Getting Started with *featherlite* Plug-ins

Author: Martin Smock, Robert von Burg
Type: *featherlite* Documentation
Date: 2011-02-18
Version: 0.10.0

apixxo ag

© Copyright 2010-2011

Abstract

This section explains the steps necessary to setup the development environment to run and extend *featherlite* Plug-ins with the help of the Eclipse integrated development environment (Eclipse IDE).

Contents

1 Installation : Required Software Packages **1**

1.1 Using the *featherlite* Eclipse SDK 1

1.2 Installing the Core Eclipse IDE 2

1.3 Installing the Subclipse SVN Plug-in 3

1.4 Installing the XML Plug-in 3

1.5 Installing the *featherlite* Plug-ins 4

1.6 Updating, and adding further *featherlite* Plug-ins 9

2 *featherlite* Core Plug-ins and Applications **10**

2.1 Core Plug-ins 10

2.1.1 RSP 10

2.1.2 RSPCore 10

2.1.3 PlanningClientCore 11

2.1.4 JEPlugin 11

2.1.5 ExecutionClientCore 11

2.2 Reference Applications 12

2.2.1 PlanningServer 12

2.2.2 ExecutionServer 12

2.2.3 PlanningClientRun 13

2.2.4 ExecutionClientRun 13

2.2.5 Model Editor 13

3 Creating Custom Projects **15**

3.1 Server 15

3.1.1 Creating the server plug-in 15

3.1.2 Add the *featherlite* plug-in dependency 18

3.1.3 Plug-in configuration 19

3.1.4 Application Extension Point 20

3.1.5 Configure the *featherlite* runtime environment 24

3.1.6 Using the ServerTemplate plug-in as a server 25

3.2 Client 27

3.2.1 Create the rich client plug-in using a template 27

3.2.2 Add the *featherlite* plug-ins as dependencies 28

3.2.3 Add the *featherlite* configuration files 29
3.2.4 Add the code to configure the application 31
3.2.5 Configure the 'WorkbenchAdvisor' 32
3.2.6 Configure the 'WorkbenchWindowAdvisor' 32
3.2.7 Using the ClientTemplate plug-in as a client 32

4 Running an Eclipse Plug-in 34

4.1 Running the Server Plug-in 34
4.1.1 Starting the server through plugin.xml 34
4.1.2 Setting up a Run Configuration Manually 34
4.2 Running the Client Plug-in 38

1 Installation : Required Software Packages

To install the Eclipse IDE for *featherlite* Plug-in Development, we recommend installing the following software packages (a detail explanation is given in the following):

- Eclipse Classic IDE - although the minimal configuration is the Eclipse for RCP Plug-in Developers IDE, the recommended package to download is the full featured classic version. The package can be loaded free of charge from the eclipse download site. Version of writing is 3.5.x
- Subclipse Plug-in - the revision control system software the developers at apixxo ag use. The package can be installed free of charge by using the Eclipse Software Update functionality from within the running eclipse application. Version of writing is 1.6.x
- Eclipse XML Editors and Tools - although not necessarily required, the use of the 'Eclipse XML editors and tools' plug-in is highly recommended to simplify modification of the XML configuration files. The package can be installed free of charge by using the Eclipse Software Update functionality from within the running eclipse application.
- *featherlite* Core Plug-ins - required for development. To simplify project specific development the *featherlite*-plug-ins are shipped as Eclipse Plug-Ins with open java source code.
- *featherlite* reference applications - useful for development. These applications are pre-compiled for different operating systems and can be directly started.

If you already have an existing eclipse installation which meets the above requirements, you can just proceed to install the *featherlite* plug-ins in your existing installation which is described in Section 1.5.

1.1 Using the *featherlite* Eclipse SDK

As an alternative, you can also download the *featherlite* development kit (*featherlite* and Eclipse Plug-In development package). This kit (shipped as a zip-archive file) includes the required eclipse platform and additional components, and eclipse is pre-configured with all steps contained in this section.

You can download the kit suited for your computer platform[1], uncompress it to your preferred location, and start it by running the *start_eclipse* script.

Note: In certain cases the Eclipse SDK does not properly recognize the *featherlite* plugins. This may happen if new plug-ins are downloaded and placed in the original plug-in folder.

[1] http://featherlite-framework.com/download#Development_Kits

This problem can be resolved by going to Window → Preferences → Plug-in Development → Target Platform. In this dialog select the “Running Platform (Active)” entry. Now press the “Reload. . . ” button. After the plug-ins are re-scanned, the Eclipse Workspace should again find the *featherlite* plug-ins. Figure 1 shows this dialog. After clicking ok on the following two dialogs, the problems should be resolved.

Now, you can proceed to Section 2, which describes the different *featherlite* plug-ins.

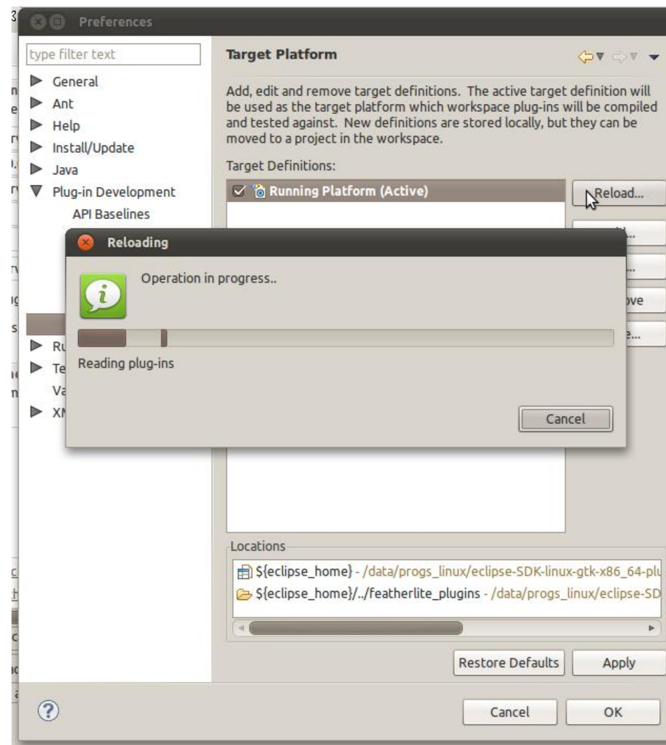

Figure 1: Reloading the platform so that new plug-in versions are found.

1.2 Installing the Core Eclipse IDE

To install the Eclipse Classic IDE proceed as follows:

- download the Eclipse Classic IDE from the eclipse download web page `www.eclipse.org/downloads`.
- unzip the Eclipse Classic IDE package to your preferred location on the hard disk.
- start the eclipse application from the installation folder.
- when the Workspace Launcher dialog asks to select a workspace, press the Browse. . . button and create a new folder for your workspace.

1.3 Installing the Subclipse SVN Plug-in

To install the Subclipse Plug-in proceed as follows:

- from the eclipse SDK 'Help' menu select 'Install New Software',
- press 'Add' to open a dialog to add the sublipse site,
- in the 'Add Site' dialog type 'subclipse' into the name field and `http://subclipse.tigris.org/update_1.6.x` as the URL.
- from the available packages select the 'Core SVNkit Library' and 'Subclipse' (see Figure 2) and press 'Next' to continue,
- from the 'Install Details': select 'Subclipse' and click 'Next' to proceed, or just proceed next if you cannot make a selection,
- in the 'Install' wizard review the license agreement, mark 'I accept the license agreement' and press 'Finish' to install.
- Review any security warning about the software plug-ins you're installing. To install successfully, you have to confirm all software and any certificate that may appear. Note that depending on your previous eclipse installations, these warnings may not appear.
- Restart Eclipse if you're asked to do so.
- Depending on your machine installation, you might be required to change the SVN interface the plug-in uses (if you get an error message about JavaHL (JNI), you have to). To this aim, under preferences in the “Window” menu, select the “Team → SVN” entry, and select “SVNKit (Pure Java)” as an SVN interface (instead of JavaHL (JNI)).

1.4 Installing the XML Plug-in

To install the 'Eclipse XML Editors and Tools' plug-in proceed as follows:

- from the eclipse SDK 'Help' menu select 'Install New Software',
- in the 'Install' dialog select 'All Available Sites' from the selection list in the upper part of the dialog (the “Work with:” selection),
- wait for the list of available plug-ins to be updated by eclipse,
- scroll to 'Web, XML and Java EE Development' and open the tree node to see all plug-ins available for that topic,
- select 'Eclipse XML Editors and Tools' (See Figure 3) and proceed with 'Next',

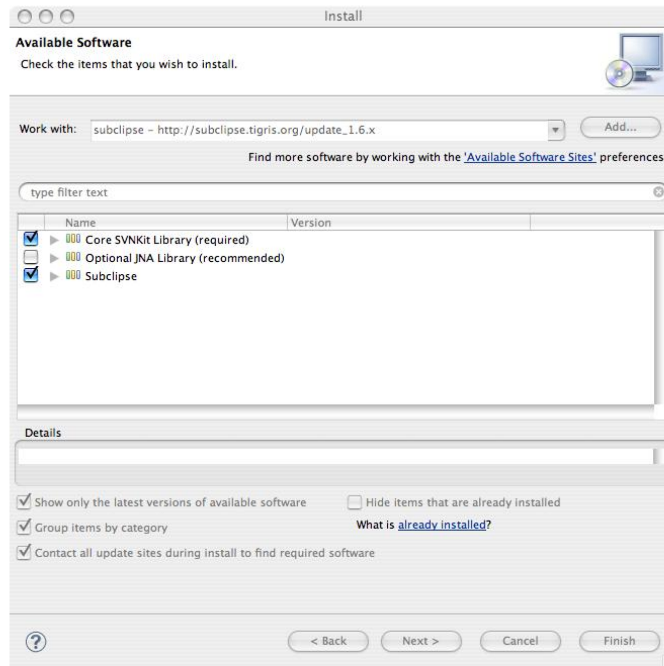

Figure 2: The dialog settings to install the 'Subclipse' plug-in.

- from the 'Install Details' select 'Eclipse XML Editors and Tools' and click 'Next' to proceed,
- in the 'Install' wizard review the license agreement, mark 'I accept the license agreement' and press 'Finish' to install.
- If you're asked to confirm the Eclipse.org Foundation certificate, you shall do so, and restart Eclipse when installation is complete.

