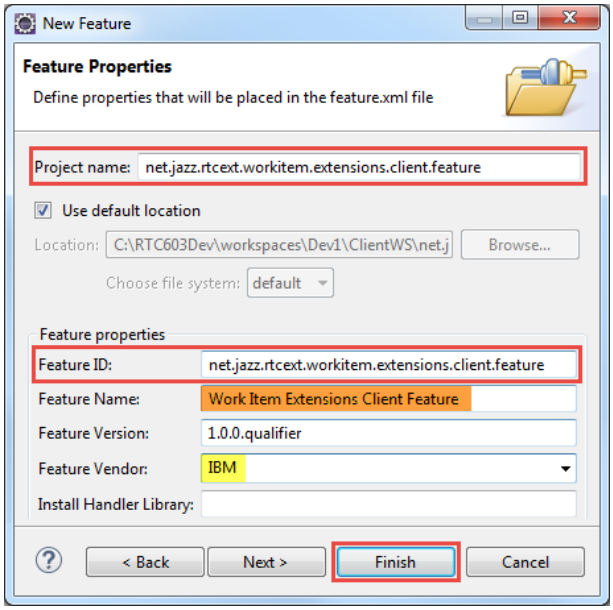


__i. If the **Plug-in Development** perspective is not available, open it by selecting **Window > Open Perspective > Other… > Plug-in Development** from the menu bar.

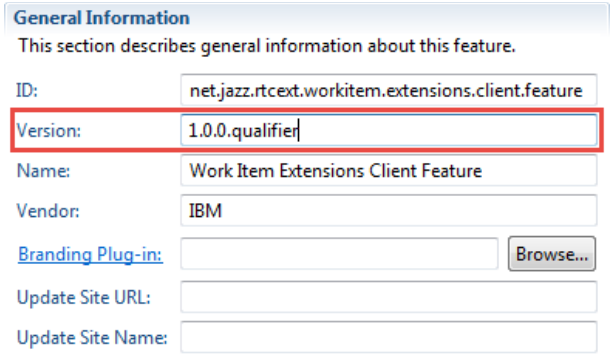
__b. From the menu bar, select **File > New > Project…** then in the **New Project** wizard, type `feature` in the filter field, select **Feature Project** from the list and then click **Next**.



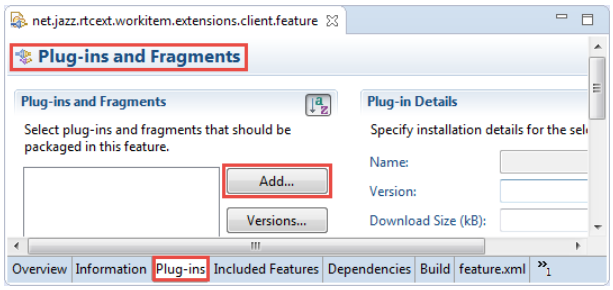
__c. On the second page of the wizard type `net.jazz.rtcext.workitem.extensions.client.feature` into the **Project name** field. As you type, the **Feature ID** is set to a reasonable value but the **Feature Name** should be set to a more descriptive name like `Work Item Extensions Client Feature`. You can set the **Feature Vendor** to yourself or your company, if you wish. It is not required. Click **Finish**.



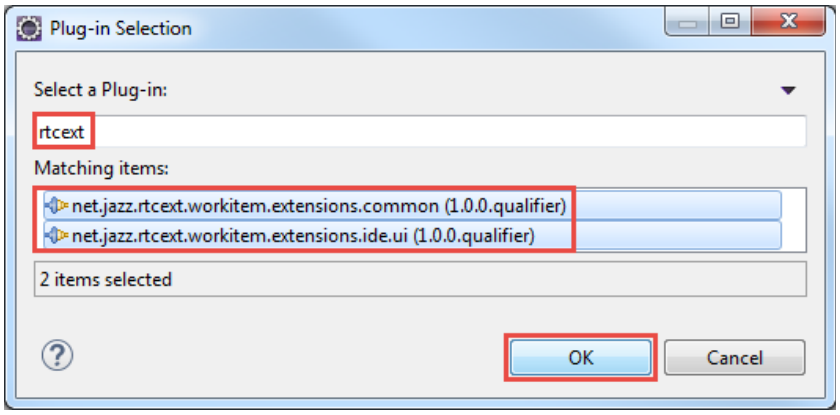
__d. Your new feature project appears in the **Package Explorer** view and an editor opens on the `feature.xml` file. On the **Overview** tab, make sure the **Version** is set to `1.0.0.qualifier`. This is the same Eclipse best practice you used for the plug-ins.



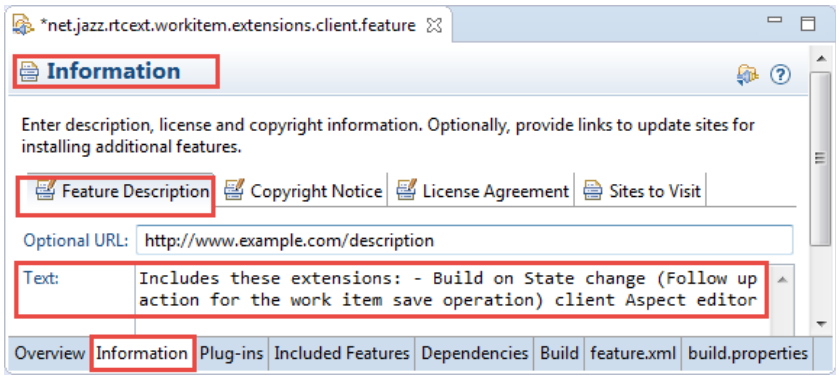
__e. The feature currently does not contain any plug-ins, you will add them now. In the editor, switch to the **Plug-ins** tab. In the section **Plug-ins and Fragments** click the **Add...** button.



__f. In the Plug-in Selection dialog type"rtcrext" to filter for your plug-ins. Select the plug-ins ***net.jazz.rtcext.workitem.extensions.common*** and ***net.jazz.rtcext.workitem.extensions.ide.ui*** and then press **OK**.

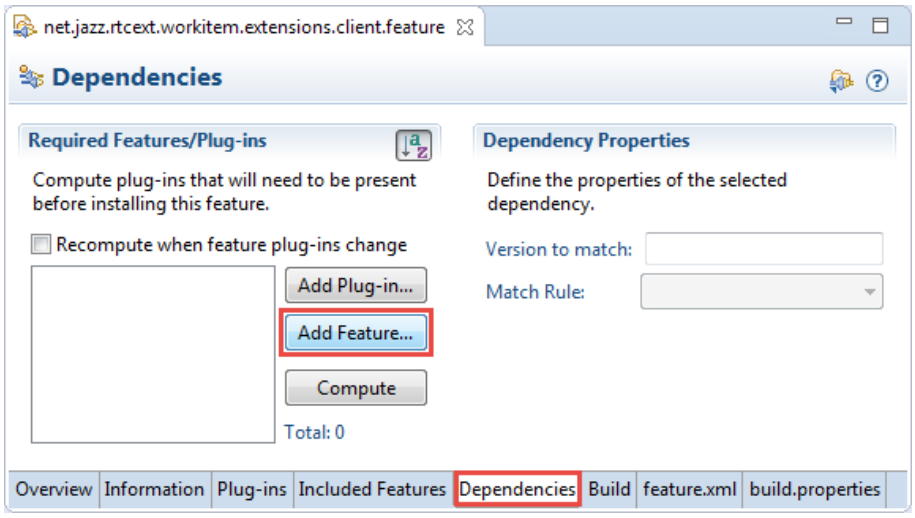


__g. Still in the editor, switch to the **Information** tab, select the **Feature Description** sub-tab and enter a **Text** description as shown here. If you wish you can look at other information that can be added, such as a copyright and license information.

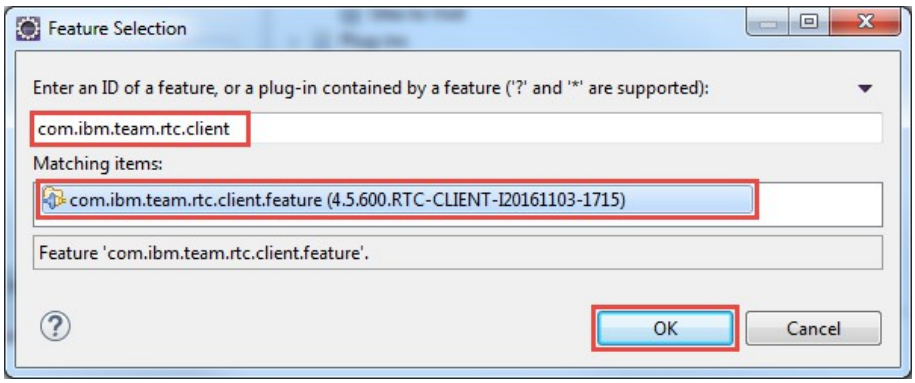


__h. Type **Ctrl+S** to save the `feature.xml` file.

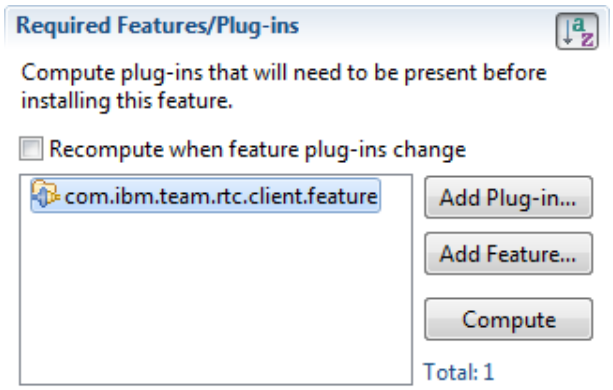
__i. It is important to add the dependencies that are needed to the feature and you will do this in the next steps. The RTC Eclipse client does not require the complex dependency management the server needs. Our client feature basically requires RTC to be installed. Switch to the **Dependencies** tab and click **Add Feature**.



__j. In the **Feature Selection** dialog type `com.ibm.team.rtc.client` into the filter field, select the feature **com.ibm.team.rtc.client.feature** and then click **OK**.



__k. The list will now look like this. The client extension feature requires the RTC client feature to be installed and work. Type **Ctrl+S** to save the `feature.xml` file. You can now close the editor.

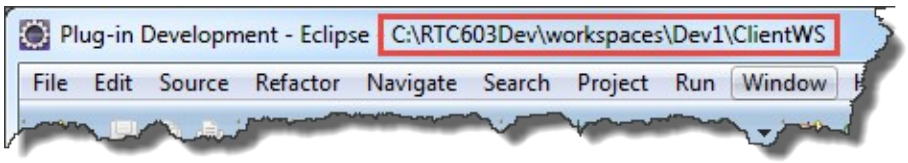


6.7 Export the RTC Eclipse client extension Feature for deployment

__164. Export the client feature.

__a. Return to the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS` from the last lab.

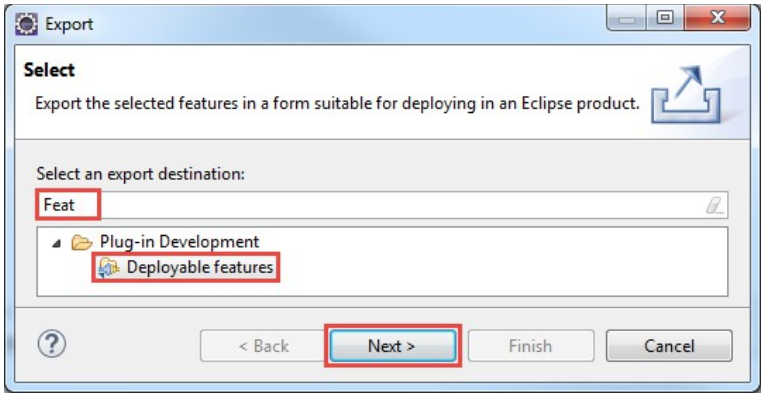
__i. Check the desired Eclipse workspace is used.



