cosylab

CSS Common Startup Code

Instructions How to Write Product Plug-ins

Revision:	1.4
Status:	Released
Repository:	acc/projects/ORNL/CSS_Study
Project:	ACC-ORNL-CSS_Startup
Folder:	doc
Document ID:	CSL-DES-08-47735
File:	CSS_Common_Startup_Instructions.doc
Owner:	Jaka Bobnar
Last modification:	June 24, 2009
Created:	June 24, 2009

Document History

Revision	Date	Changed/ reviewed	Section(s)	Modification
1.0	2009-04-10	jbobnar	All	Created.
1.1	2009-04-28	jbobnar	All	Revision
1.2	2009-04-29	jbobnar	All	Step By Step configuration instructions expanded, Grammar checking, revision
1.3	2009-05-11	kkasemir	All	The parts about the Plugin Activator removed
1.4	2009-06-24	jbobnar	All	Updates regarding the base plug-in split into startup and startuphelper plug-ins

Confidentiality

This document is classified as a **`public document`**. As such, it or parts thereof are openly accessible to anyone listed in the Audience section, either in electronic or in any other form.

Scope

This document describes how a customized product startup plug-in for Control System Studio can be created.

Audience

The audience of this document is:

- All members of the CSS development group at DESY and SNS/ORNL
- All members of development team at Cosylab.

The document can be disclosed to all developers legible to modify the CSS core code and to develop client specific CSS products.

Table of Contents

1. INTRODUCTION

Control System Studio is a rich client application which consists of many Eclipse plug-ins. In order to run the application the appropriate product start-up plug-in has to be started. This product plug-in defines which features will be included in the deployment and how the start-up procedure will be carried out. Therefore, each end user who wishes to run the CSS in a specific way has to provide the product plug-in (either using one which is already provided by the community or create his own). This document describes what the structure of such product plug-in should be and which features are already provided by the community and can be used for making this customized plug-in.

This manual does not explain or teach how to make an eclipse plug-in. It only describes the specifics of the CSS product plug-in and how to easily create such product plug-in. For details about the eclipse plug-in configuration refer to the eclipse manual.

2. CREATING A CUSTOM START-UP PLUG-IN

The start-up plug-in architecture consists of two (or more) CSS specific plug-ins. The first one is the base start-up plug-in which is named *org.csstudio.startup* and provides the common code to be used during application startup and tear down. The second plug-in is the end product specific plug-in which 'extends' the base plug-in and provides all the necessary resources to start the Eclipse product. There is no prescription how to name the product plug-in but the preferred way is to use a name in the style of *org.csstudio.<specific name>.product*. Beside the aforementioned two plug-ins another plug-in *org.csstudio.startuphelper* exists which provides default implementation of the interfaces from the *org.csstudio.startup* as well as the resource that might be required by the product plug-in.

2.1. BASE PLUG-INS DESCRIPTION

The base plug-in for all the CSS products is the plug-in *org.csstudio.startup*. This plug-in provides all the common files which are necessary to start the Control System Studio. The plug-in's content is already sufficient to make the basic application. All that is needed is to create a minimal Eclipse plug-in which uses the `org.eclipse.equinox.app.IApplication` [3] implementation provided by this base plug-in. We will later see how this is done. The structure of this base plug-in and the files that can be found inside are as follows:

- `build.properties`: defines which resources will be included in the binary and/or source build of this plug-in
- `plugin.xml`: the plug-in configuration file which defines the extensions and extension points of this plug-in (see section 2.2)
- `META-INF/MANIFEST.MF`: defines the basic properties of the startup plug-in (name, dependencies, exported packages, etc.)
- `schema/org.csstudio.startuphelper.modules.exsd`: the schema definition for the startup extension point of this plug-in (see section 2.2)
- `src`: all the source code is divided into three packages
  - `org.csstudio.startuphelper.application`: default implementation of the `IApplication` interface
  - `org.csstudio.startuphelper.module`: the interfaces which are linked with the *org.csstudio.startuphelper.modules* extension point

When a user wants to develop his own customized product start-up plug-in for CSS he can include/extend this plug-in and use the provided features. Even more, the plug-in *org.csstudio.startuphelper* provides some default implementation of the extension points from the startup plug-in and also some other files and resource that the developer can use:

- `plugin_customization.ini`: the customization file which defines a set of properties used by different plug-ins

- `resources`
  - `icons`: the list of icons which can be used to set up the looks of the product
  - `intro`: the product plug-in can provide the intro or welcome page at the start-up of the application; this folder provides the basic intro page which can be used for this purpose
  - `splash.bmp`: the default splash screen for the product plug-in
- `src`: all the source code is divided into two packages
  - `org.csstudio.startuphelper`: set of externalized messages used by the plug-in
  - `org.csstudio.startuphelper.module`: the default implementations of the extension point interfaces from the parent package

2.2. Extendable Features of Base Plug-in

The startup plug-in provides two extendable entry points.

1. The first one is the implementation of the `org.eclipse.equinox.app.IApplication`, which can be found in the package `org.csstudio.startuphelper.application`, class `Application`. This class defines how the application is started. It provides several methods and each of these methods provides a certain feature which can be loaded by the application. This class also defines the sequence how the features are loaded. For instance, first the startup parameters are read, which is followed by applying the locale settings etc. If not programmed differently the following sequence is used:
   - read startup parameters using the startupParameters extension
   - apply locale settings using the locale extension
   - prompt for login using the login extension
   - prompt for workspace using the workspace extension
   - start services using the service extension
   - open projects using the project extension
   - execute actions before the workbench starts, start workbench using the workbench extension
   - execute actions when the workbench is closed using the workbench extension
   - close projects using the project extension
   - before shut down actions are executed just before the application is stopped

   By extending this class user can specify his own sequence of events or even his own procedures that will be carried out during startup by overriding the appropriate methods from this class (for reference see javadoc).

2. The base plug-in defines an extension point [3] *org.csstudio.startuphelper.modules* through which the user can set several different implementations to be used during startup. By default these implementations are executed by the `Application` class, which is described above. This extension point defines 8 different extensions where each of them is coupled with an interface from the `org.csstudio.startuphelper.extensions` package. These extensions can be used to customize the startup procedure (for the details of each extension refer to the interface's javadoc and to the descriptions in the extension point schema found in `schema/org.csstudio.startuphelper.modules.exsd`):

   - locale (`LocaleSettingsExtPoint`) defines the locale settings that are to be applied to the application.
   - login (`LoginExtPoint`) defines how the users login into the application.
   - project (`ProjectExtPoint`) takes care of the projects' lifecycles. It enables opening/closing a specific project when the application starts/stops.
   - services (`ServicesStartupExtPoint`) is responsible for starting all additional services in the application.
   - shutdown (`ShutDownExtPoint`) is responsible for taking the appropriate actions just before the application closes.
   - startupParameters (`StartupParametersExtPoint`) takes care of reading the application startup parameters.
   - workbench (`WorkbenchExtPoint`) handles the workbench lifecycle as well as constructs a user specific workbench with desired advisors (implementation allows only one extension of this type).
   - workspace (`WorkspaceExtPoint`) enables workspace manipulation at startup.

   By default the Control System Studio does not require any of these extensions. The `Application` will automatically load if whatever is missing and required (workbench startup only). However, if one decides that the he wants to customize the application, there exist default implementations of some of the above extensions. They can be found in the package `org.csstudio.startuphelper.extensions.impl`. The package provides most commonly used implementations of the extension points, which do not expect any specific configuration (for the details about each extension refer to the javadoc).

2.3. PRODUCT PLUG-IN

All the following instructions assume that Eclipse 3.4 or later is being used to configure the plug-ins.

2.3.1. Required Setup

To create a runnable Control System Studio a product plug-in has to be defined. This plug-in will define the actual lifecycle of the application and the features that will be loaded. We will first describe the configurations that one needs to make to create such a plug-in (for details and step-by-step procedure how to make a product plug-in see section 2.4).

The following files are required by the plug-in:

- META-INF/MANIFEST.MF
- build.properties

If using the Eclipse plug-in project creator wizard [3], these two files are generated automatically.

The product plug-in requires some additional libraries (projects), which have to be imported. Basically, only two plug-in dependencies are required. These are the base plug-in *org.csstudio.startup* and *org.eclipse.core.runtime*, which is used to start a rich client application. If the product is going to use an intro page, the *org.eclipse.ui* and o*rg.eclipse.ui.intro* also have to be provided in the dependencies tab. User is free to use other dependencies such as for instance *org.csstudio.startuphelper* if required.