1.5 Installing the *featherlite* Plug-ins

To install the *featherlite* Core Plug-ins proceed as follows:

1. Download the *featherlite* plug-ins from the *featherlite* download[2]. The minimum required plug-in is the "featherlite framework" plug-in, which has the plug-in id "RSP". The other plug-ins are optional, depending on your needs. Downloading the "featherlite core sdk" zip will give you all plug-ins in a single archive.
2. Designate and create a dedicated *featherlite* plug-in directory on your computer, where all *featherlite* plug-ins will be kept,

[2] `http://featherlite-framework.com/download#Plugins`

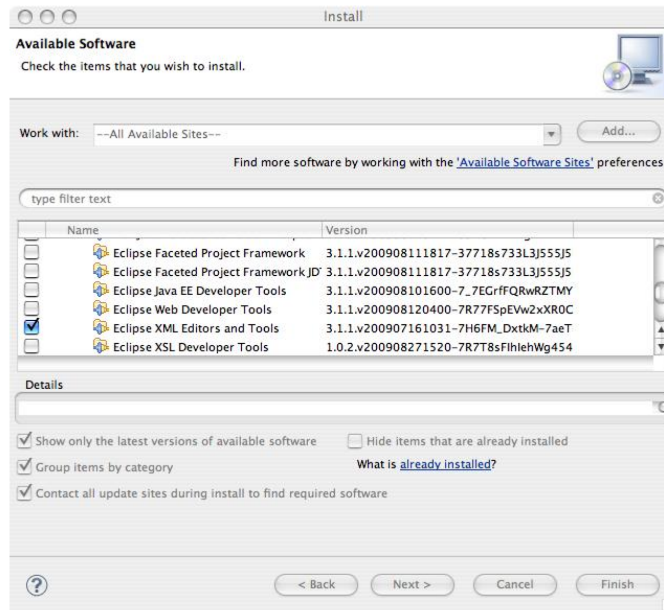

Figure 3: Install dialog settings to install the 'Eclipse XML Editors and Tools' Plug-in.

3. unpack the zip-files containing the *featherlite* plug-ins to the dedicated *featherlite* plug-in directory[3],
4. in Eclipse, open the preferences under the Window menu. Under Plug-in Development, select Target Platform, as shown in Figure 4,
5. edit the Running Platform (Active). Under Locations click Add to add a new directory as a location. Choose the location where the *featherlite* plug-ins have been unpacked to, as shown in Figures 5 to 7,
6. now the Locations tab should show an additional entry with the *featherlite* plug-ins, as shown in Figure 8,
7. after clicking on Finish, and closing the dialog with OK, Eclipse will update its platform as is shown in Figure 9. Once the platform has been re-initialized the *featherlite* plug-ins can be used in new Eclipse projects.

[3] The directory structure inside the zip files must be retained. Plug-ins which have additional JAR dependencies are delivered in a directory structure, plug-ins with no additional JAR dependencies are delivered as JAR files. Also check out the contents of the archive; depending on the plug-in, the archive might contain example configuration files and models which can help getting started with the plug-in.

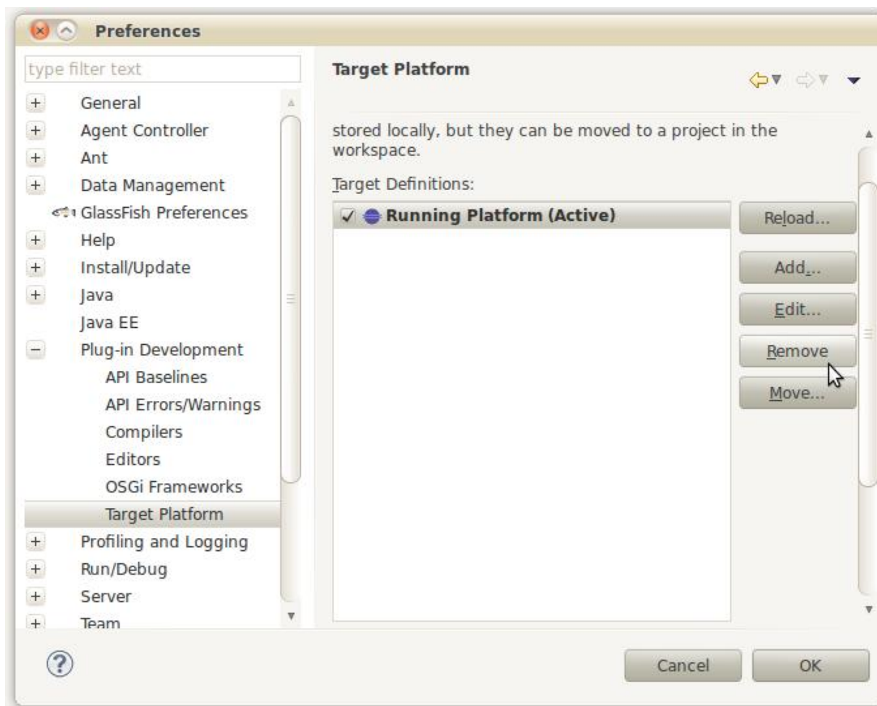

Figure 4: Eclipse Preferences showing the target platform.

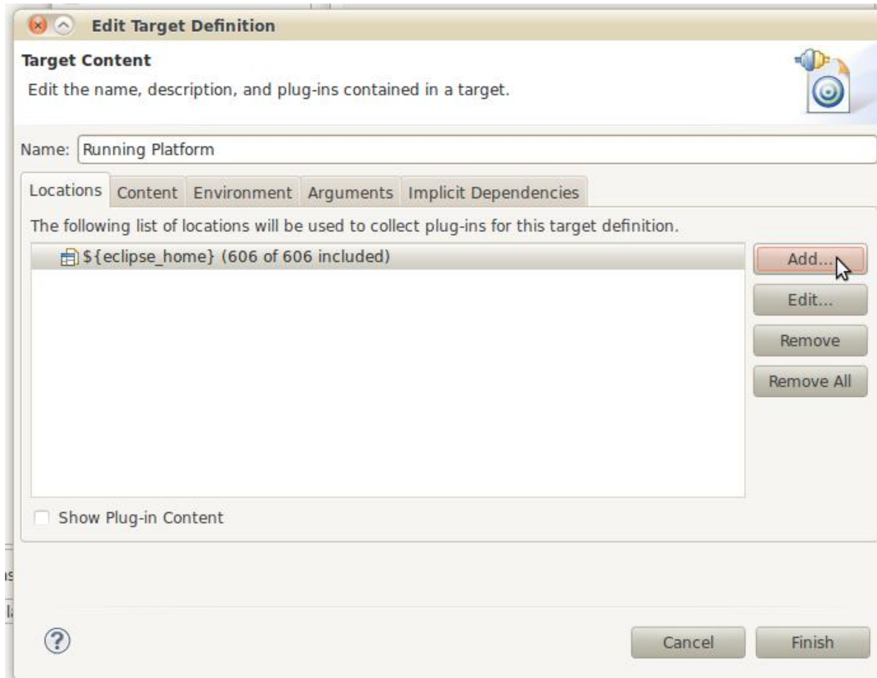

Figure 5: Editing of the currently active and running target platform.

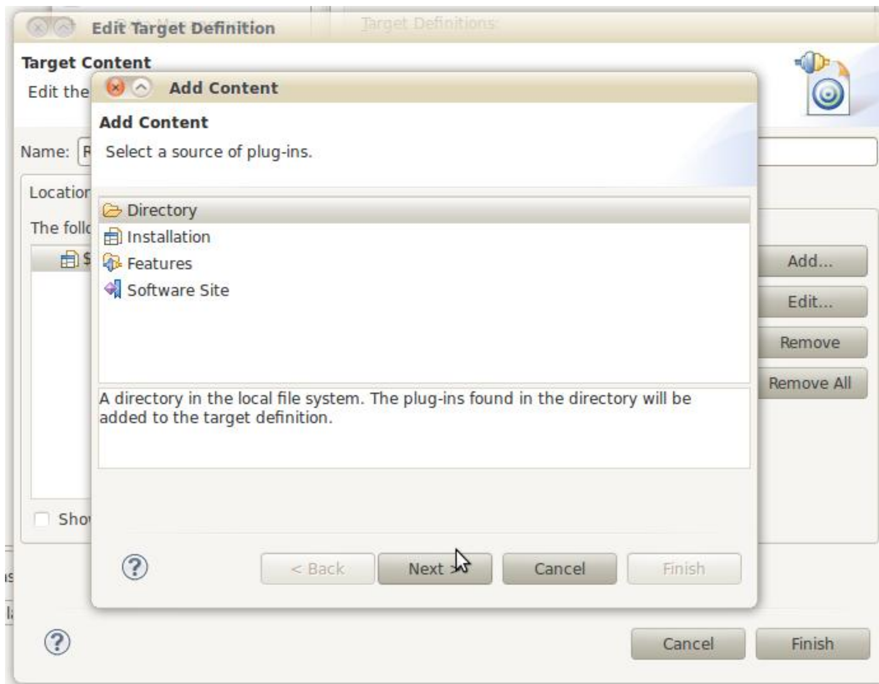

Figure 6: Adding an additional plug-in location to the platform.

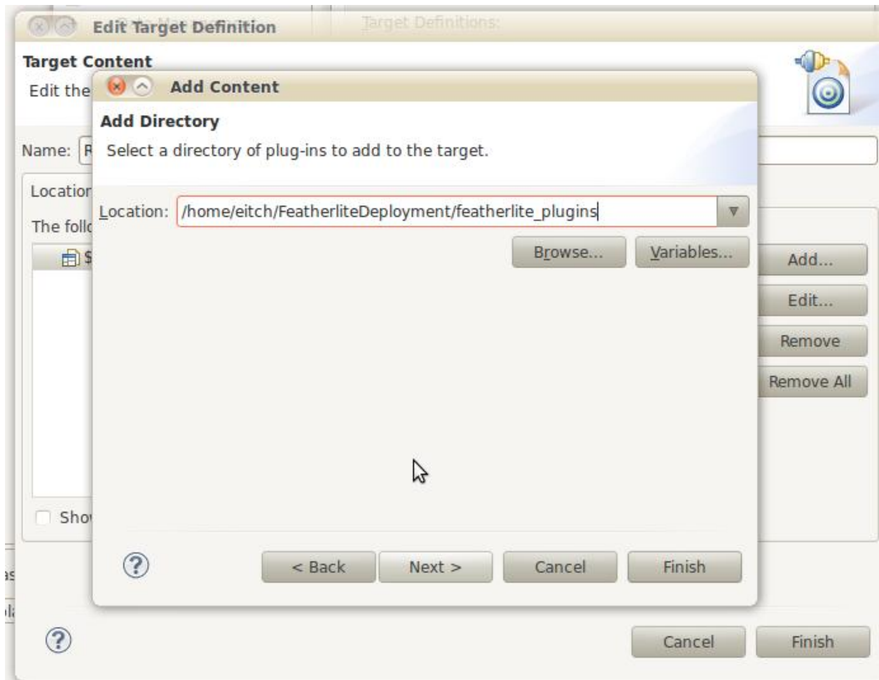

Figure 7: Defining the directory location for additional plug-ins.