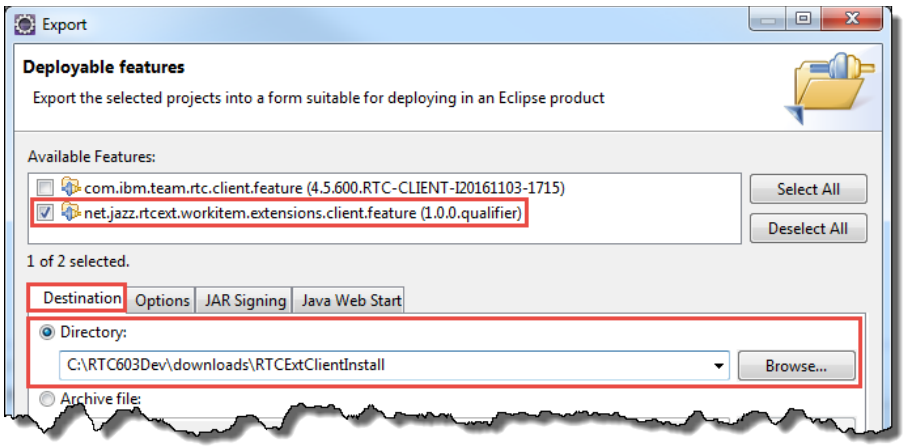
__b. Switch to the **Plug-in Development perspective**. In the toolbar toward the right, click **Plug-in Development** to switch the perspective.

__c. In the **Package Explorer** view, select the client feature `net.jazz.rtcext.workitem.extensions.client.feature` you just created. Right click the selection and select **Export...**

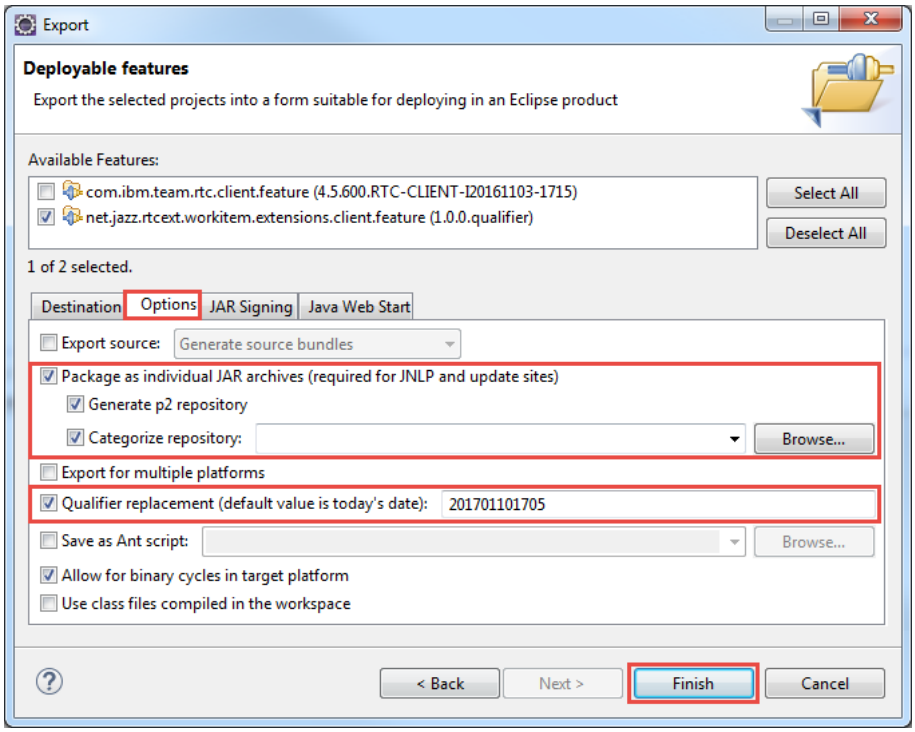
__d. In the **Export** wizard, type **Feat** in the filter, select **Deployable Features** from the list and then click **Next**.



__e. Make sure the client feature `net.jazz.rtcext.workitem.extensions.client.feature` and nothing else is selected. On the **Destination** tab select Directory and enter `C:\RTC603Dev\downloads\RTCExtClientInstall`.



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



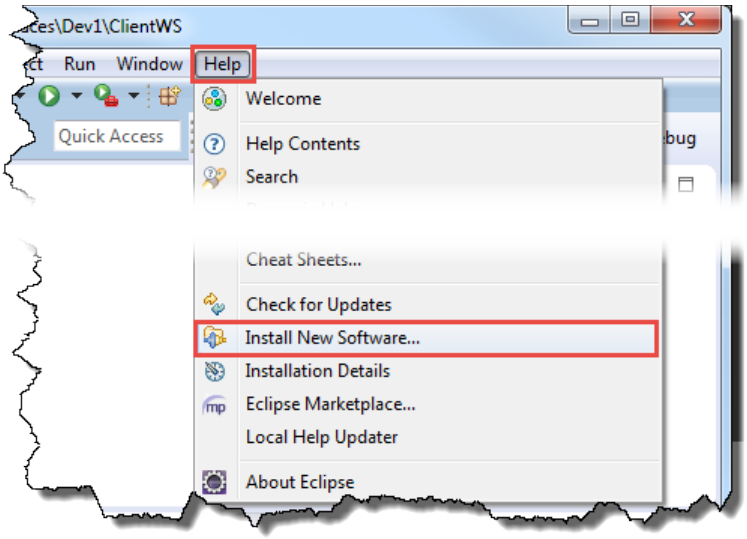
6.8 Deploy the RTC Eclipse client extension Feature

__165. Install the RTC Eclipse client feature extension.

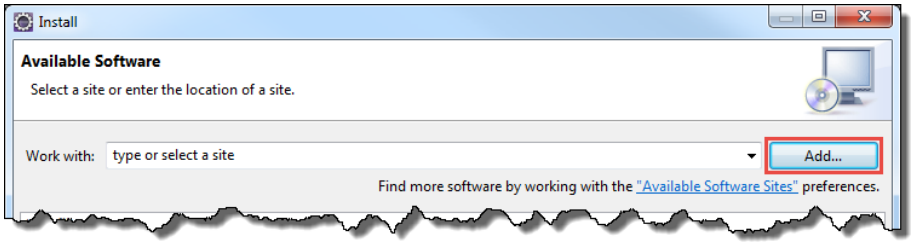
__a. Return to the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS` from the last lab.

__i. Check the desired Eclipse workspace is used.

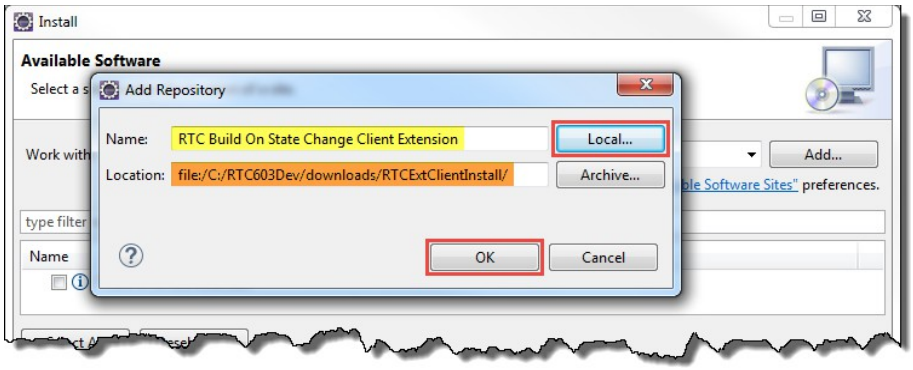
__b. From the menu bar, select **Help > Install New software...**



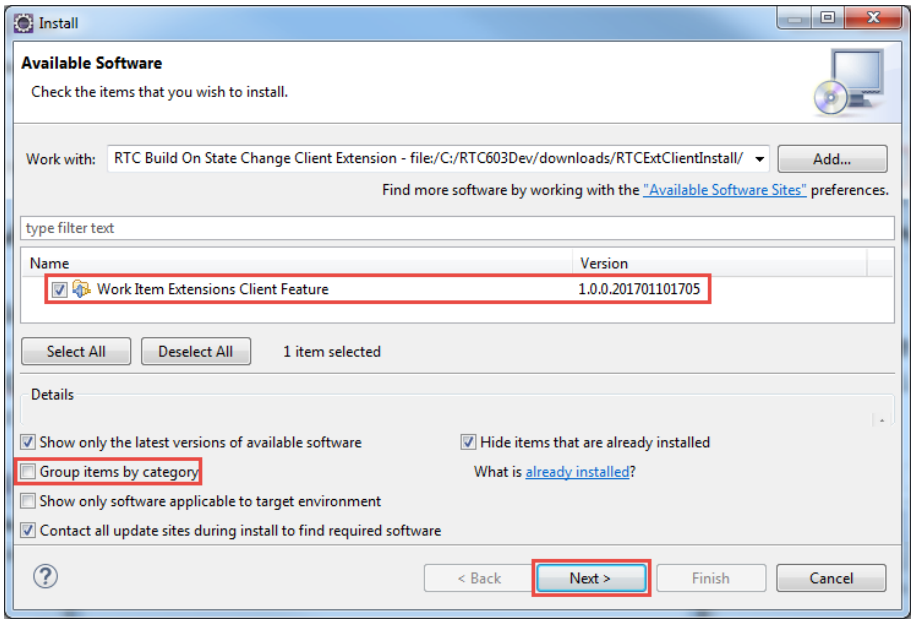
__i. **T**hen in the **Install** wizard, click **Add...** to add a site.



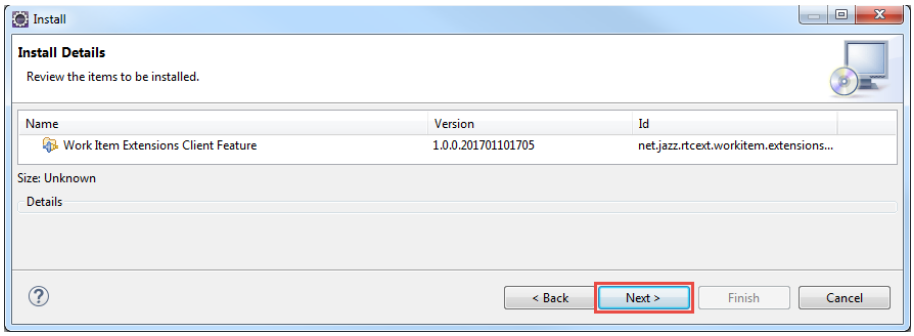
__ii. Click **Local...** and browse to the location `C:\RTC603Dev\downloads\RTCExtClientInstall` used during the previous export. Optional enter a name like `RTC Build On State Change Client Extension` for the location, then click OK



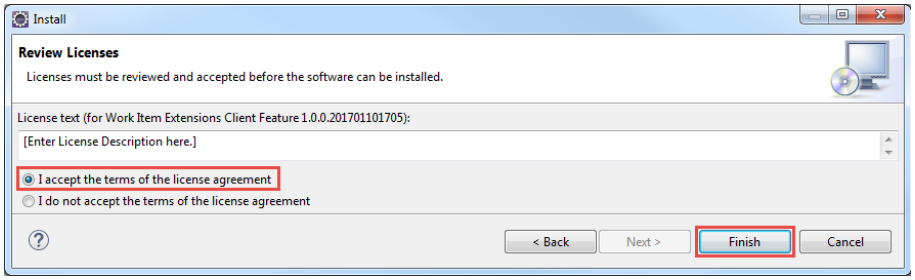
__iii. Uncheck **Group items by category**, then check the appearing **Work Item Extensions Client Feature** and click **Next**.