To be able to run the CSS product an application extension *org.eclipse.core.runtime.applications* needs to be set up. This extension defines the `IApplication` implementation that will be loaded when the application starts. If nothing specific is required, user can use the implementation from the base plug-in `org.csstudio.startuphelper.application.Application` (see section 2.2 for details).

This is in fact everything what is necessary to run the most basic version of the CSS. See the next section on how to introduce optional settings.

2.3.2. Optional Setup

2.3.2.1. Application Startup Procedure

To define his own procedures how to read the startup parameters, create the workbench etc. user can use the *org.csstudio.startup.modules* extension point which is described in section 2.2. An arbitrary number of extensions of each type can be added to the product plug-in. For each of the extensions an implementation needs to be provided, which will be executed by the `Application` (if user did not specify his own implementation of the `IApplication` interface). User is free to provide his own implementations of the extension points' interfaces or he can use the default ones provided by the base plug-in.

2.3.2.2. Welcome Page

If user wants to display a welcome page when the application starts, he has to make sure that the appropriate plug-ins are listed in the dependencies (*org.eclipse.ui* and *rg.eclipse.ui.intro*). After that we have to add two extensions: org.eclipse.ui.intro and org.eclipse.ui.intro.config and properly configure their values. The basic configuration of the necessary extensions is made automatically when configuring the product configuration file (for references see these two extension point's documentation [3]).

2.3.2.3. Branding

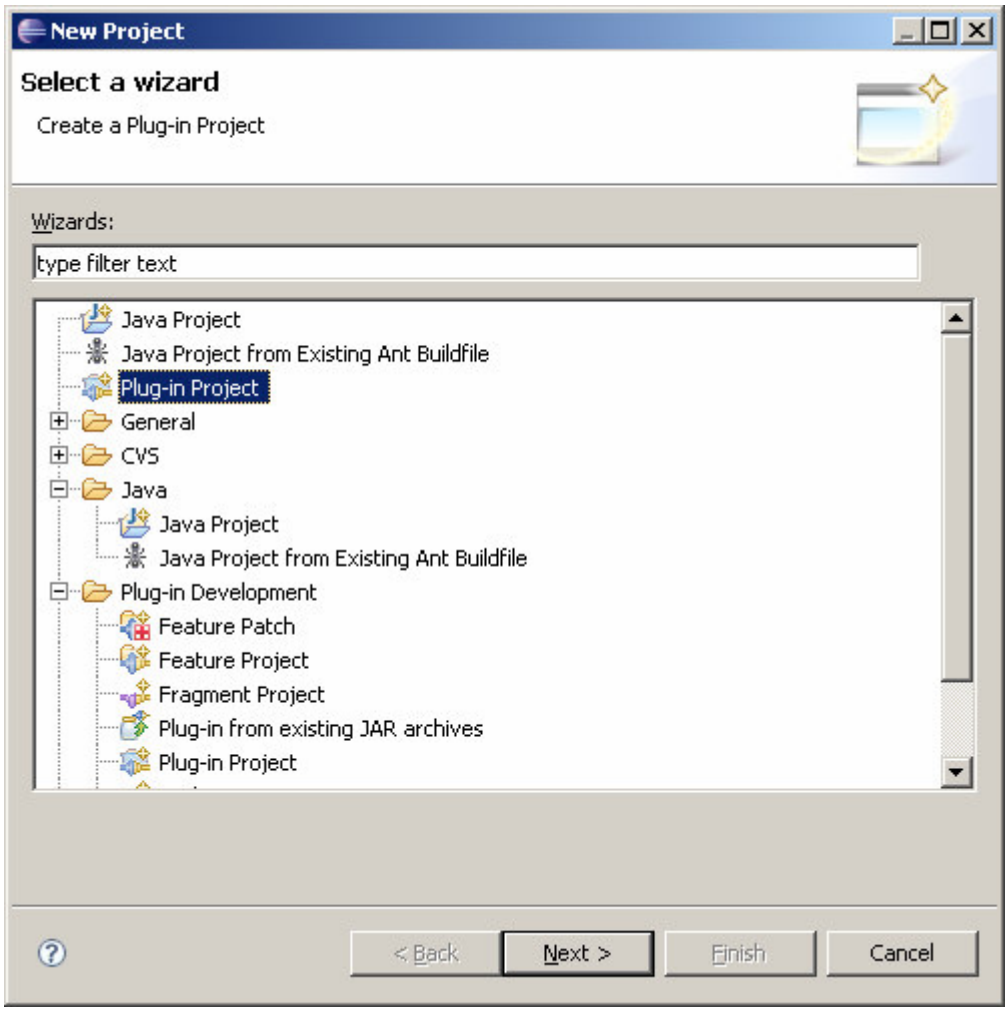
If application is required to have more professional looks the plug-in can be configured to use specific branding such as splash screen, about text, about text, welcome page, icons etc. For this reason one needs to setup the *org.eclipse.core.runtime.products* extension. Through this extension user can set up different properties used by the application (for details refer to the extension point documentation [3]). For more extensive branding, user can also specify the `<name>.product` file (product configuration file), where the `<name>` is the name of the product. The file should be located in the root of the plug-in project. By properly configuring this file the user can set his own icons, license, splash… to be used by the application (some details how to set these are described in section 2.4, otherwise refer to the eclipse plug-in documentation [3]). By configuring this product file, the necessary extensions will be automatically added to the `plugin.xml` file.