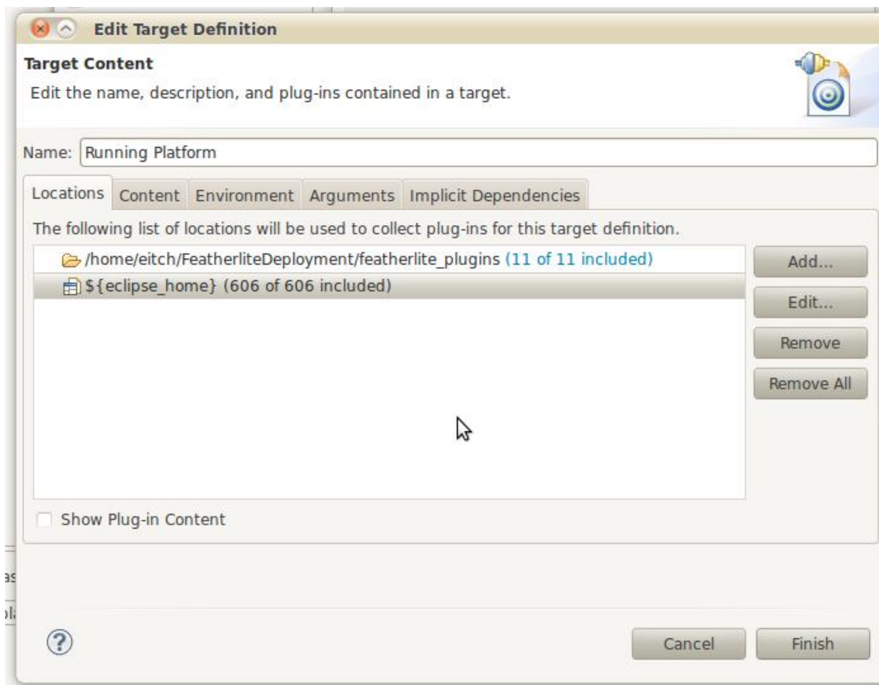

Figure 8: The new locations settings for the running target platform.

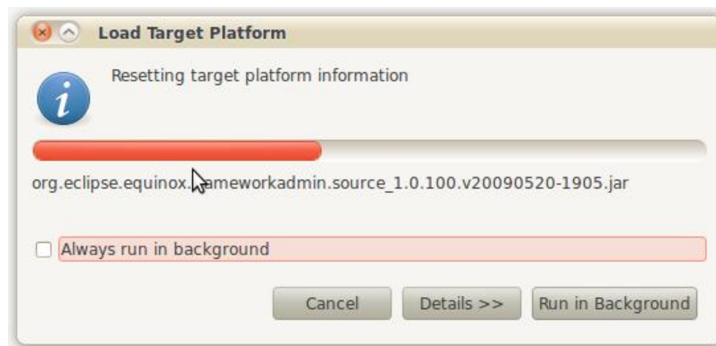

Figure 9: Eclipse is updating the target platform, adding the *featherlite* plug-ins to the runtime environment.

1.6 Updating, and adding further *featherlite* Plug-ins

To update, or add further *featherlite* plug-ins the new plug-ins must be dropped in the same directory which was defined in section 1.5. Now all that is needed, is to let Eclipse re-scan the current platform.

This is done by going to Window → Preferences → Plug-in Development → Target Platform. In this dialog select the "Running Platform (Active)" entry. Now press the "Reload..." button. After the plug-ins are re-scanned, the Eclipse Workspace should again find the *featherlite* plug-ins. After closing all dialogs by clicking on "Ok", the new versions of the *featherlite* plug-ins should be available. In rare cases a restart of Eclipse should resolve any oddities.

Figure 10 shows this dialog.

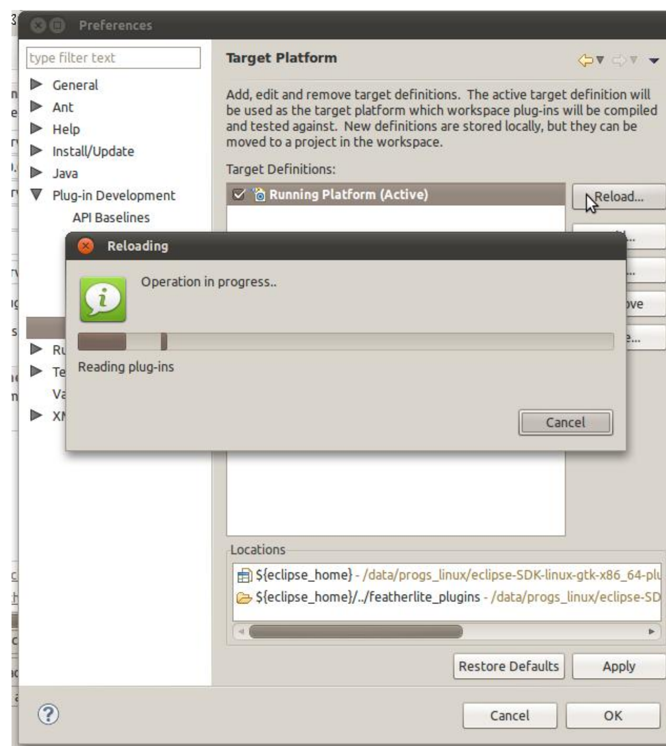

Figure 10: Reloading the platform so that new plug-in versions are found.

2 *featherlite* Core Plug-ins and Applications

In what follows, we explain the *featherlite* Core Plug-ins which are used to create *featherlite* projects and the reference implementations which are delivered as pre-compiled applications.

2.1 Core Plug-ins

featherlite core plug-ins are used to build *featherlite* applications.

2.1.1 RSP

The *featherlite* RSP plug-in provides all the core classes used to build the *featherlite* server plug-ins to be deployed either on a Web Application Container (JBoss, Apache Tomcat, and others) or in a pure Java virtual machine, if running in standalone mode. The *featherlite* plug-in is the core plug-in, that is extended by all server and client plug-ins. The main core classes include:

- the *featherlite* Data Model elements, including resources, orders, scripts, workflows and many other components,
- the *featherlite* Handler, including the Control Handler Scheduler, the Integration Handler, PLC Communication Handler, the Client Handler, and many others,
- the *featherlite* Core Policies with many algorithms ready to use,
- the core *featherlite* Services and Commands,
- and other common functions.

2.1.2 RSPCore

The *featherlite* RSPCore plug-in covers the code and configuration files common to all Rich Client Plug-ins described below. As such, it cannot be deployed itself, but has to be extended by the application specific plug-ins. The plug-in consists of the following parts, which can be used in all extending plug-ins:

- core classes for the table and list views,
- *featherlite* Gantt Chart View and related classes,
- *featherlite* Object Inspector classes,
- *featherlite* Error View,
- content provider implementing the remote observer pattern,

- the core actions and dialogs,
- default views for *featherlite* order and resource objects,
- and other common functions.

2.1.3 PlanningClientCore

The *featherlite* PlanningClientCore plug-in extends the *featherlite* RSPCore plug-in and provides ready to use views and functions to build rich client applications to use in a planning and scheduling context. It contains:

- specialized views for *featherlite* order and resource objects,
- a set of actions for common activities in the context of planning,
- and other commonly used functions.

2.1.4 JEPlugin

The *featherlite* JEPlugin is a plug-in which implements an object-oriented persistence using the Berkeley DB Java Edition[4].

The JEPlugin implements all DAOs to persist the *featherlite* objects including a RSP-ComponentLifecycle class which is to be configured as a *featherlite* component so that the JEPlugin is activated as the *featherlite* DAOHandler.

2.1.5 ExecutionClientCore

The *featherlite* ExecutionClientCore plug-in extends the *featherlite* RSPCore plug-in to be deployed as a rich client application. It provides ready to use views and functions to build rich client applications for logistics control. It contains:

- a specialized view for *featherlite* order objects,
- a set of actions for commonly used activities in the execution context,
- a set of views and actions to control workflow execution and monitor messages exchanged with the shop floor,
- and other commonly used functions.

[4] `http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html`

2.2 Reference Applications

featherlite reference applications are pre-compiled application which can be started. These applications showcase the possibilites of the *featherlite* platform, or can be used as a starting point for custom *featherlite* projects.

Further the reference clients can be used to access a running *featherlite* instance and modify runtime data.

2.2.1 PlanningServer

The PlanningServer is the *featherlite* reference implementation for a server application used in planning and scheduling context. It defines all the needed configuration to run a *featherlite* instance in a planning and scheduling context. The application should be understood as a starting point for the development and testing of customer specific plug-ins. For this purpose the PlanningServer plug-in contains

- the core configuration file config/RSPConfig.xml, defining the basic runtime properties of the application,
- the hibernate configuration file for the Object Relational Mapping, located at src/hibernate.cfg.xml,
- the application configuration files, available in the 'config' folder, defining the runtime components and other details of the application,
- a collection of data model configurations serialized as xml files, located in the models folder,
- and other common functions.

As the PlanningServer is delivered as an application, these files are located in the plugins/PlanningServer_<version> directory of the application.

2.2.2 ExecutionServer

The ExecutionServer is the *featherlite* reference implementation for a server application used in execution control context. It defines all the configuration needed to run a *featherlite* instance in a scheduling and control context. The application should be understood as a starting point for the development and testing of customer specific server plug-ins. For this purpose the ExecutionServer plug-in contains

- the core configuration file 'config/RSPConfig.xml', defining the basic runtime properties of the application,
- a hibernate configuration file for the Object Relational Mapping, located at src/hibernate.cfg.xml,

- the application configuration files, available in the 'config' folder, defining the run-time components and other details of the application,
- a collection of data model configurations serialized as XML files, located in the 'models' folder,
- and other common used functions

As the ExecutionServer is delivered as an application, these files are located in the plugins/ExecutionServer_<version> directory of the application.

2.2.3 PlanningClientRun

The PlanningClientRun is the *featherlite* reference implementation for a rich client application in the context of planning. It packages the PlanningClientCore plug-in as a standalone application. It contains:

- the application configuration file 'config/clientConfig.xml', defining the basic, run-time properties of the application
- the 'config/log4j.properties' configuration file, defining the logging properties of the application.

As the PlanningClient is delivered as an application, these files are located in the plugins/PlanningClientRun_<version> directory of the application.

2.2.4 ExecutionClientRun

The ExecutionClientRun is the *featherlite* reference implementation for a rich client application in execution context. It packages the ExecutionClientCore plug-in as a standalone application. It contains:

- the application configuration file 'config/clientConfig.xml', defining the basic run-time properties of the application,
- the 'config/log4j.properties' configuration file, defining the logging properties of the application.

As the ExecutionClient is delivered as an application, these files are located in the plugins/ExecutionClientRun_<version> directory of the application.

2.2.5 Model Editor

The *featherlite* Model Editor application provides views and actions to monitor and maintain *featherlite* Core Objects at runtime using a set of graphical views. The Model Editor is delivered as a standalone application and provides:

- tree views to display orders, resources and workflows, work calendars as well as parameters, policies, state variables, etc,
- actions and dialogs to create new *featherlite* core objects at runtime,
- actions and dialogs to edit parameters, assign new policies, state variables, etc,
- the application configuration file 'config/clientConfig.xml', defining the basic runtime properties of the application,
- the 'config/log4j.properties' configuration file, defining the logging properties of the application,
- and other common functions.

As the Model Editor is delivered as an application, the files are located in the plugins/-ModelEditor_<version> directory of the application.

3 Creating Custom Projects

In the previous sections we explained which *featherlite* plug-ins exist, and which plug-ins can be used and extended for what purpose to build new projects using the *featherlite* platform.

In general, individualized plug-ins for a specific project have their own policies, services, queries etc. and some other — non *featherlite* — classes or libraries as well.

In what follows we explain the steps required to create a custom server and a custom client plug-ins based on the *featherlite* code-base. A separation of the client and the server component is not necessary in every application area. Obviously, this is the more general application scenario we start with, leaving the simpler case of a client-only application to the reader.