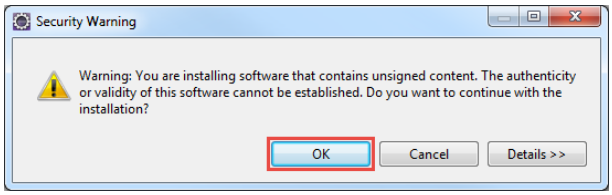
__iv. Wait while the requirements are calculated then click **Next**.



__v. Review and accept the license agreement then click **Finish**.

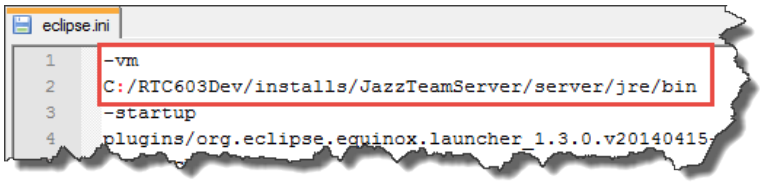


__vi. If a security warning comes up, because you did not sign your content, click **OK** to continue.



__vii. Wait for the install process to finalize and then press OK to restart the Eclipse client.

__viii. If Eclipse won't start, use a text editor and open the file `C:\RTC603Dev\installs\TeamConcert\eclipse\eclipse.ini`. Make sure the VM you specified in lab 1 is still set.



__ix. If not, add the following lines at the top of the `eclipse.ini` file:

```
-vm
C:/RTC603Dev/installs/JazzTeamServer/server/jre/bin
```

__x. Restart the Eclipse client if you had to provide the VM again.

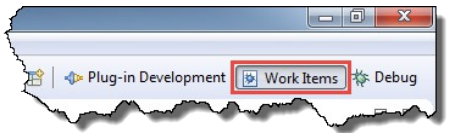
6.9 Test the Deployed RTC Eclipse client extension

__166. Return to the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS` from the last lab.

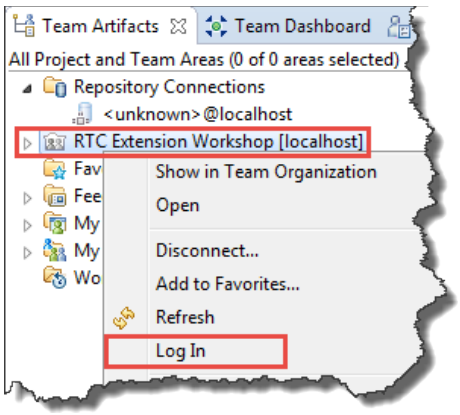
__a. Check the desired Eclipse workspace is used.

__167. 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.

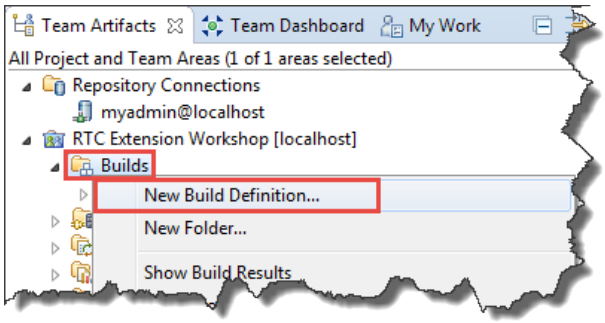
__a. Open the **Work Items** perspective.



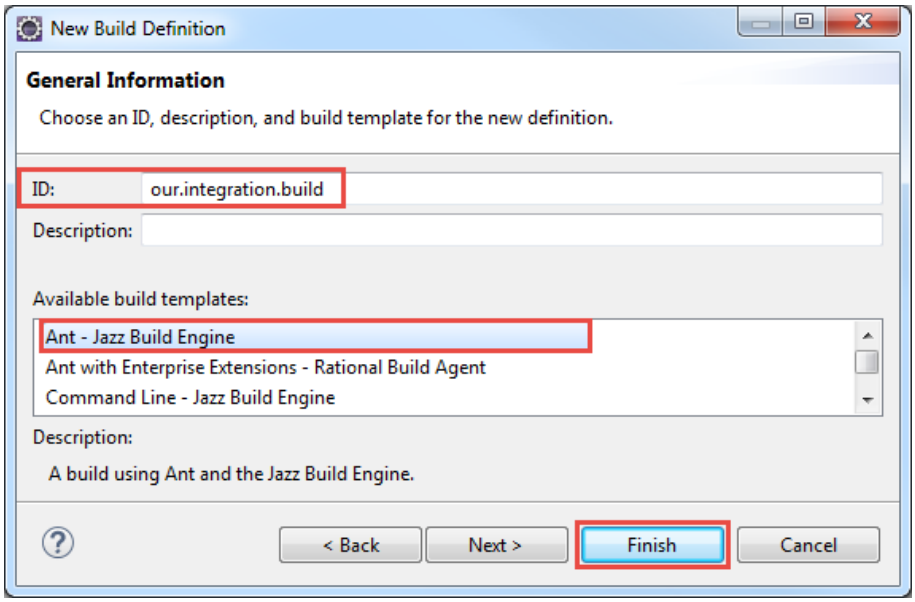
__b. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.



__c. Expand the **RTC Extension Workshop** node, right click **Builds** and then click **New Build Definition…**

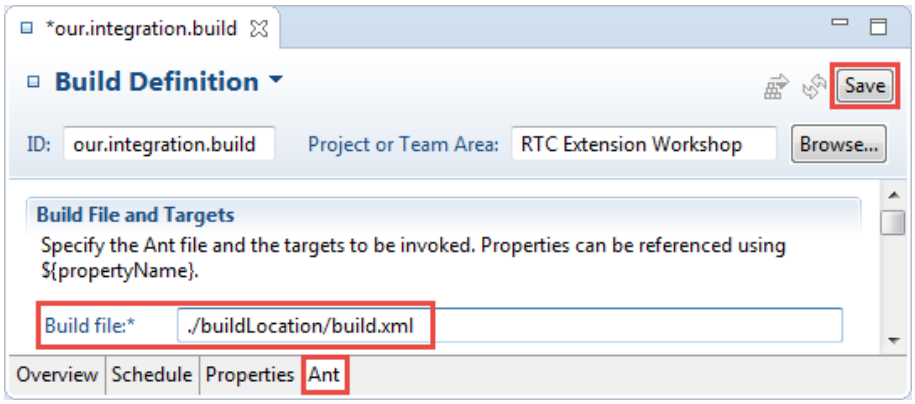


__d. In the **New Build Definition** wizard, make sure **Create a new build** is selected and then click **Next**. On the second page of the wizard, change the **ID** to `our.integration.build`, make sure **Ant - Jazz Build Engine** is selected and then click **Finish**.



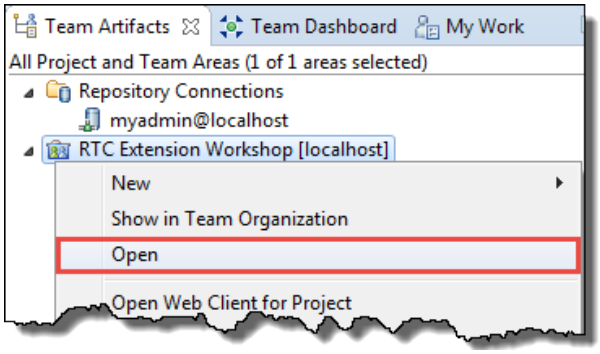
__e. In the build definition editor that opens, switch to the **Ant** tab, and enter a path for the **Build** file and then click **Save**. You may now close the editor.

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 here `./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.

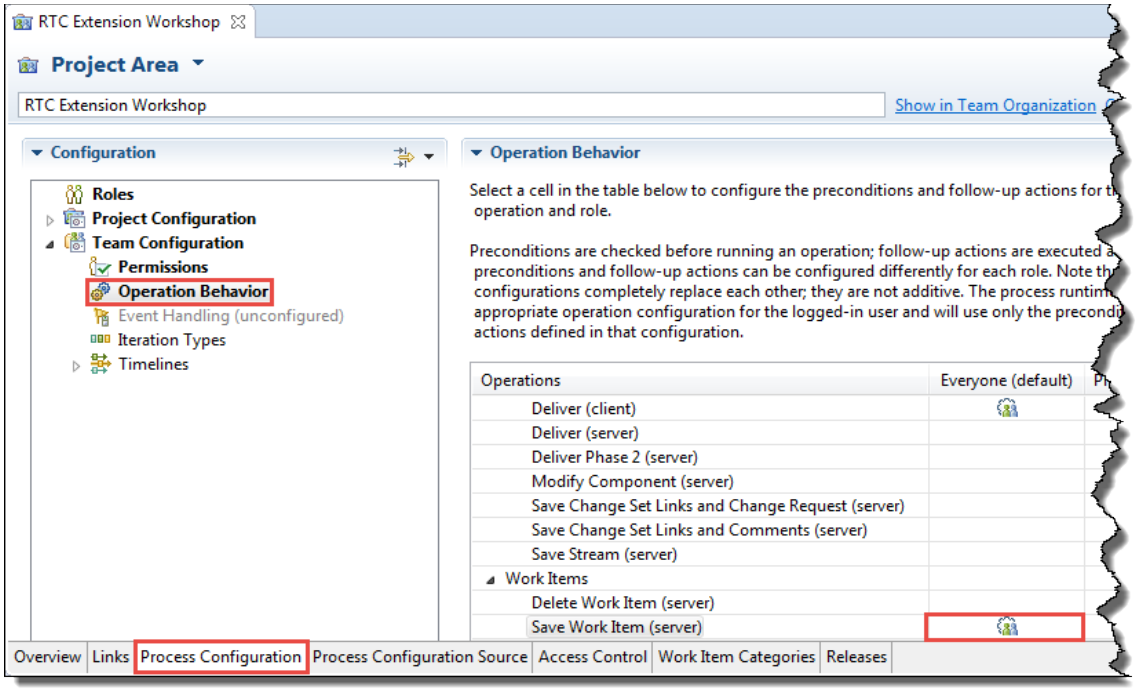


__168. Add the follow-up action to the project area.

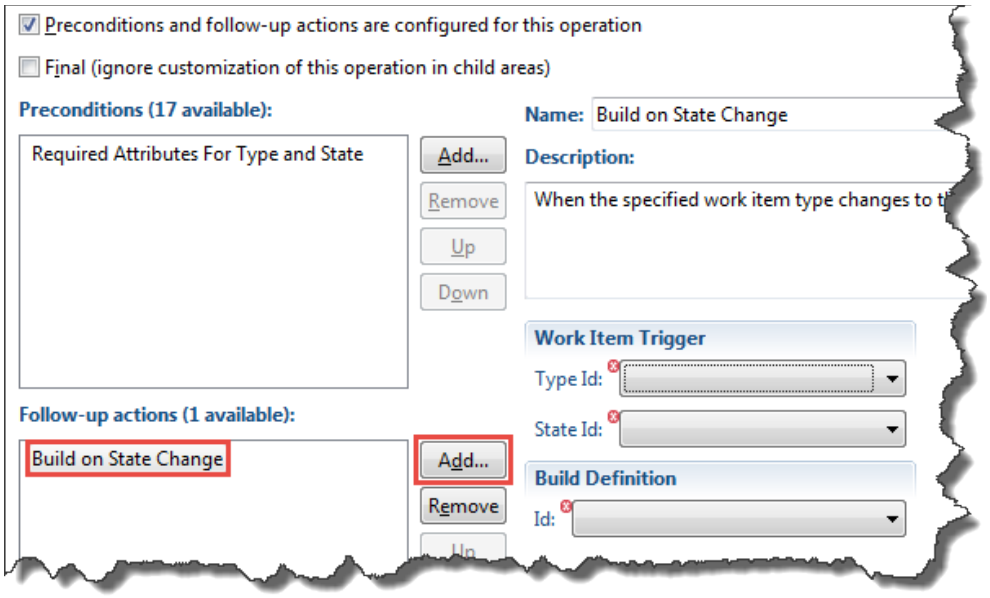
__a. In the **Team Artifacts** view, right click the **RTC Extension Workshop** project area and select **Open** from the menu.