In addition to the branding, the `.product` file also defines the configuration (properties) file which is to be loaded at start-up.

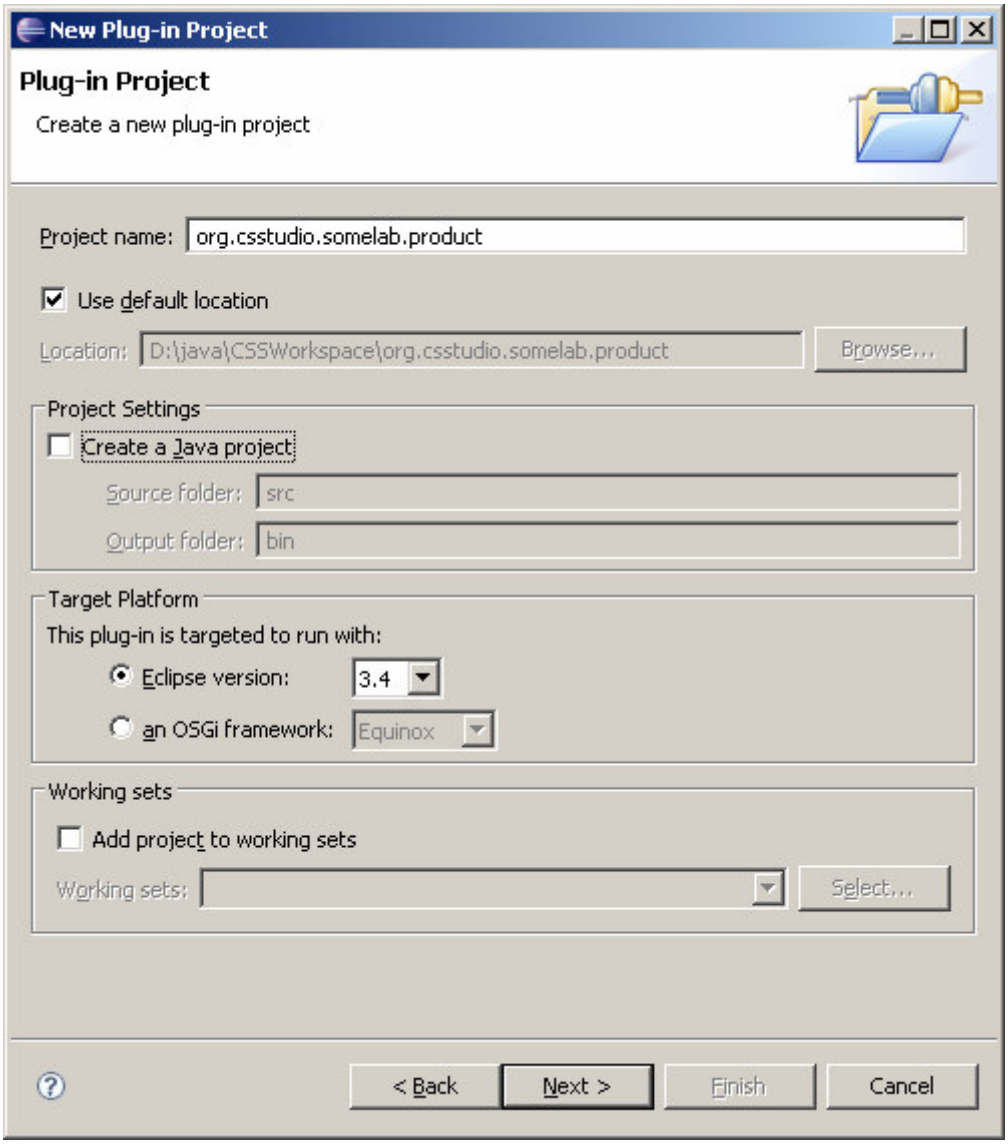
2.4. Step By Step Example

Now that we know the basics of the CSS start-up plug-ins we will make a custom product plug-in step by step.