3.1 Server

Since *featherlite* plug-ins are Eclipse plug-ins, we can set the server plug-in up by using an Eclipse wizard. The steps to perform, which are explained in detail in the following, are:

1. create the server plug-in,
2. add the *featherlite* plug-in as a dependency,
3. configure the plug-in,
4. add the application extension points and define the application class,
5. add the code to start the *featherlite* instance to the application class,
6. create the required directory to store the configuration files in,
7. add the *featherlite* configuration files,
8. add a *featherlite* XML model file.

As an alternative, you can also take the server template available on our website for development and include it into eclipse. Details on how to do this are given in Section 3.1.6. In this way, you'll skip the steps above and can start the server immediately.

3.1.1 Creating the server plug-in

To set the server plug-in up, proceed as follows:

- from the 'File' menu, select 'New' and 'Other',
- from the 'New' dialog select 'Plug-in Project' which is in the 'Plug-in Development' folder (see Figure 11)

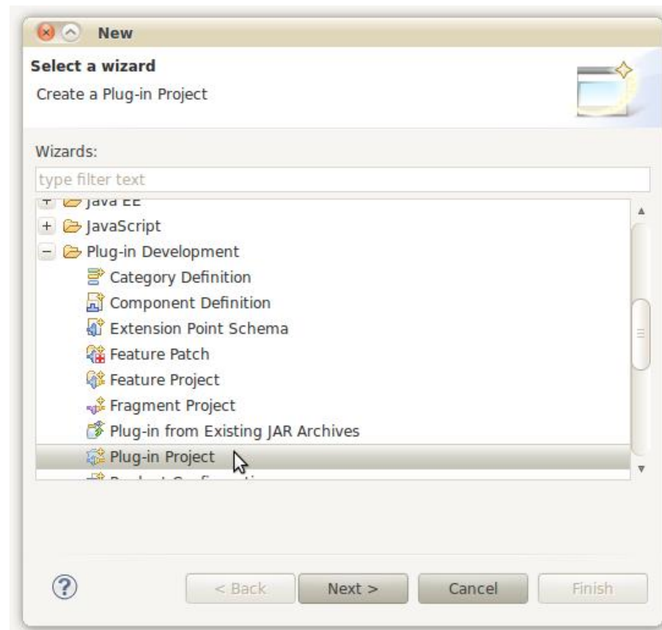

Figure 11: Screen shot of the 'New' dialog to select the type of project to create.

In the 'New Plug-in Project' dialog set

- your preferred project name and project location,
- check the 'Create a Java project' check box,
- select the current eclipse version to be the target platform,
- add it to your working set, if required,
- and press next to continue (see Figure 12).

In the second page of the 'New Plug-in Project' wizard

- type the ID of the plug-in to be created,
- adjust the settings of the version, name and the provider according to your needs,
- select the execution runtime environment to be used with the plug-in, if you don't want to use the default one,
- check the 'Generate Activator...' check box and type in the name of the plug-in activator class,
- deselect the 'This plug-in will make contributions to the UI' check box
- check the 'No' radio button, since we don't want to create a rich client application,
- press next to continue (see Figure 13).

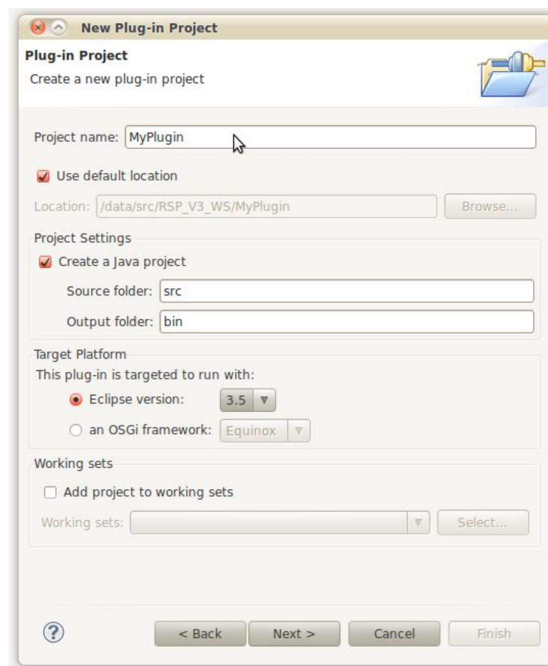

Figure 12: Screen shot of the fist wizard page of the 'New Plug-in Project' dialog to configure the plug-in to be created.

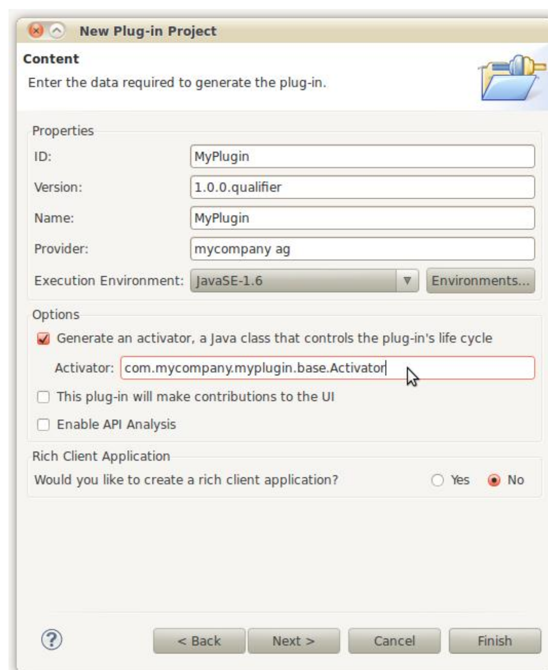

Figure 13: Screen shot of the wizard page to configure the plug-in to be created.

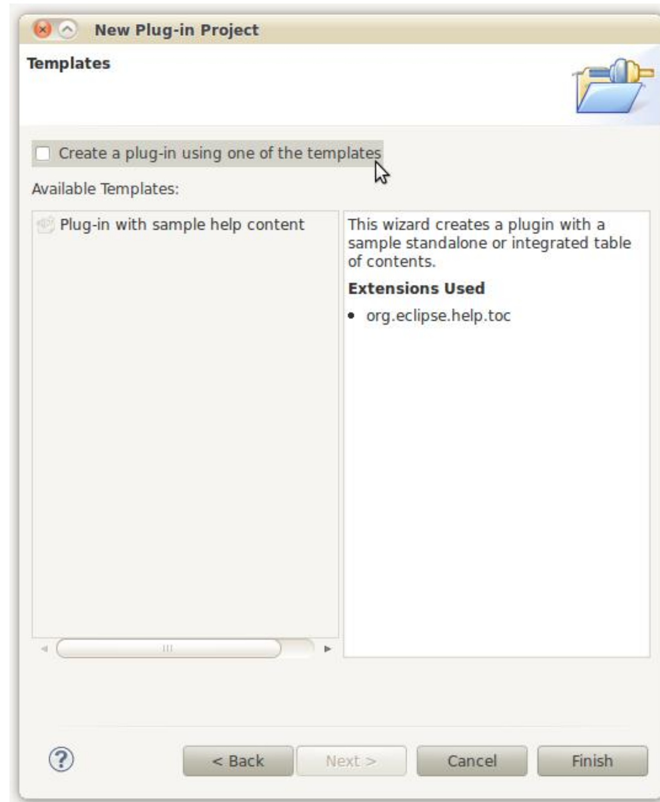

Figure 14: Screen shot of the wizard page to configure the plug-in to be created. For the server plug-in we don't want to add any UI components and have to deselect the check box to create the plug-in from a template.

Finally, in the last page of the 'New Plug-in Project' wizard, uncheck the 'Create a plug-in using one of the templates' check box (see Figure 14).

After pressing the 'Finish' button the IDE automatically creates the plug-in body with the required configuration data as well as a single plug-in activator class.

3.1.2 Add the *featherlite* plug-in dependency

The server plug-in created so far does not yet have references to the *featherlite* or the eclipse plug-ins.

The plug-in configuration is held in the 'MANIFEST.MF' file which is kept in the 'META-INF' directory. To add the *featherlite* plug-in as a dependency, open the 'MANIFEST.MF' file by double clicking on it. Should the source of the file open in a text editor, then close the view and use the right mouse button on the file and under 'Open With' choose the 'Plug-in Manifest Editor' entry.

Switch to the 'Dependencies' tab and use the 'Add. . . ' button to add the plug-in 'RSP' as a dependency (you may have to type "RSP" to see the available plug-ins).

If this project is going to use the free object oriented database from Berkeley, then also add the 'JEPlugin' as a dependency.

Figure 15 shows the defined dependencies.

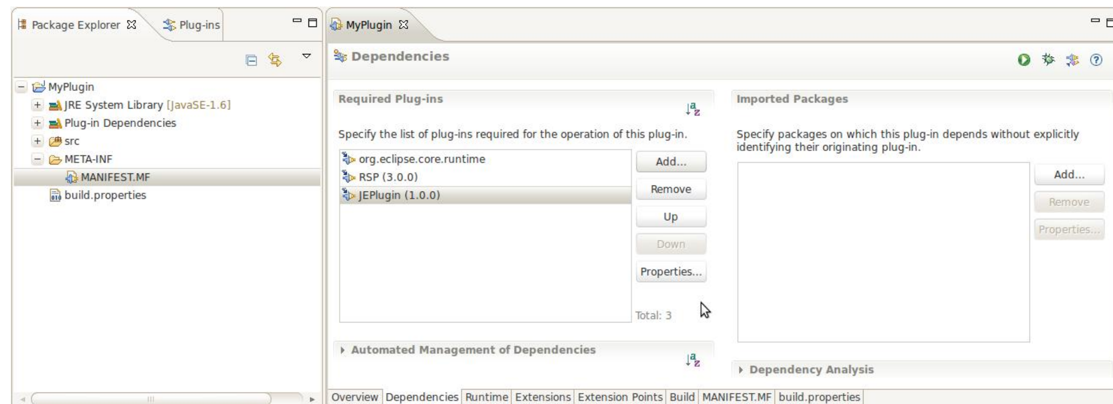

Figure 15: Add the *featherlite* plug-in and the required eclipse runtime plug-ins to the dependency list of your new server plug-in.

3.1.3 Plug-in configuration

Change to the 'Runtime' tab. In the "Classpath" section of the configuration window, create a new library (button "New") with the name of the custom project e.g. 'MyProject.jar'. Do not delete the additionally created library '.' as this is needed when the project is exported as a plug-in (See Figure 16).

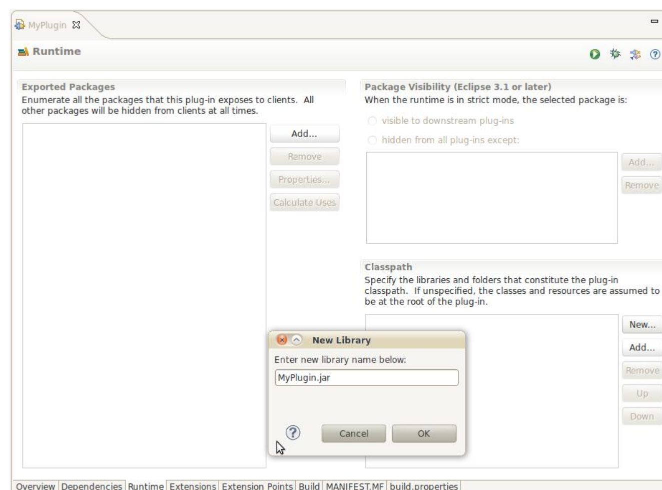

Figure 16: Add the 'MyPlugin.jar' library as a runtime dependency.