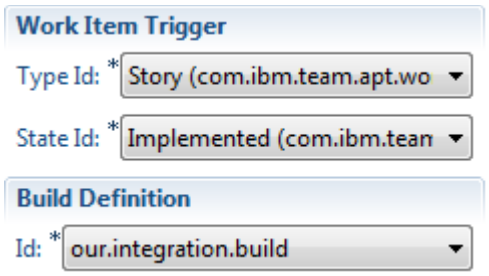
__b. In the project area editor, switch to the **Process Configuration** tab and then on the left, expand the **Team Configuration** tree then select **Operation Behavior**. Then, on the right, scroll down to the **Work Items > Save Work Item (server)** operation and select the **Everyone (default)**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



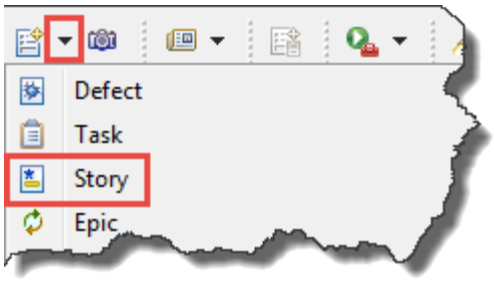
__d. Fill in the **Work Item Trigger** as shown here. You may, of course, choose different values for the work item type and state, but then you will need to adjust the following steps accordingly.



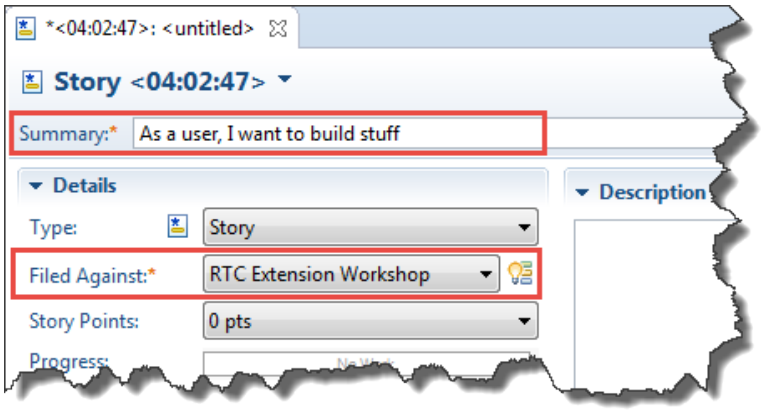
__e. Click **Save** at the top right of the editor. You may now close the project area editor and any other editors that may still be open.

__169. Create a Story and move it to the target state.

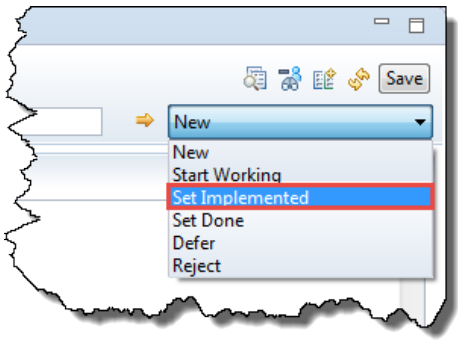
__a. Click the dropdown menu arrow next to the **New Work Item** toolbar icon and then click **Story**.



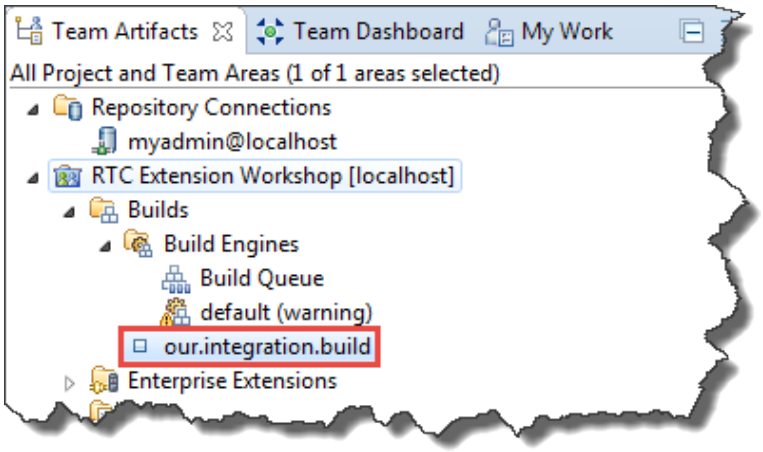
__b. In the new work item editor that opens, set the two required fields and shown here and then click **Save** in the upper right corner.



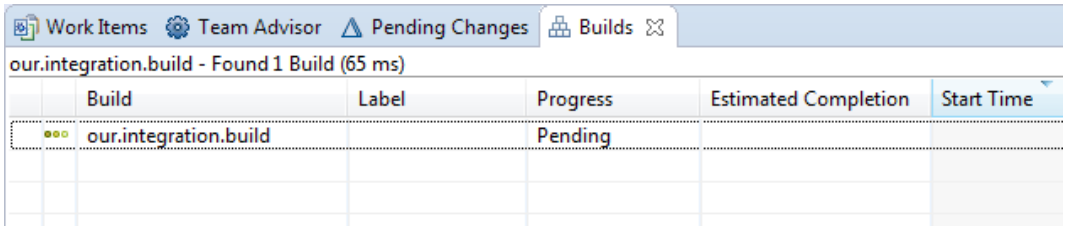
__c. At the upper right portion of the work item editor, select **Set Implemented** and then click **Save**.



__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.



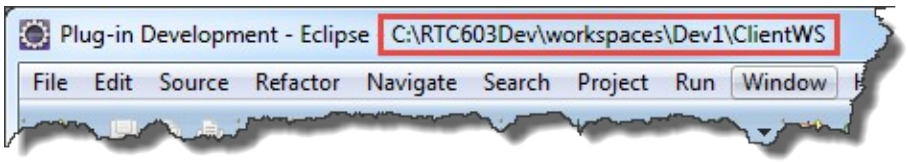
6.10 Create an Update Site for the RTC Eclipse client extension

Similar to the update site for the server extension you can create an update site for the client extension which can be used like the export destination to install the RTC Eclipse client extension.

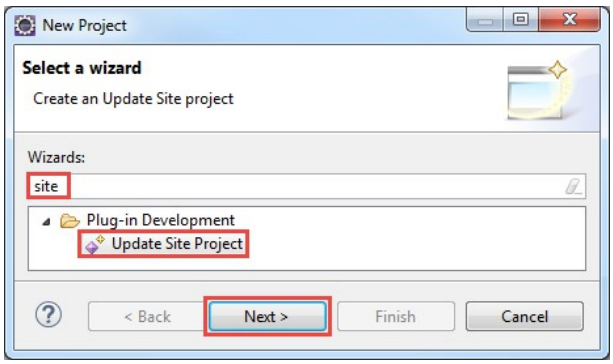
__170. Create the update site used to generate the install files.

__a. Return to the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS` from the last lab.

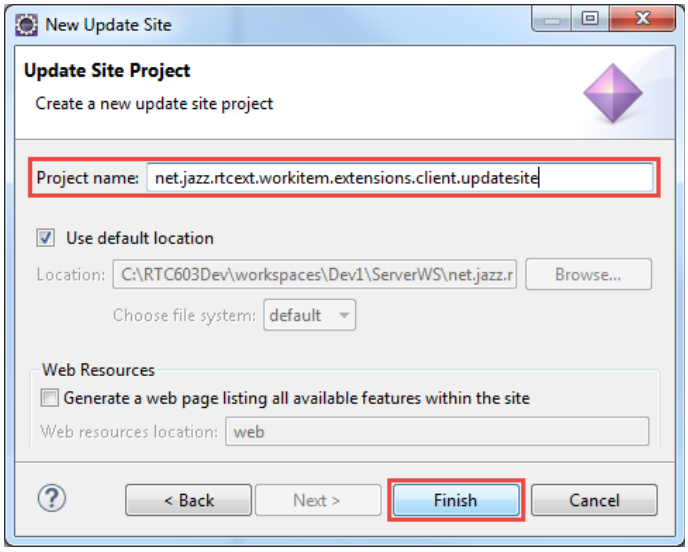
__i. Check the desired Eclipse workspace is used.



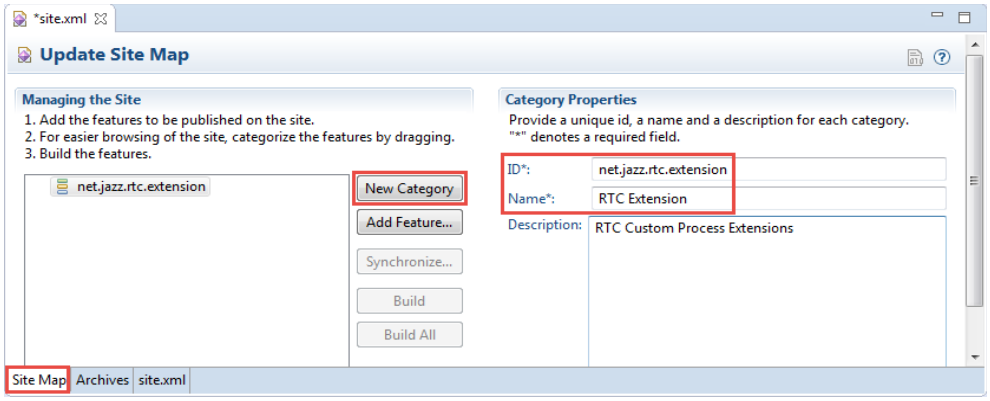
__b. From the menu bar, select **File > New > Project…** then in the **New Project** wizard, type `site` in the filter field, select **Update Site Project** from the list and then click **Next**



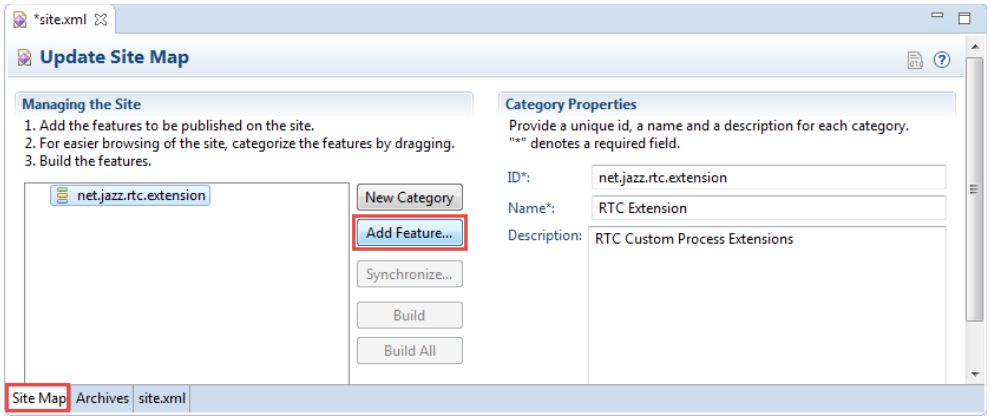
__c. On the second page of the wizard type `net.jazz.rtcext.workitem.extensions.client.updatesite` into the **Project name** field. Click **Finish**.