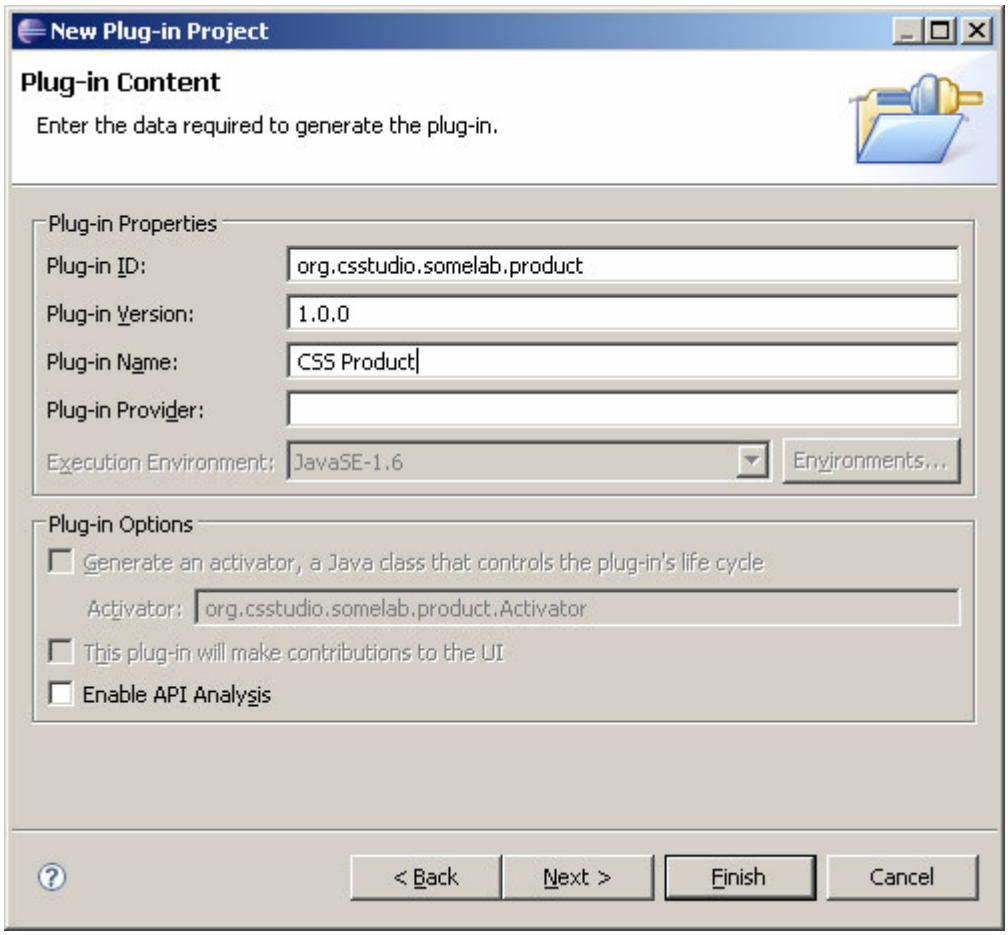
First we create a new Eclipse plug-in project using the Eclipse project creation wizard. From the *File* menu choose *New* and then choose *Project...* A wizard will pop up, which will guide you through the creation of the project.