An important step in the plug-in configuration is to register the *featherlite* plug-in as an eclipse buddy. This is required, since the OSGI framework (on which eclipse is based) runs each plug-in in its own class loader instance, if not otherwise declared. Since the

featherlite runtime components will in general call classes located in the plug-in created above, we will declare the plug-in to be a *featherlite* buddy, which ensures that classes from both plug-ins will be managed by the same class loader instance [5].

To do so, switch to the 'MANIFEST.MF' tab in the 'Plug-in Manifest Editor' and add the line 'Eclipse-RegisterBuddy: RSP' (see Figure 17).

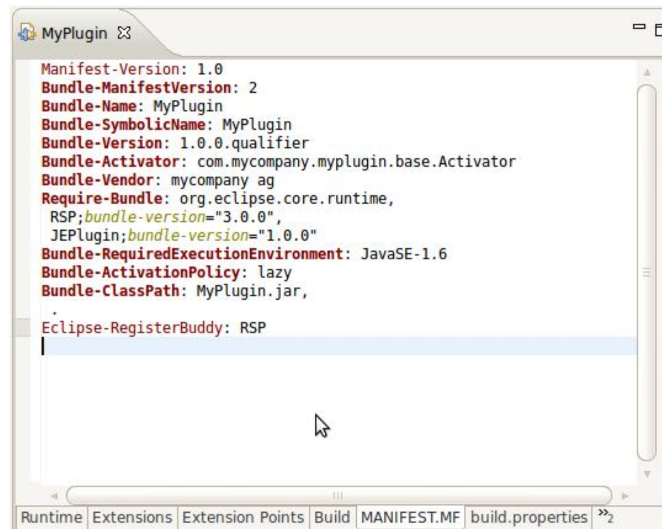

Figure 17: Add the last line to the plug-in manifest to register this plug-in as a *featherlite* buddy.

3.1.4 Application Extension Point

For the plug-in to be an executable eclipse application we need to add an application extension point which references a class implementing the 'org.eclipse.equinox.app.IApplication' interface.

To add the extension point perform the following:

- open the 'Extensions' tab in the Manifest Editor of the project.
- press the 'Add...' button to open the 'New Extension' dialog
- type 'application' in the filter text field and select the 'org.eclipse.core.runtime.applications' extension point (see Figure 18), and "Finish" the dialog,
- right click the '(application)' sub element of the application extension point and select 'New' → 'run' (see Figure 19)

[5] If you miss to register the plug-in as a *featherlite* buddy, you will get an error message whenever *featherlite* tries to call a class from your plug-in. The error message is somewhat misleading, since it's caused by the missing security manager, which is due to the fact that OSGI tries to load classes from other plug-ins via Remote Method Invocation (RMI), if they are not in the same plug-in or not configured to be a buddy.

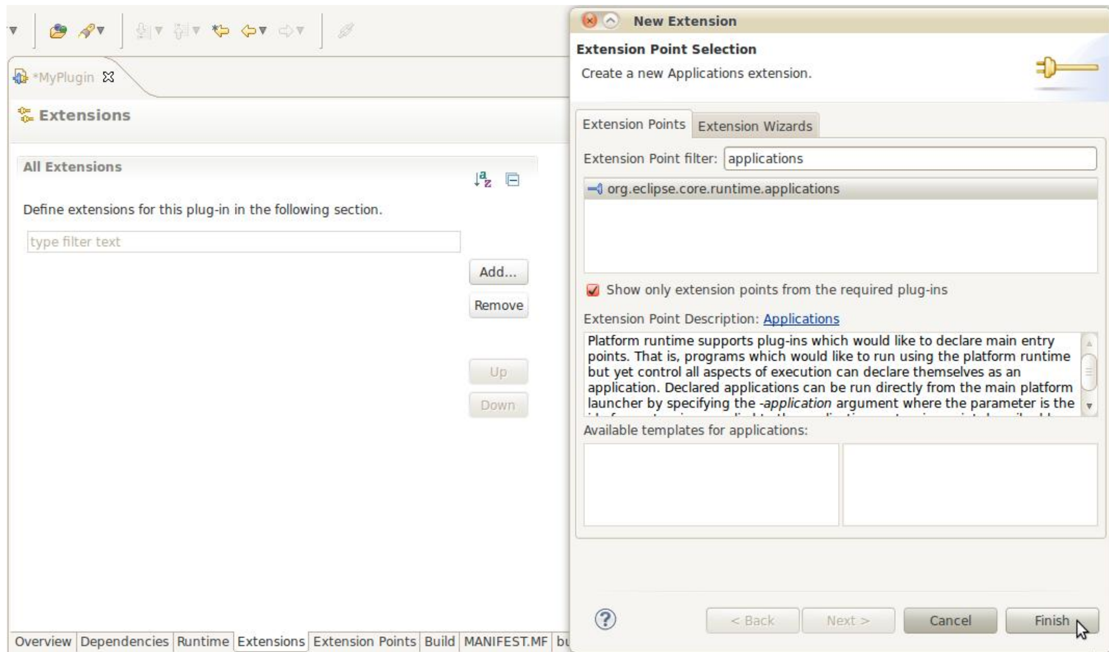

Figure 18: Adding application extension point to plug-in.

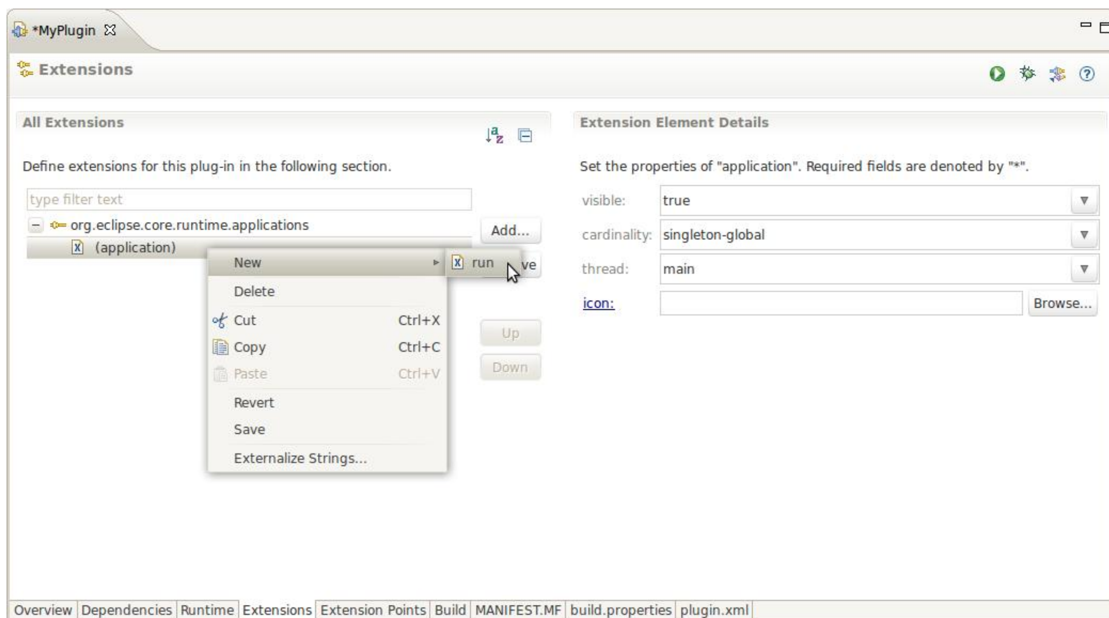

Figure 19: Adding the run property to the application extension point 1/2.

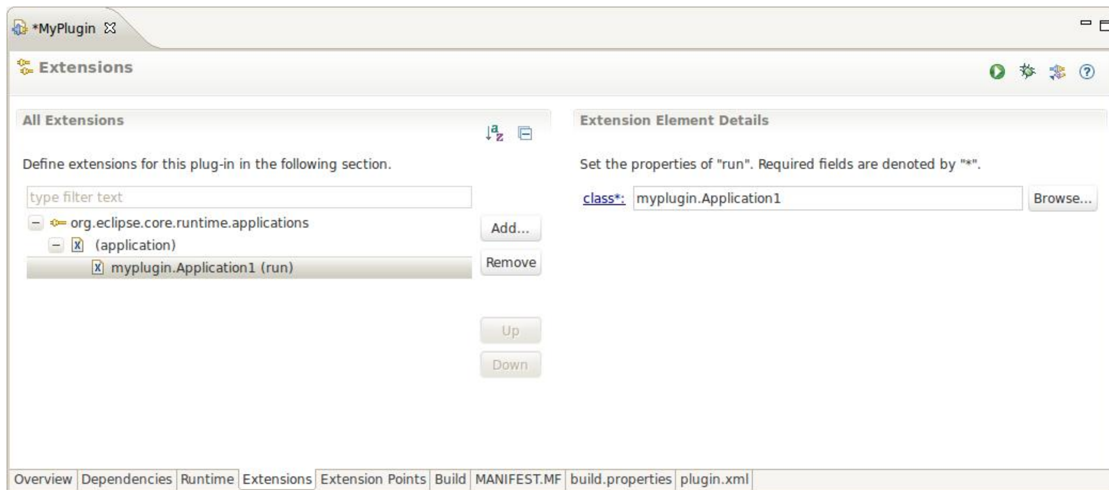

Figure 20: Adding the run property to the application extension point 2/2.

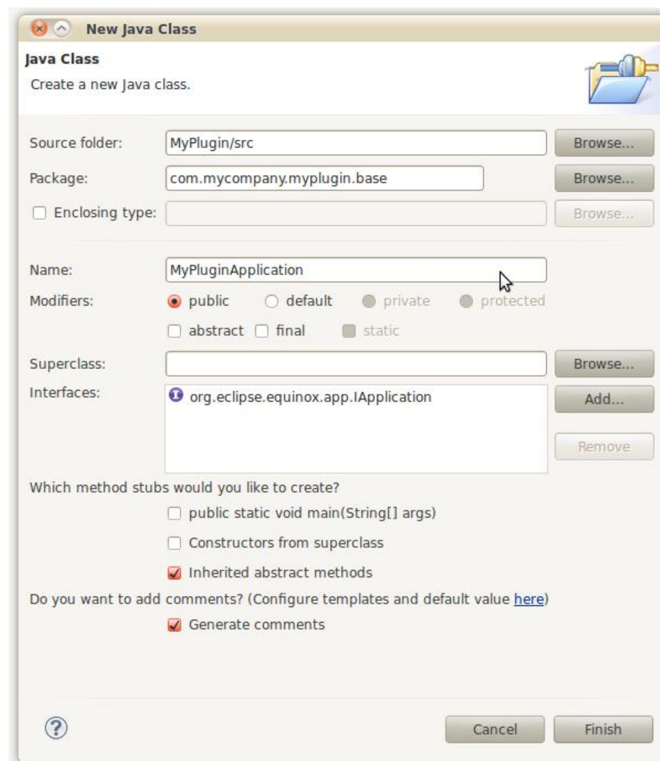

Figure 21: Create the application class.