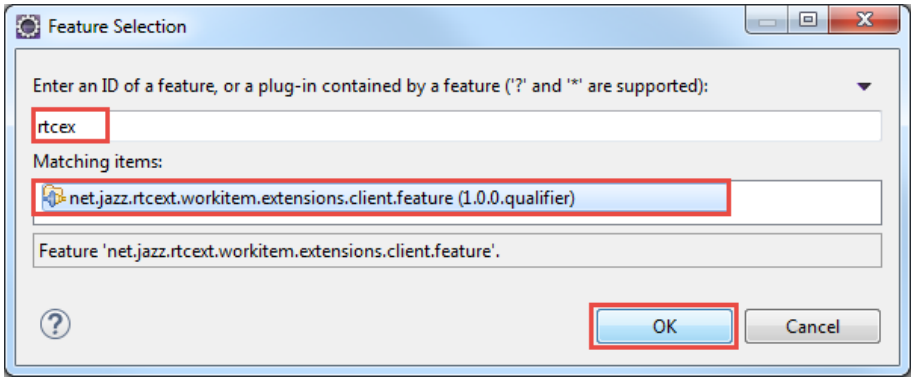
__d. Your new update site project appears in the **Package Explorer** view and an editor opens on the `site.xml` file. In the editor, remain on **Site Map** tab and click **New Category**. As ID enter `net.jazz.rtc.extension`. Provide `RTC Extension` as name



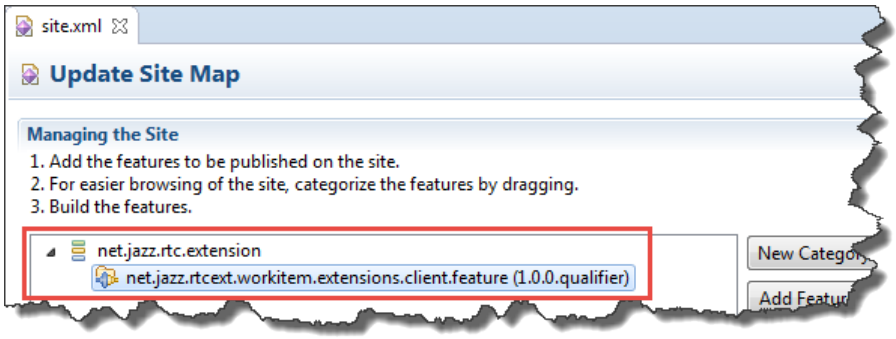
__e. In the editor select the new category and click **Add Feature**.



__i. In the **Feature Selection** dialog, type `rtcext` into the filter, select the feature you created in the last section and then click **OK**.

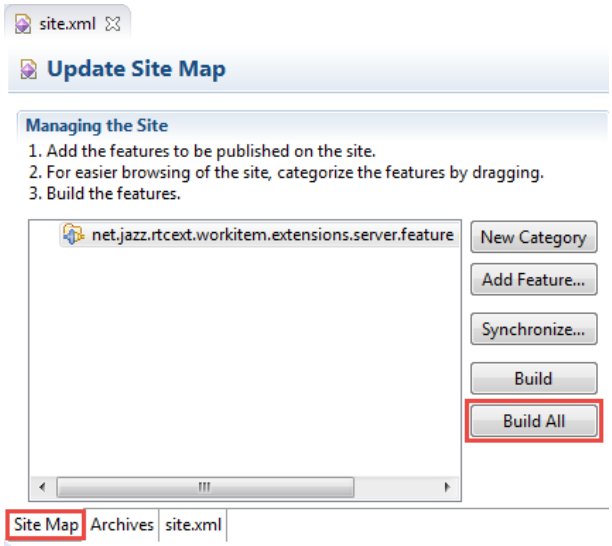


__ii. Check the feature is in the new category.



__iii. Back on the site.xml editor type **Ctrl+S** to save the file.

__iv. Later you can build the update site as explained for the server extension. The whole project with all generated files can be used similar to the install using the export. The Update site avoids the repeated manual process of exporting and makes the process precisely reproducible.



6.11 Complete RTC Eclipse Client extension development

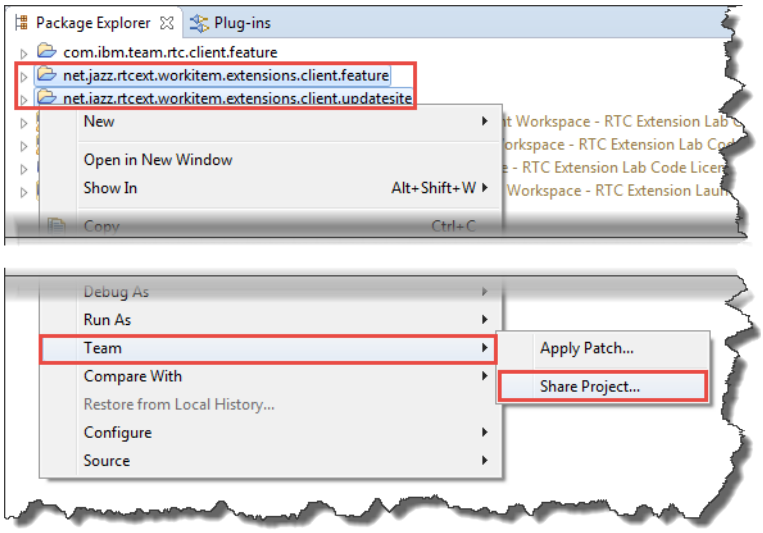
__171. Share the new projects to your client development repository workspace and deliver your work to the stream. Some complexity is added by suppressing specific components to focus on only client and only server development. The repository workspaces show only the components needed and the stream has all components. This requires to specify which components are available and which should flow where.

__a. Return to the Eclipse client with the workspace used for RTC Client SDK API development `C:\RTC603Dev\workspaces\Dev1\ClientWS` from the last lab.

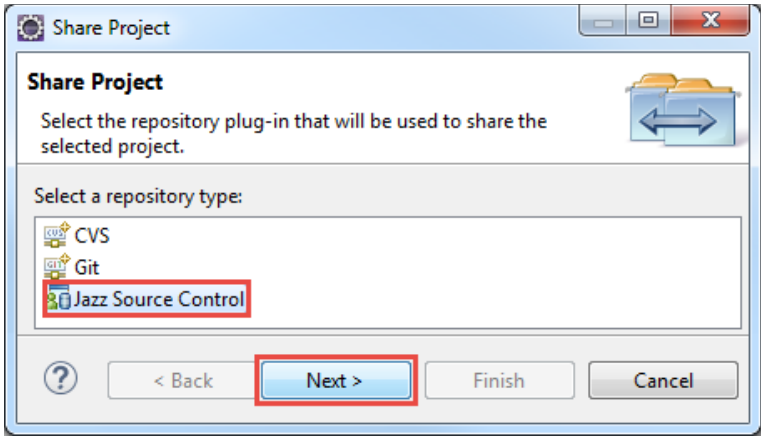
__i. Check the desired Eclipse workspace is used.

__b. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view in the **Work Items** perspective and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.

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

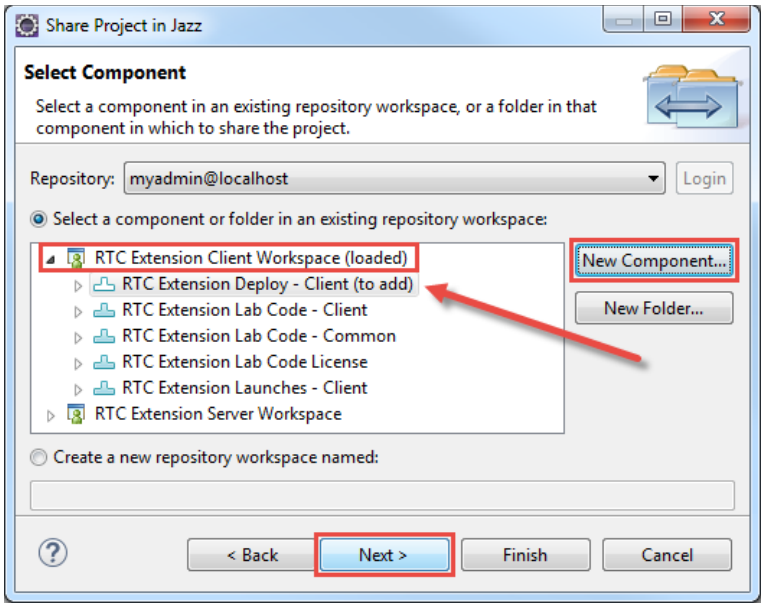


__d. In the **Share Project** wizard, select **Jazz Source Control** then click **Next**.

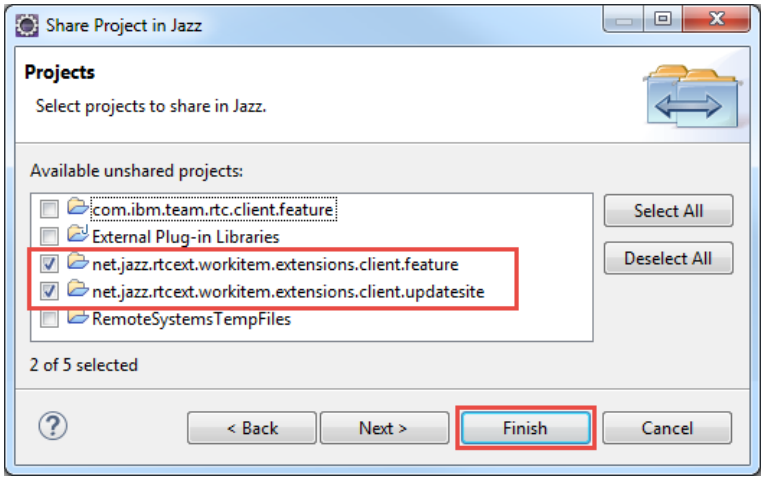


__e. On the second page of the wizard, select the **RTC Extension Client Workspace** (as highlighted with a red box) and click **New Component**.

In the **New Component** dialog, enter `RTC Extension Deploy - Client` 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**.

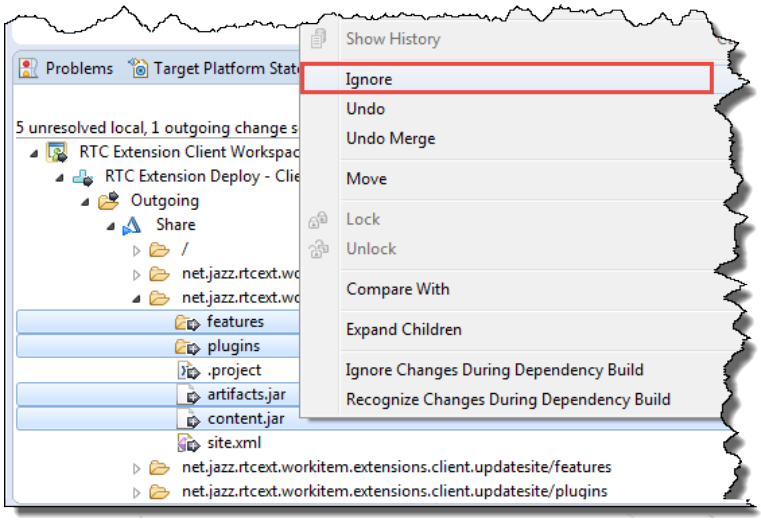


__f. On the third page of the wizard, confirm that the feature and update site projects are selected and then click **Finish**.