- Now create the new application class by clicking on the 'class*' on the right hand side of the view (see Figure 20); you can set the name of the class in the next step.
- In the 'New Java Class' dialog fill in the Package and Name fields to match your needs[6] and click the Finish button (see Figure 21)

The newly created application class contains the two self-generated empty methods *start* and *stop*. To have a working *featherlite* plug-in, you must complete the methods as shown in Listing 1 (with the implementation-specific imports classes given).

```
1  import java.io.File;
2  import java.net.URL;
3
4  import org.eclipse.core.runtime.FileLocator;
5  import org.eclipse.equinox.app.IApplication;
6  import org.eclipse.equinox.app.IApplicationContext;
7  import org.osgi.framework.Bundle;
8
9  import com.rsp.core.base.init.KeepAliveMutex;
10 import com.rsp.core.base.init.RSPStarterInit;
11
12 public class MyServerApplication implements IApplication {
13
14
15   /**
16    * @see org.eclipse.equinox.app.IApplication#start(org.eclipse.equinox.app.IApplicationContext)
17    */
18   @Override
19   public Object start(IApplicationContext context) throws Exception {
20
21     // put project specific code here
22
23     // find the root of the RSP
24     Bundle bundle = Activator.getDefault().getBundle();
25     URL relativeURL = bundle.getEntry("/");
26     URL localURL = FileLocator.toFileURL(relativeURL);
27     File root = new File(localURL.getPath());
28     RSPStarterInit.startRSP(root);
29
30     // block this thread for the application up-time
31     KeepAliveMutex.getInstance().block();
32
33     return null;
34   }
35
36   /**
37    * @see org.eclipse.equinox.app.IApplication#stop()
38    */
39   @Override
40   public void stop() {
41
42     // put project specific code here
43
44     // release the pseudo sleep
45     KeepAliveMutex.getInstance().setRun(false);
46   }
47 }
```

Listing 1: Starting a *featherlite* instance from an Eclipse application project.

[6] Here, the package and class names are irrelevant for *featherlite*; however, they are probably important to you to keep the project nicely structured.

The *start* method has the following lines of code to start *featherlite*:

- lines 24-27 - find the root path of the plug-in. This is required for *featherlite* to be able to locate the configuration files,
- line 28 - start the *featherlite* plug-in,
- line 31 - finally use a *featherlite* provided mutex class to block the application to wait for the shutdown.

The *stop* method is simpler. Here we only have the line of code to unblock the application thread by notifying the *featherlite KeepAliveMutex* in line 45.

Note that you may have to save the MANIFEST.MF file, since it's extended with an attribute when the application is created.

3.1.5 Configure the *featherlite* runtime environment

The *featherlite* runtime used by the plug-in code needs some configuration data that is read at *featherlite* initialization. This data is stored in fixed relative locations to the root folder of the plug-in, that is, at the same level where the "src" folder and the "META-INF" folders are located.

The *featherlite* runtime configuration files are stored in the following folders:

models the folder to store your *featherlite* object model configurations,

config the folder for the *featherlite* configuration files,

dist the folder containing any licence files,

logs the folder in which temporary log files are kept. This directory is mostly used in development as log4j can be configured to save the log files anywhere.

Create these folders by right clicking on the plug-in "MyPlugin" in the package explorer view of Eclipse. Select the "New → Folder" menu entry, leave the parent folder set to your plug-in folder and add the folder names given above.

To configure *featherlite* runtime we have to add some configuration files to the *config* folder.

RSPConfig.xml the core configuration file of the *featherlite* platform,

RSPContainer.xml the container configuration declaring the handler components to be used with this installation and defining the start and stop sequence,

policy.xml the configuration of the key to class mapping of the *featherlite* policy handler,

formulas.xml the configuration of the key to class mapping of the *featherlite* formula handler,

log4j.properties the configuration file of the log4j logging framework used by *featherlite*.

For the configuration files, as a starter you can use *RSPConfig.xml, policy.xml, formulas.xml* and *log4j.properties* as they are in the *config* folder of the Server Template plug-in.

For the *RSPContainer.xml* it's a little more complex. There are several pre-configured samples of them in the config folder of the Server Template Plug-In. If unsure, you can start by taking the *RSPContainer.xml* file from the samples. This will have the server start in planning context without persistence. The other files are pre-configured for the server to run with or without the JEPlugin for persistence, and for either planning or execution context. So, you may also select the one appropriate to you (the suffix in the file name should make clear which one you need), place the file in the config folder of your plug-in and rename the file *RSPContainer.xml*. Be aware that you can only use the files pre-configured to run with the JEPlugin if you have added the JEPlugin as a dependency (see Section 3.1.2). Moreover, to run the JEPlugin you must also copy the *applicationConfig_with_JEPlugin.xml* from the Server Template config folder into the *config* folder of your plug-in as *applicationConfig.xml*, and you need to activate the persistence layer in the *RSPConfig.xml* file (by setting the *rsp.database* key to 'on').

By using these files we are done and have created a runnable plug-in which you can use as the starting point to implement your own code using, or extending the *featherlite* core components.

You will still have to adapt the *featherlite* runtime and the *featherlite* object model configuration according to the requirements of your project. How this can be done is explained elsewhere in the *featherlite* documentation and is beyond the scope of this getting started guide.

If you want to build your own individualized rich client user interfaces, you have to set a client plug-in up, which we explain in Section 3.2.

3.1.6 Using the ServerTemplate plug-in as a server

As stated earlier, you can have a quick start by using the ServerTemplate plug-in instead of configuring your own. This has the advantage that you have to perform only minimal configuration to start working, with the drawback that the plug-in is called ServerTemplate [7].

To this aim, you proceed as follows. Start by retrieving a copy of our ServerTemplate plug-in from our website [8]. Once you have it, unpack the zip file to one of your folders. You should now see that there is an additional folder, called ServerTemplate, in this folder. The ServerTemplate folder contains the plug-in we're installing.

Now, copy the whole ServerTemplate folder to your eclipse workspace folder. If you're unsure what this might be, you can check it by selecting Help → About Eclipse SDK→

[7] Of course, as an expert eclipse developer, you'll be able to change this name afterwards.
[8] http://www.featherlite-framework.org/downloads

Installation details→ Configuration, and looking at the folder configured under the key "osgi.instance.area". For example, assume that the entry for that configuration reads "osgi.instance.area=file:/data/development/workspace/". Then, you shall copy the template to the "/data/development/workspace/" folder. Be aware that you have to keep the folder name ("ServerTemplate") for things to work.

Once you've done this, start eclipse. In eclipse, select the "File → New → Other" menu entry and, in the wizard that opens, select "Plug-In Project" (as shown in Figure 11), and click "Next". As project name in the "Plug-in Project" page of the wizard, enter ServerTemplate, that is, the same name as the folder we just copied in the eclipse workspace, and proceed to the next page. In the "Content" page of the wizard, you need to:

1. Disable the "Generate an activator, a java class that controls the plug-in's life cycle" option, since the template already has such a class;
2. Disable the "This plug-in will make contributions to the UI" option;
3. Set the "Would you like to create a rich client application" radio button to "No".

It's important that you perform these steps, as the plug-in will not work otherwise. Click "Next", and in the following "Template" page of the wizard, disable the "Create a plug-in using one of the templates" option, and click "Finish".

The ServerTemplate should now load, and display one error in the "MANIFEST.MF" file of the "META-INF" folder. To correct it, open this file by double-clicking on it. Switch to the "Overview" tab of the view and select the "This plug-in is a singleton" checkbox and then save the changes.

In this same view, clicking on the "Launch an Eclipse Application" link will start the ServerTemplate.

As a final step, we must set the "i18n" folder that contains the data for the internationalization of the plug-in as a source folder. To that aim, right click on the "i18n" folder of the "ServerTemplate" in the Navigator view, select "Build Path"→"Use as Source Folder".

Now, you're ready to proceed to create a rich client plug-in (Section 3.2) or to run the template server (Section 4.1).

3.2 Client

Since *featherlite* plug-ins are Eclipse plug-ins, we can set up the rich client plug-in using Eclipse wizards. The steps to perform are:

1. create the rich client plug-in using a template,
2. add the *featherlite* plug-ins as dependencies,
3. add the *featherlite* configuration files,
4. add the code to configure the application,
5. configure the 'WorkbenchAdvisor',
6. configure the 'WorkbenchWindowAdvisor'.

Similar to the server you can, as an alternative, take the client template available on our website and include it into eclipse. Details on how to do this are given in Section 3.2.7. In this way, you'll skip the steps above and can start the server immediately.

3.2.1 Create the rich client plug-in using a template

To setup the rich client plug-in, proceed as follows:

- from the 'File' menu, select 'New' and 'Other',
- from the 'New' dialog select 'Plug-in Project' which is in the 'Plug-in Development' folder (see Figure 22)

In the 'New Plug-in Project' dialog set

- your preferred project name and project location,
- check the 'Create a Java project' check box,
- select the current eclipse version to be the target platform,
- add it to your working set, if required,
- and press next to continue (see Figure 23).

In the second page of the 'New Plug-in Project' wizard

- type in the ID of the rich client plug-in to be created,
- adjust the settings of the version, name and the provider according to your needs,
- select the execution runtime environment to be used with the plug-in, if you don't want to use the default one,

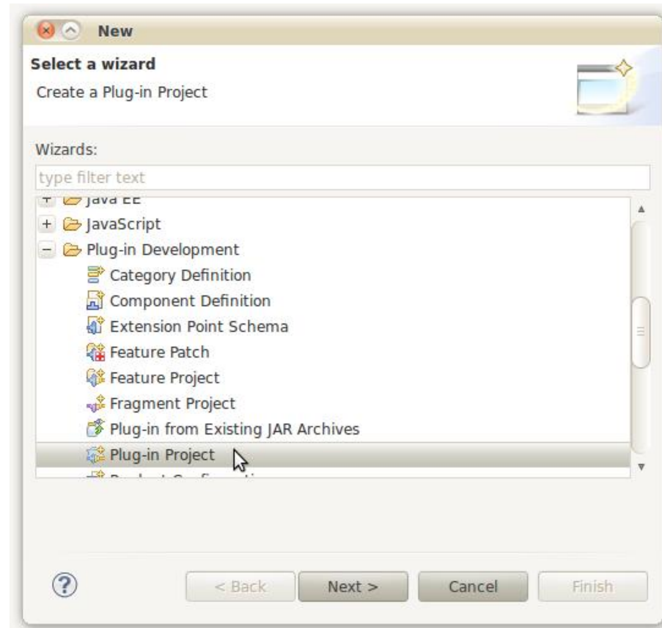

Figure 22: Screen shot of the 'New' dialog to select the type of project to create.

- check the 'Generate Activator. . . ' check box and type in the name of the plug-in activator class,
- select the 'This plug-in will make contributions to the UI' check box
- check the 'Yes' radio button, since we are creating a rich client application,
- press next to continue (see Figure 24).

In the next page of the 'New Plug-in Project' wizard select 'RCP application with a view' (see Figure 25).

Finally, in the last page of the 'New Plug-in Project' wizard configure the window title of the application and define the name of the application class and its package (see Figure 26).

3.2.2 Add the *featherlite* plug-ins as dependencies

The rich client plug-in created so far does not yet have references to the *featherlite* plug-ins.

Plug-in configuration is held in the 'MANIFEST.MF' file which is kept in the 'META-INF' directory. To add the *featherlite* RSPCore plug-in as a dependency, open the 'MANIFEST.MF' file by double clicking on it. Should the source of the file open in a text editor, then close the view and use the right mouse button on the file and under 'Open With' choose 'Plug-in Manifest Editor'.

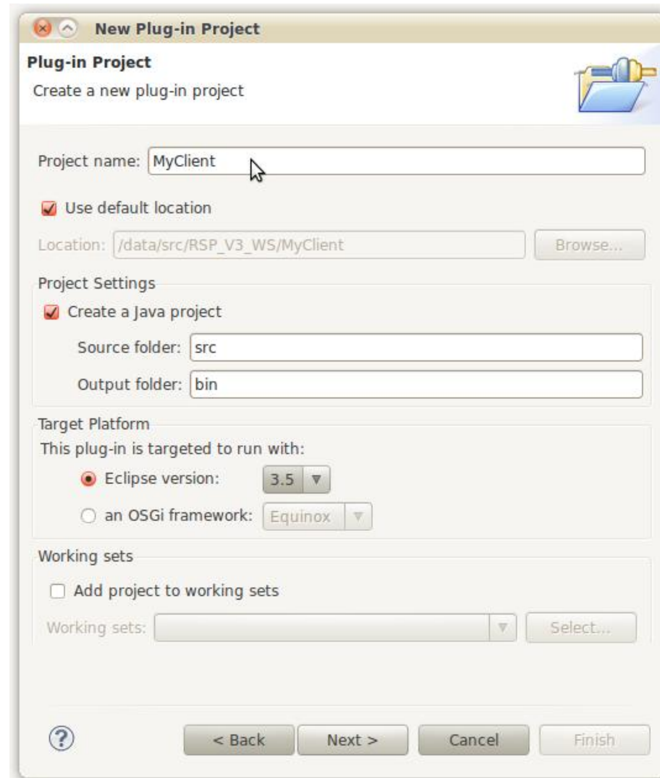

Figure 23: Screen shot of the fist wizard page of the 'New Plug-in Project' dialog to configure the plug-in to be created.

Switch to the 'Dependencies' tab and use the 'Add. . . ' button, type 'RSP' in the selection and add the plug-in 'RSPCore' as a dependency. If this rich client is to be used in a planning context, then also add 'PlanningClientCore' as a dependency, or for an execution context also add the 'ExecutionClientCore' dependency.

Figure 27 shows the defined dependencies.

3.2.3 Add the *featherlite* configuration files

The *featherlite* client runtime used by the plug-in code needs some configuration data. As is the case for the server part we discussed in the previous section, this data is stored in fixed relative locations to the root folder of the plug-in, that is, at the same level where the "src" folder is located:

config the folder for the *featherlite* client configuration files,

logs the folder in which temporary log files are kept. This directory is mostly used in development as log4j can be configured to save the log files anywhere.

Again, create these folders by right clicking on the client plug-in in the package explorer view of Eclipse. Select the "New → Folder" menu entry, leave the parent folder set to

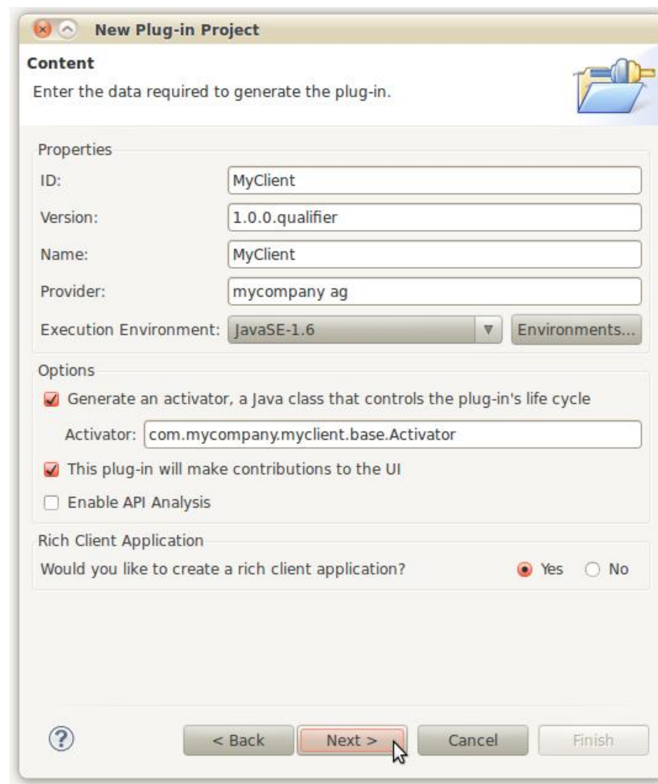

Figure 24: The form to add the data required to generate a new rich client plug-in by selecting '... make contributions to the UI' and 'Yes' to '... create a rich client application'.

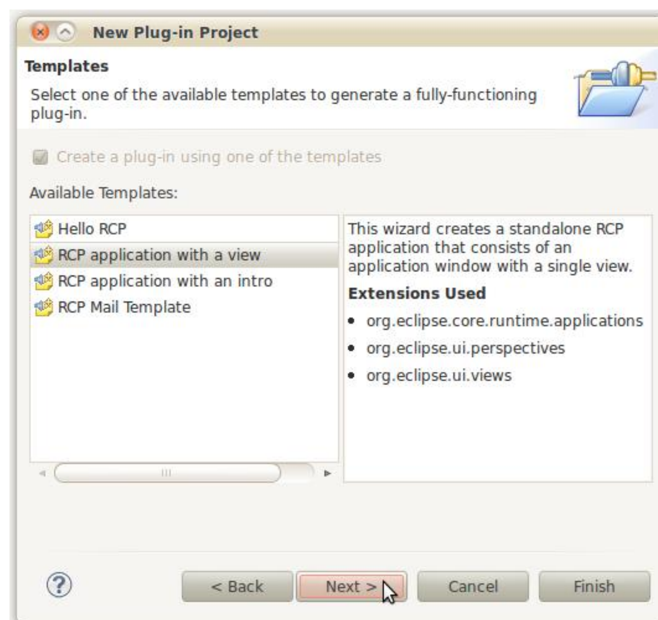

Figure 25: Create a new rich client plug-in with a view.

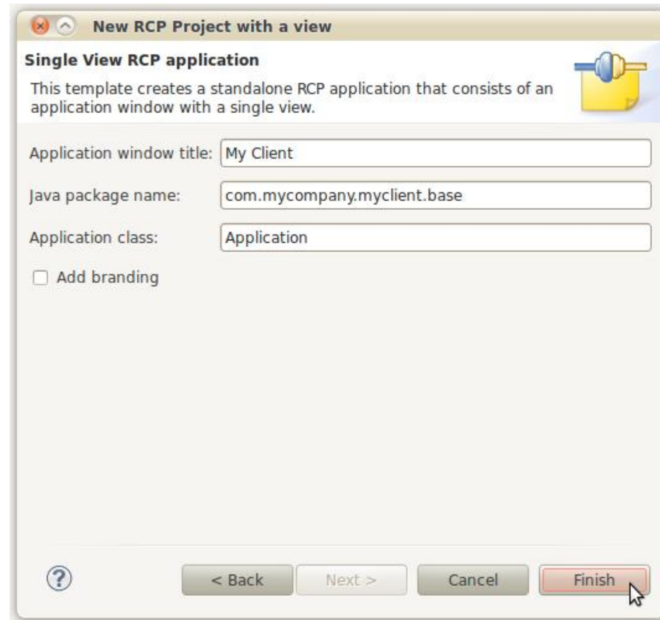

Figure 26: Define the rich client application details.

your plug-in folder and add the folder names given above.

To configure *featherlite* client runtime we have to add some files. For starters, they are best copied from the Client Template. The following files are required:

clientConfig.xml the client configuration file,

log4j.properties the configuration file of the log4j logging framework used by *featherlite*.

The files from the Client Template plug-in are pre-configured to work with the server started on the same host. Thus, if you are using the sample configuration files from the Client Template for your client, and the server uses the configuration files from the Server Template plug-in, the two components are already configured to work together.

3.2.4 Add the code to configure the application

In the 'Application' class which was generated at plug-in creation, add the following line just after the display is created, but before the workbench is created and run:

```
ClientConfigLoader.loadClientConfig(Activator.getDefault());
```

This line of code will read the configuration files and initialize any *featherlite* client specific configuration needed.

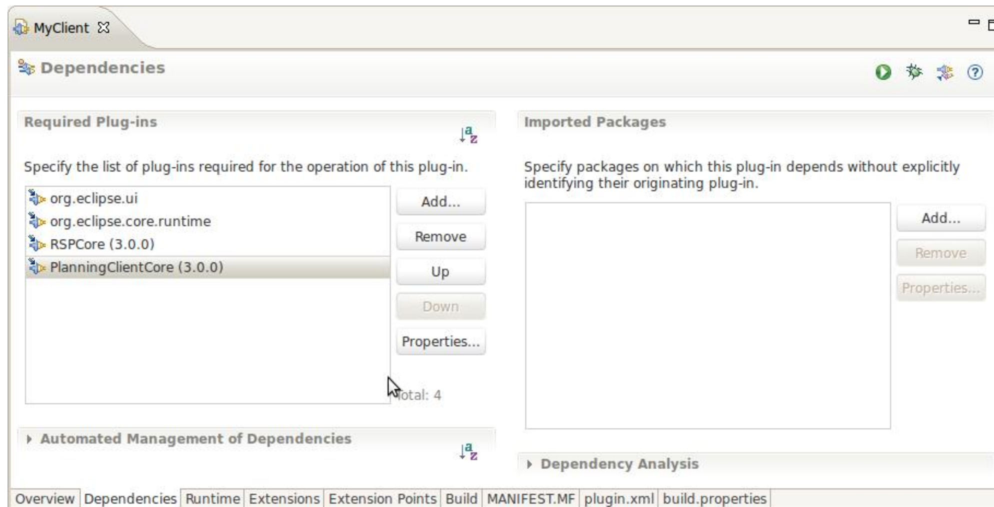

Figure 27: Add *featherlite* dependencies to the rich client plug-in e.g. the RSPCore and PlanningClientCore *featherlite* plug-ins.

3.2.5 Configure the 'WorkbenchAdvisor'

The 'ApplicationWorkbenchAdvisor' class controls how the application window is created, what default perspective is used, and so on.

If this client plug-in is for a planning context, then return *com.rsp.planning.client.base.RSPPlanningPerspective.ID* in the *getInitialWindowPerspectiveId()*-method.

If this client plug-in is for an execution context, then return *com.rsp.execution.client.base.RSPExecutionPerspective.ID* in the *getInitialWindowPerspectiveId()*-method.

3.2.6 Configure the 'WorkbenchWindowAdvisor'

The 'WorkbenchWindowAdvisor' defines window specific details. In the *preWindowOpen()*-method one would define the size of the window or set the application title.

Further in the *createActionBarAdvisor()*-method is a good place to see which menus are created and is propably the starting point to add further actions to the client plug-in.