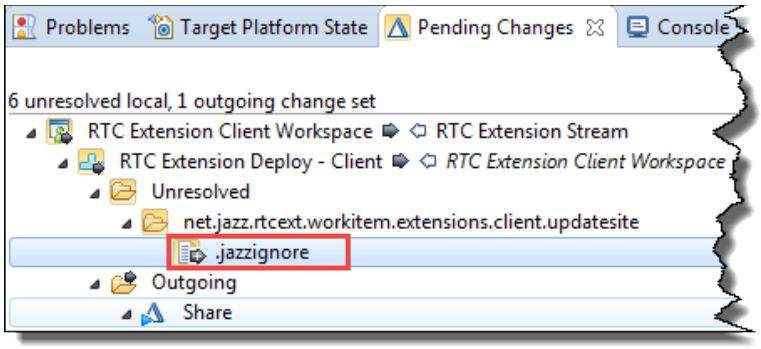


__g. The **Pending Changes** view will show your outgoing component addition with its newly shared projects. You will deliver them later.

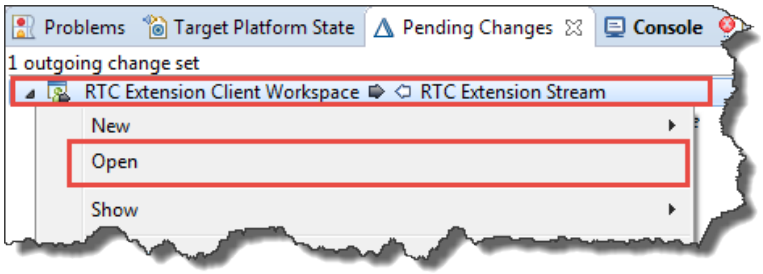
__h. If you created the client feature update site, unfold the sharing for the node `net.jazz.rtcext.workitem.extensions.client.updatesite`. If you built the update site, there are several files and folders that should not be under version control. The next steps make sure it never happens. Select the folders `features` and `plugins`. In addition select the files `artifacts.jar` and `content.jar`. Right click the selection and then select **Ignore** from the menu. 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.



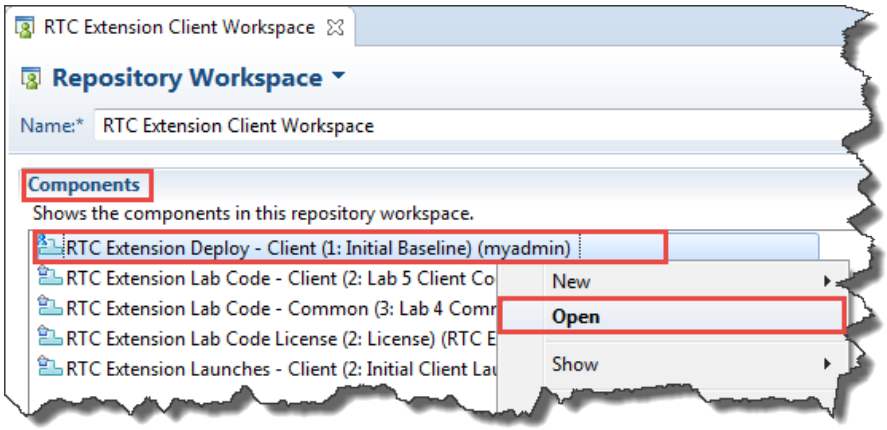
__i. The **Pending Changes** view will now show a new **.jazzignore** file as **Unresolved** for the component. Go ahead and check it in now by right clicking the file and then selecting **Check-in > Share** from the menu.



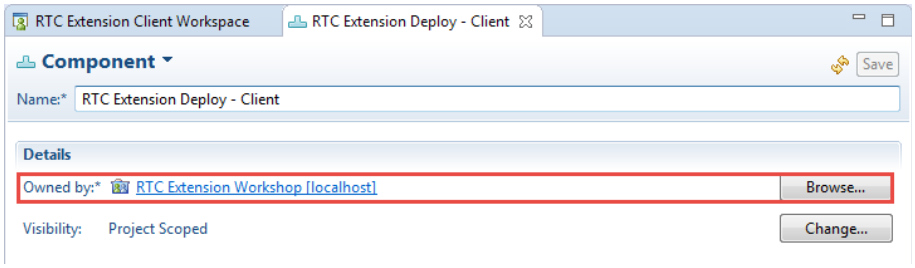
__j. The new component is not yet in the stream. To add the new component to the stream, right click on the repository workspace in the pending changes view and from the menu, select **Open**.



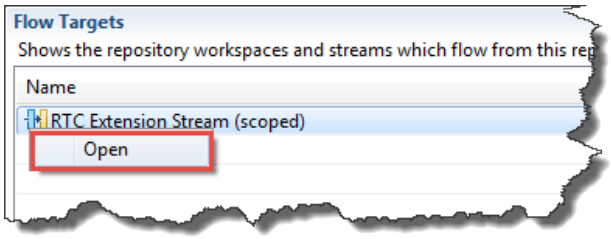
__k. The repository workspace editor opens. Scroll down to the **Components** section. The component is currently private to make it public right click the component **RTC Extension Deploy – Client** and then select the menu **Open**.



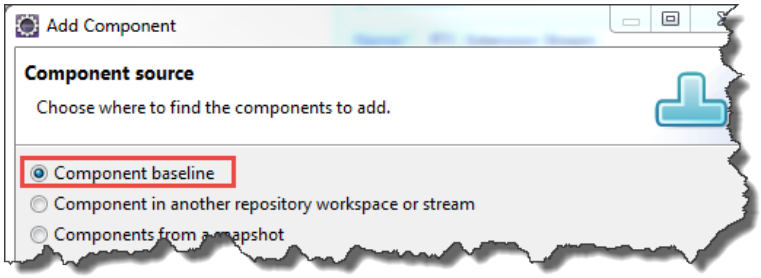
__l. In the component editor click **Browse...** and change **Owned by** to the project area **RTC Extension Workspace** and then click **Save**. Close the Component Editor.



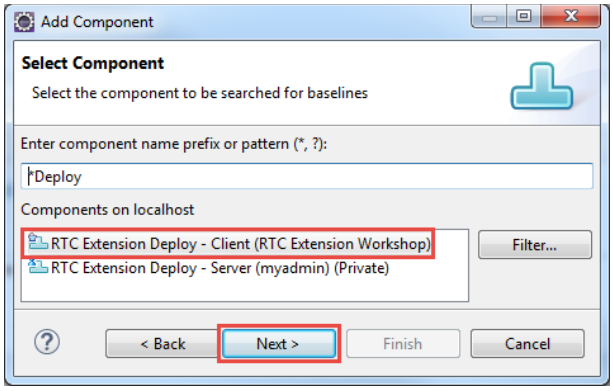
__m. In the Repository Workspace editor, scroll down to the **Flow Targets** section. In the flow target section right click the RTC Extension Stream and in the menu select **Open**.



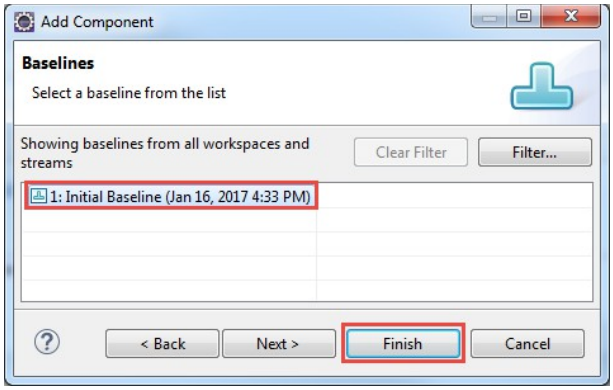
__n. In the stream editor go to the components section and click **Add...**. In the Add Component wizard select **Component Baseline** and click **Next**.



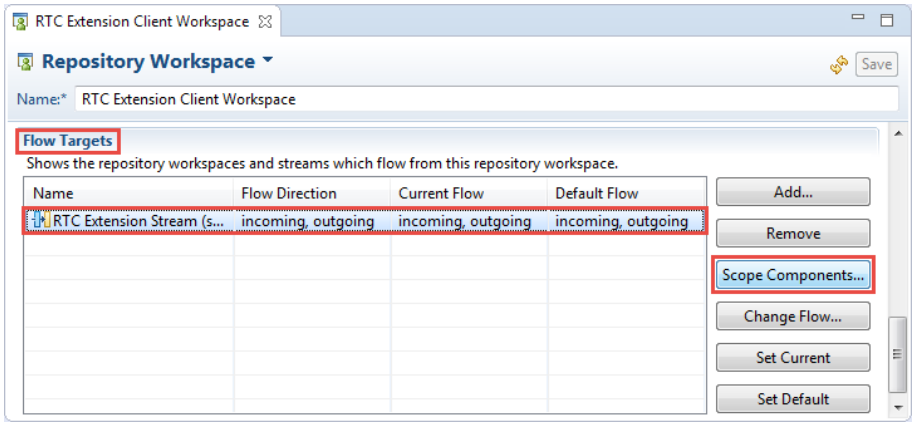
__o. In the filter type *Deploy. Select the component `RTC Extension Deploy - Client`. Click next.



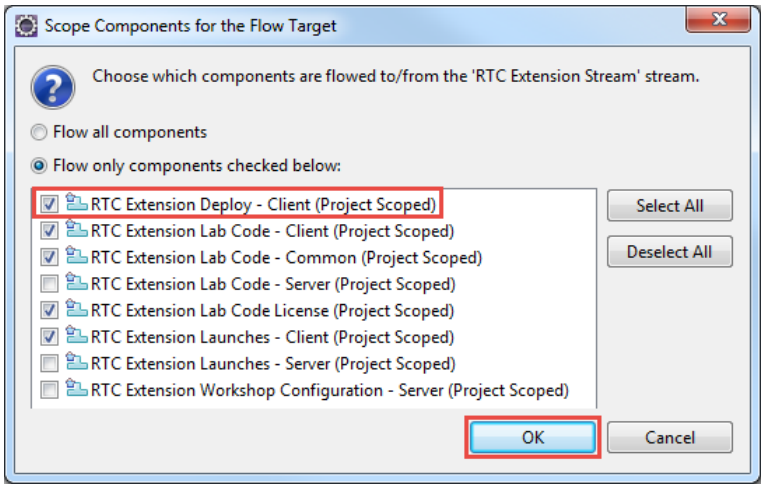
__p. Select the initial baseline of the component and click **Finish** to add it to the stream. **Save** the Stream. Close the stream editor.



__q. In the Repository Workspace editor in the **Flow Targets** section. Select the RTC Extension Stream and then click **Scope Components...**.

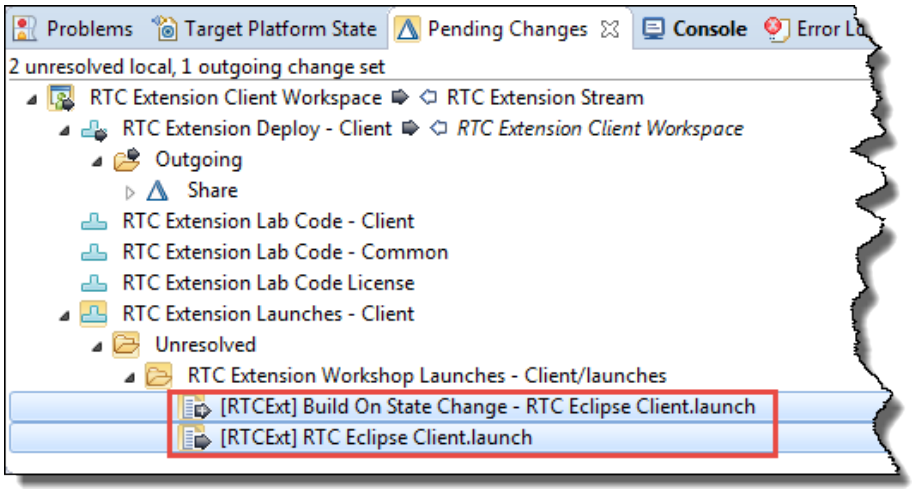


__r. Check the check box for the component RTC Extension Deploy – Client and make sure the following components are selected. Click **OK**.