3.2.7 Using the ClientTemplate plug-in as a client

As stated in Section 3.2, you can use *featherlite*'s ClientTemplate plug-in as a starting point for your plug-in, thus requiring only minimal initial configuration.

To do so, retrieve a copy of the ClientTemplate plug-in from our website[9]. Once you

[9] http://www.featherlite-framework.org/downloads

have it, unpack the zip file to one of your folders. You should now see that there is an additional folder, called ClientTemplate, in this folder. The ClientTemplate folder contains all data for the plug-in we're installing.

Now, copy the whole ClientTemplate folder to your eclipse workspace (if unsure, see Section 3.1.6 to determine where this is). Be aware that you have to keep the folder name ("ClientTemplate") for things to work.

Once you've done this, start eclipse. In eclipse, select the "File → New → Other" menu entry and, in the wizard that opens, select "Plug-In Project", and click "Next". As project name in the "Plug-in Project" page of the wizard, enter ClientTemplate, that is, the same name as the folder we just copied in the eclipse workspace, and proceed to the next page. In the "Content" page of the wizard, you need to:

1. Disable the "Generate an activator, a java class that controls the plug-in's life cycle" option, since the template already has such a class;
2. Enable the "This plug-in will make contributions to the UI" option;
3. Set the "Would you like to create a rich client application" radio button to "No".

It's important that you perform these steps, as the plug-in will not work otherwise. Also, note that you *really* have to set the last point to "No" even though we are creating a rich client application: the template already contains the information that would be added with this option. Click "Next", and in the following "Template" page of the wizard, disable the "Create a plug-in using one of the templates" option, and click "Finish".

The ClientTemplate should now load, and display one error in the "MANIFEST.MF" file of the "META-INF" folder. To correct it, open this file by double-clicking on it. Switch to the "Overview" tab of the view and select the "This plug-in is a singleton" checkbox and then save the changes.

In this same view, clicking on the "Launch an Eclipse Application" link will start the ClientTemplate.

As a final step, we must set the "i18n" folder that contains the data for the internationalization of the plug-in as a source folder. To that aim, right click on the "i18n" folder of the "ClientTemplate" in the Navigator view, select "Build Path"→"Use as Source Folder".

Now, you're ready to run the plug-ins.

4 Running an Eclipse Plug-in

In what follows we explain the configuration to run an application plug-in inside the Eclipse IDE by using the previously created custom projects as a running example.

4.1 Running the Server Plug-in

There are two ways to start the plug-in server. The first one (Section 4.1.1) is the single-click quick and easy way, but you do not know what happens in the background. The second one (Section 4.1.2) is a manual setup of the run configuration, which can be used if the easy way does not work or to understand what gets configured in the background.

4.1.1 Starting the server through plugin.xml

To start the server through the 'plugin.xml' file, select the 'plugin.xml' file by double-clicking on it and switch to the 'Overview' tab. In the Testing part of the overview, select the "Launch an Eclipse application" link. Through this action, the Console log output should come into focus, and at last, you should be able to read the following message: ***** RSP finished startup and is ready to use :-) *****

4.1.2 Setting up a Run Configuration Manually

The method described below gives you a finer control on how the application can be started, and can be used if something goes wrong or you need additional configuration to run the server.

To create the run configuration for *featherlite* server plug-ins from inside eclipse IDE, proceed as follows:

1. Open the 'Run Configuration' dialog from the eclipse 'Run' menu,
2. In the 'Run Configuration' dialog select 'Eclipse Application' and, from the context menu, select the 'New' button to create a new configuration (see Figure 28),
3. In the dialog of the new configuration enter a new name in the 'Name' field (Execution Server for example).
4. In the 'Main' tab select 'Run an applicaton' in the 'Program to Run' setting and select the application with the name you created – in our case 'MyPlugin.id1' – from the selection list (see Figure 29).
5. In the 'Arguments' tab, add '-consoleLog' to the 'Program Arguments' text field (see Figure 30).

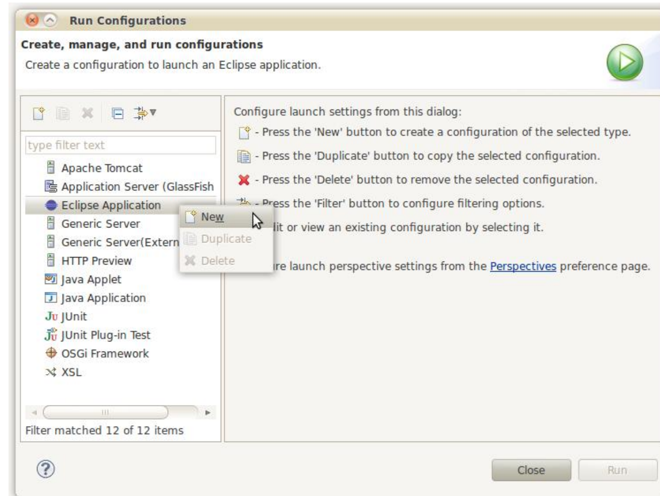

Figure 28: Select Eclipse Application from the Run dialog and press 'New' to create a new configuration.

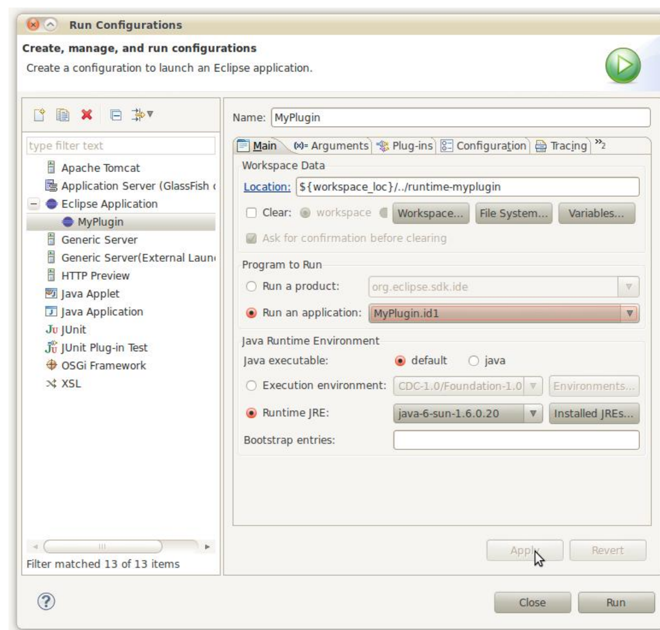

Figure 29: Type in a name, select 'Run an application' and select the application id of the custom project e.g. 'MyPlugin.id1' from the selection list.

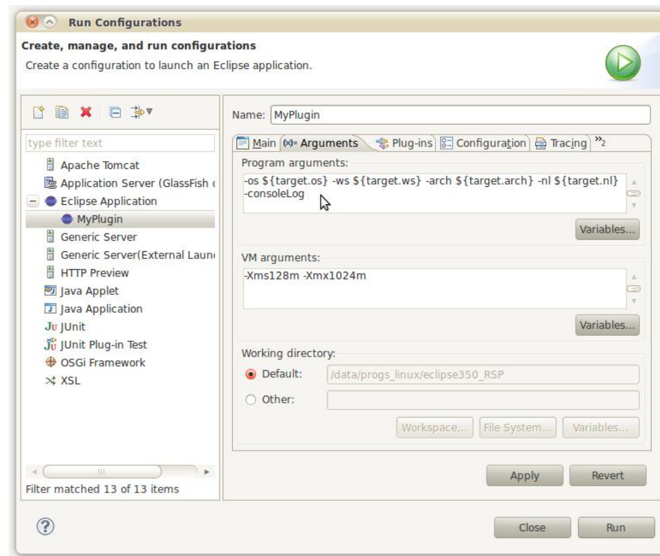

Figure 30: Add '-consoleLog' to the 'Program arguments' field.

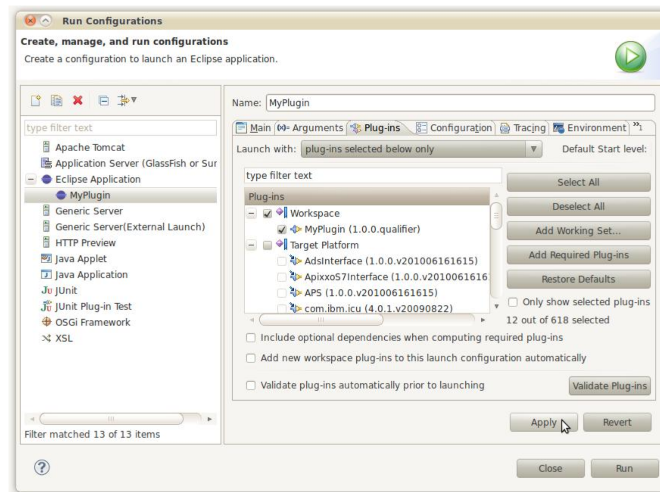

Figure 31: Select the required plug-ins to run.

6. In the ’Plug-ins’ tab, select ’plug-ins selected below only’ from the ’Launch with’ selection list, press ’Deselect All’, deselect ’Add new workspace to this launch configuration automatically’ and finally press ’Add required Plug-ins’ (see Figure 31).
7. Press the ’Apply’ button to save the configuration.
8. Finally press ’Run’ to start the application.

4.2 Running the Client Plug-in

There are again two methods to start the client: through the 'Launch an Eclipse application' link in the client's 'plugin.xml' file (as described in Section 4.1.1 for the server plug-in, we omit a detailed description here) and through the runtime configuration.

In the manual run configuration, a client is generally configured in the same way as servers are, with two significant differences. The first one is that in the 'Main' tab, you must select the Client application plug-in as the application to run ('MyClient' in our example); the second one is that on the 'Main' tab of the run configuration dialog, you should select 'Clear' and 'Ask for confirmation before clearing' in the 'Workspace Data' setting. This option helps to avoid inconsistent plug-in states during development (see Figure 32). For instance, by clearing the workspace data, the size of the window is not saved and used for the next session. The remaining configuration steps (the consoleLog addition and selection of the plug-ins) is the same as for the server.

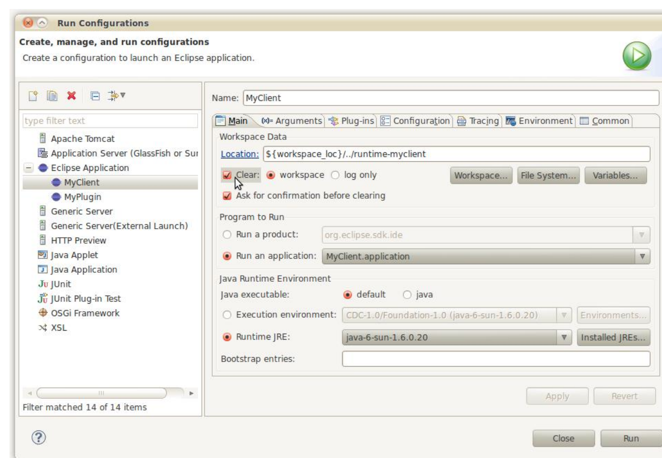

Figure 32: Check the 'Clear' checkbox in a rich client plug-in run configuration.