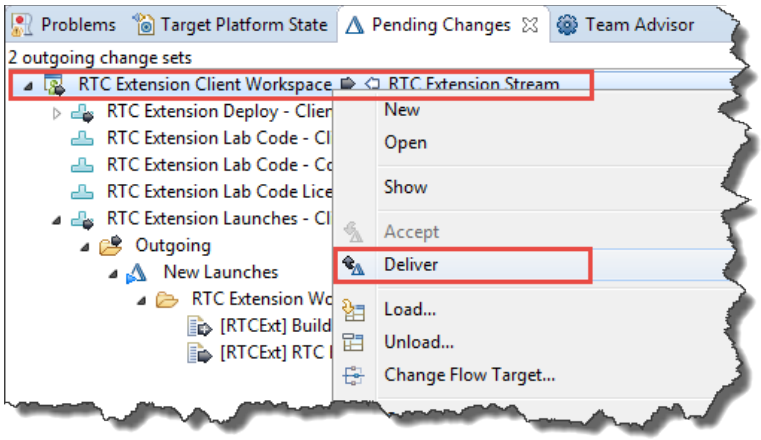


__s. Click **Save** to save the Repository Workspace.

__t. Check the other components for unresolved changes. You created a launch for the RTC Eclipse client which should be shared. Check the unresolved changes for the RTC Extension Workshop Launches – Client component into a new change set and comment it with **New Launches**.



__u. Right click the repository workspace and from the menu click **Deliver** to deliver all the changes to all components.



Note that the process template that was used for the project area on the development server defines some advisors. They prevent from delivering code with unused imports and compiler errors. In addition delivery requires a work item associated to each change set. The item needs to have an owner and must be planned for the current iteration.

You can remove the advisers. You can clean up the code, check-in the changes, assign a work item with owner and planned for the current iteration. You can try to retry deliver with override, if possible.

You have now checked in all changes made on your client development repository workspace and delivered them to the stream.

6.12 Complete RTC Server extension development

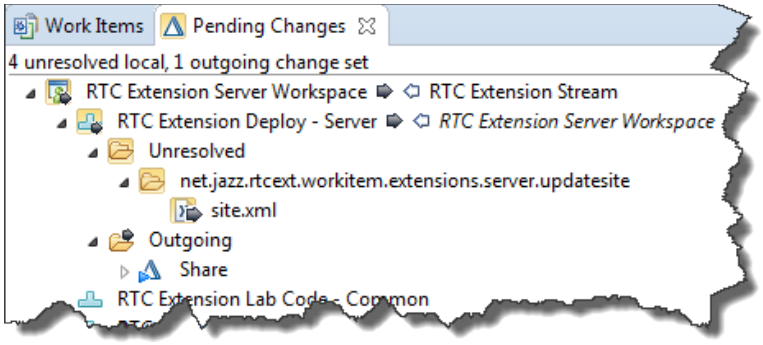
__172. Share the changes to your Server development repository workspace and deliver your work to the stream.

__a. Return to the Eclipse client with the workspace used for RTC Server SDK API development `C:\RTC603Dev\workspaces\Dev1\ServerWS`.

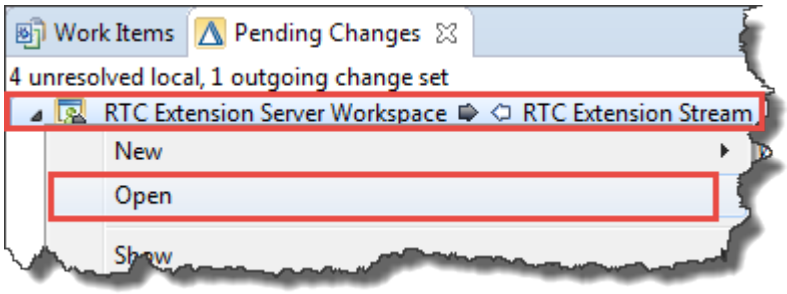
__i. Check the desired Eclipse workspace is in use.

__b. Make sure the client is connected to the **RTC Extension Workshop** project area. Go to the **Team Artifacts** view in the **Work Items** perspective and check the status of your repository connection. If disconnected **right click** on the project area and click **Log In**. Use `myadmin` as user ID and password.

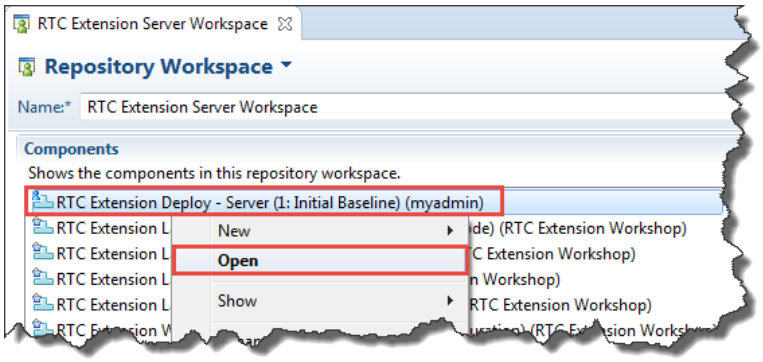
__c. The **Pending Changes** view will show your unresolved changes and the outgoing shared projects. You will deliver them later in this section.



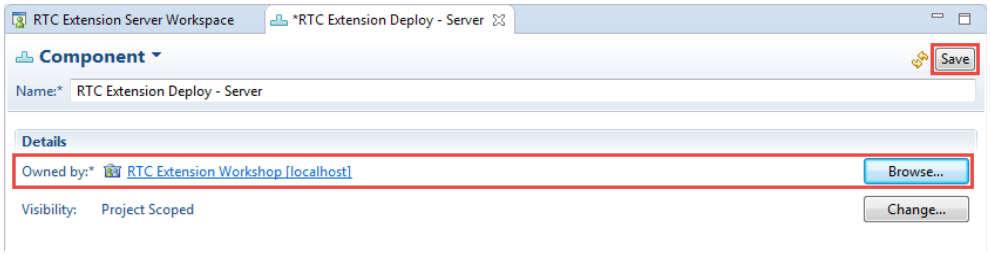
__d. The new component you created before is not yet in the stream. To add the new component to the stream, right click on the repository workspace in the pending changes view and from the menu, select **Open**.



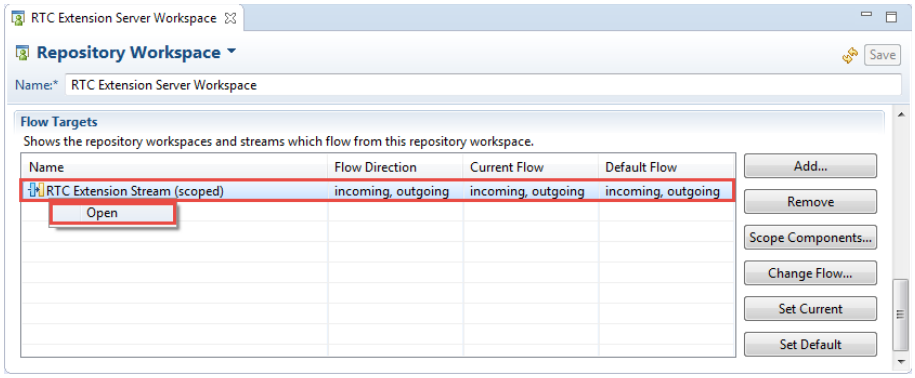
__e. The repository workspace editor opens. Scroll down to the **Components** section. Right click the component **RTC Extension Deploy – Server** and then select the menu **Open**.



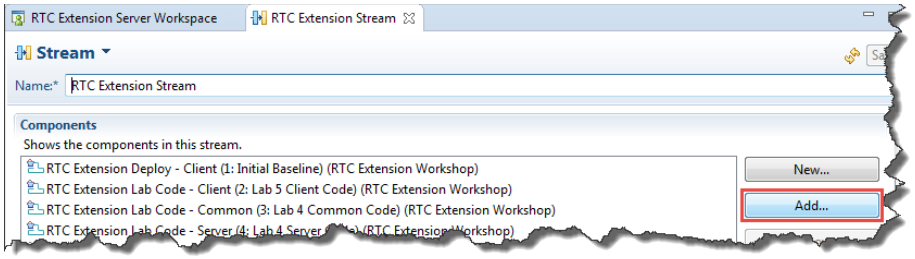
__f. In the component editor click **Browse...** and change **Owned by** to the project area **RTC Extension Workspace** and then click **Save** and close the component editor.



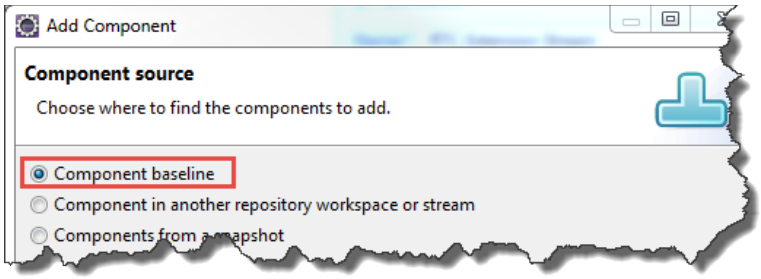
__g. Open the RTC Extension Stream.



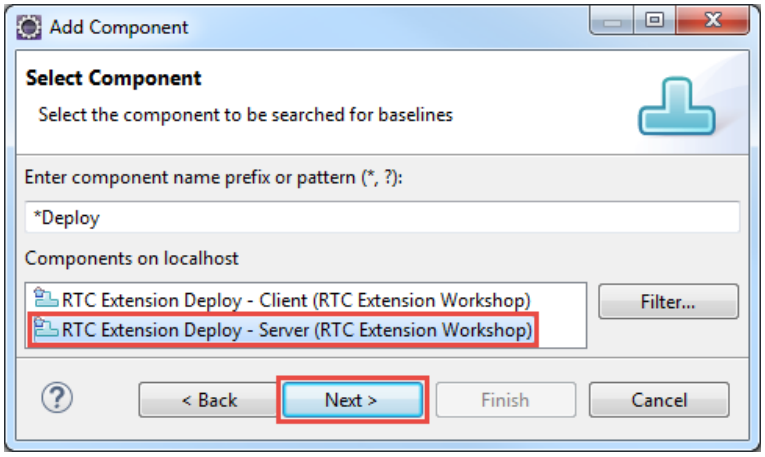
__h. In the Stream Editor go to the components section of the stream. Click **Add...**



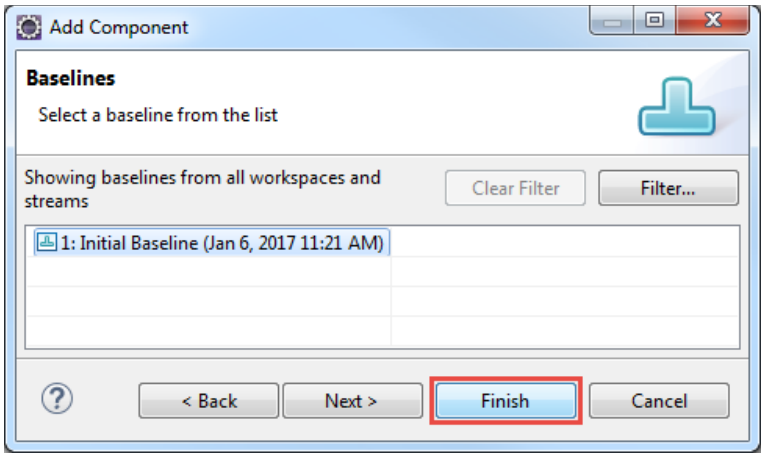
__i. In the Add Component wizard select **Component Baseline** and click **Next**.



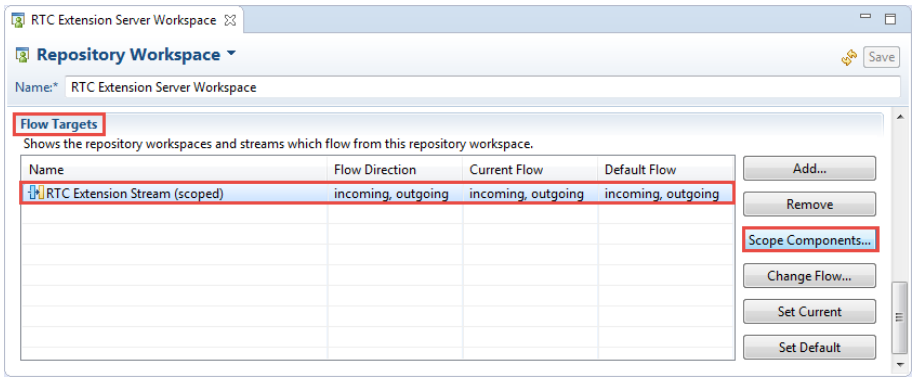
__j. In the filter type *Deploy. Select the component `RTC Extension Deploy - Server`. Click **Next**.



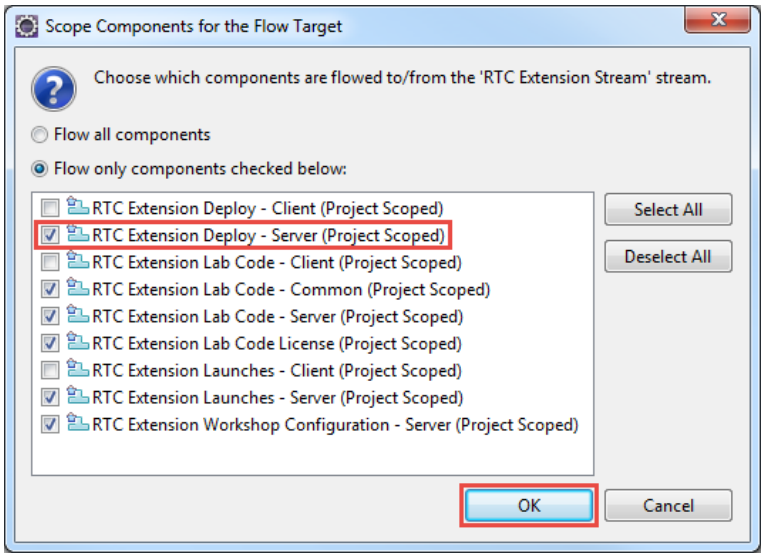
__k. Select the initial baseline of the component and click **Finish** to add it to the stream. **Save** the stream and close the stream editor.



__l. In the Repository Workspace Editor scroll down to the **Flow Targets** section. Select the RTC Extension Stream and then click **Scope Components...**.

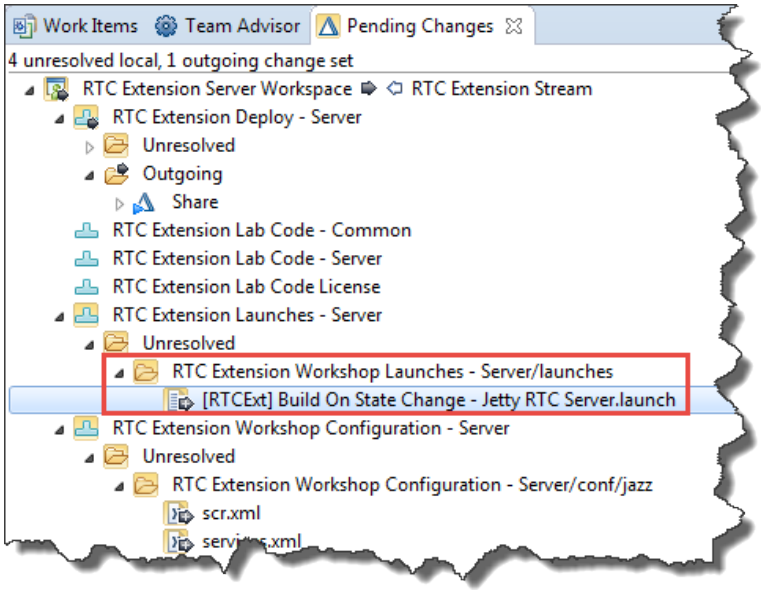


__m. Select the component `RTC Extension Deploy - Server`. Click **OK**.

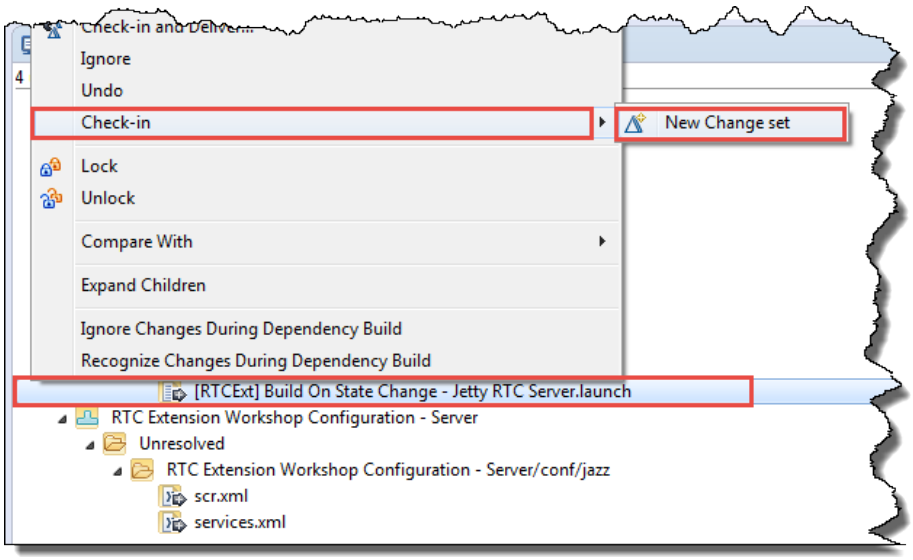


__n. Save the repository workspace and close the repository workspace editor.

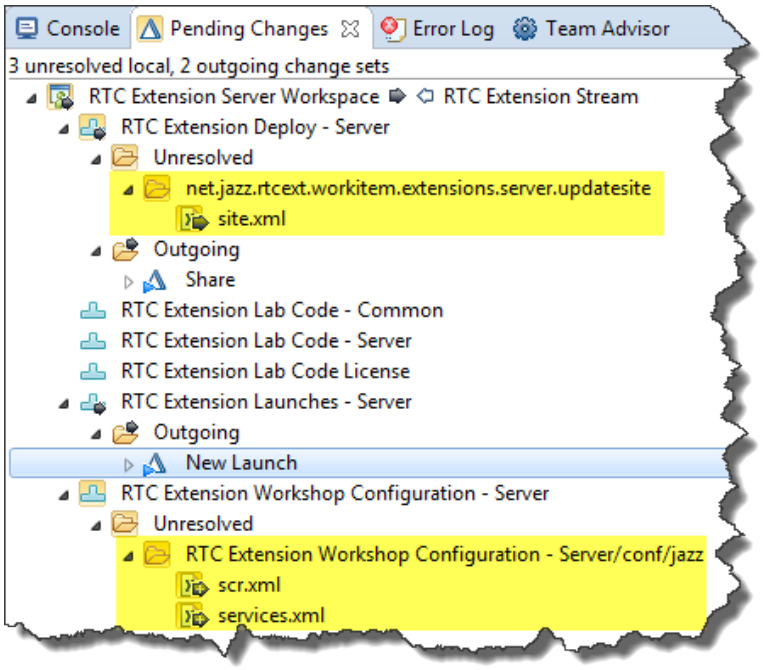
__o. Check the other components for unresolved changes. You created a launch for the Jetty RTC Server with the Build On State Change server extension that should be shared.



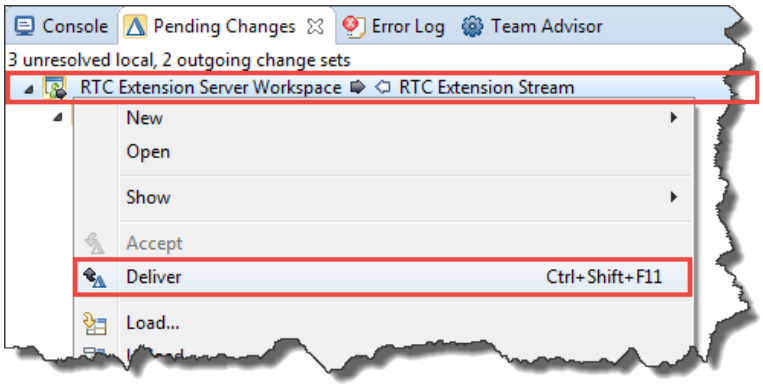
__p. Check the unresolved change for the RTC Extension Workshop Launches – Server component into a new change set and comment it with **New Launch**.



__q. The other changes should not be checked in. The site.xml file will be regenerated from the version that already is checked in. The scr.xml and the services.xml are specific to the current version of RTC and should be manually copied from the specific server install during setup.



__r. Right click on the workspace in the pending changes view and click the menu item Deliver.

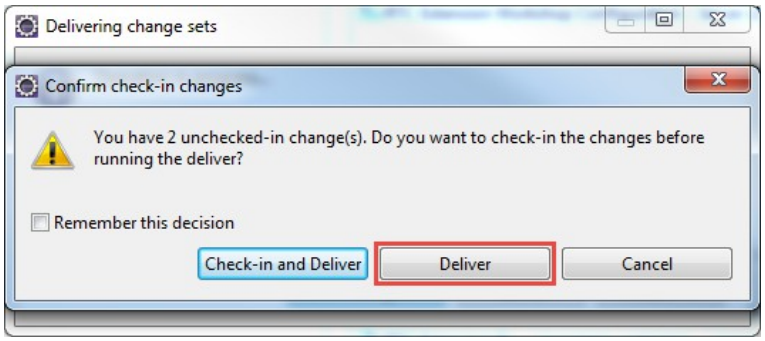


Note that the process template that was used for the project area on the development server defines some advisors. They prevent from delivering code with unused imports and compiler errors. In addition delivery requires a work item associated to each change set. The item needs to have an owner and must be planned for the current iteration.

You can remove the advisers. You can clean up the code, check-in the changes, assign a work item with owner and planned for the current iteration. You can try to retry deliver with override, if possible.

You have now checked in all changes made on your client development repository workspace and delivered them to the stream.

__s. To avoid checking in the changes mentioned above click Deliver on the warning message that comes up.



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 (https://jazz.net/wiki/bin/view/Main/RtcSdk30). 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.

There are many examples describing how to use the Java Client and Server API in this blog. A good overview how to get started that also references this workshop is the post Getting Started with the RTC Java API’s. You can search the blog for key words. The page Interesting Links points to various sources with examples to solve specific needs.

Appendix A Notices

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

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

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

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

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

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

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

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

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

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

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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

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

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

COPYRIGHT LICENSE:

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

Appendix B Trademarks and copyrights

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

IBM	IBM Logo	Rational	Jazz

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

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

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

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

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

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

NOTES

NOTES

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

Please Recycle

Rational® software