Browse to find the *Plug-in Project* and click *Next*. A new dialog will open allowing you to specify the name of the project (and some other general properties).



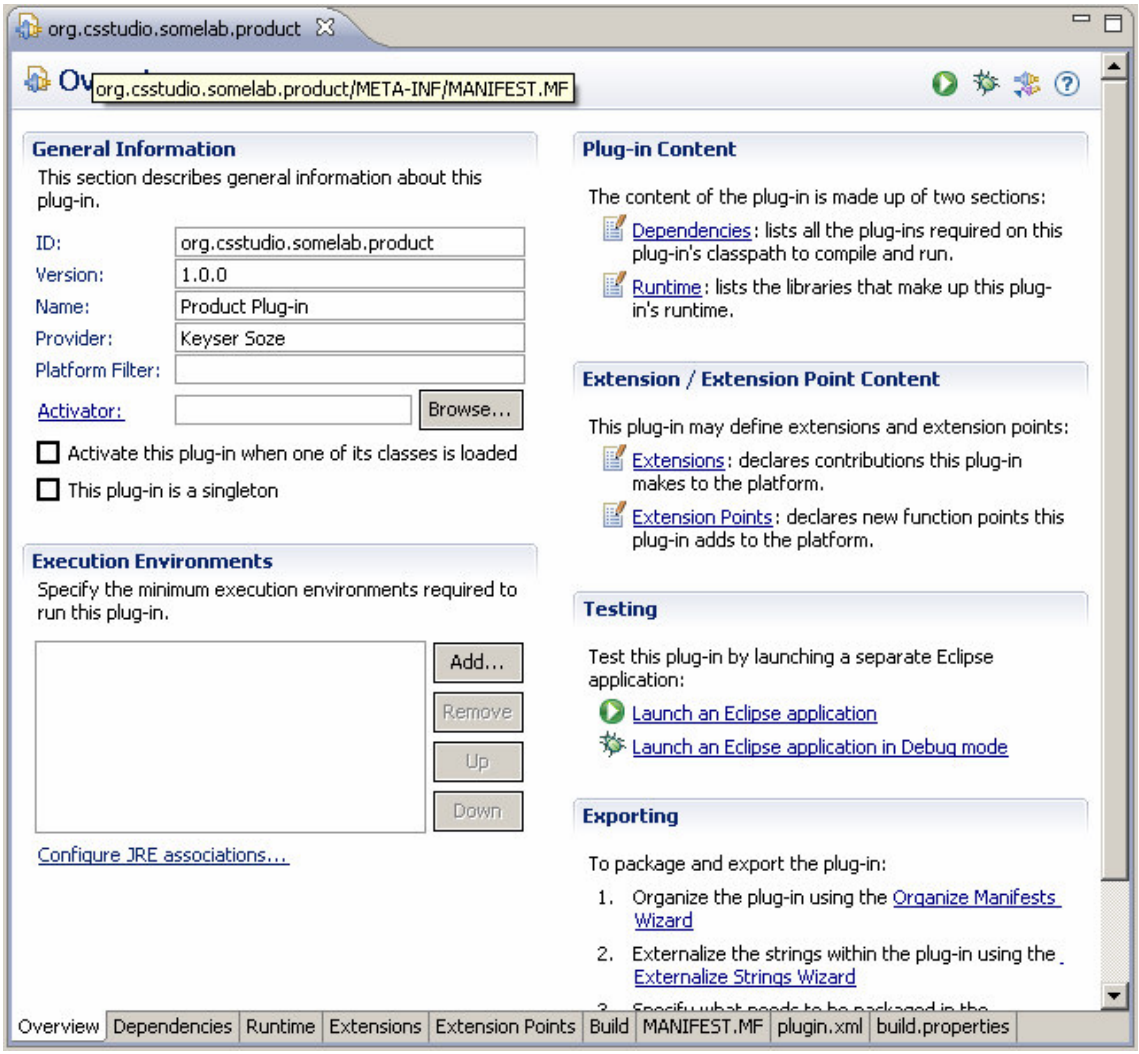
We set the appropriate project name and decide if we want to introduce some additional java code (that can also be done later, but we will have to add the newly created packages to the classpath). We choose the appropriate Eclipse version and click *Next*. The next dialog allows us the specify the most general plug-in settings, such as the ID, version etc.



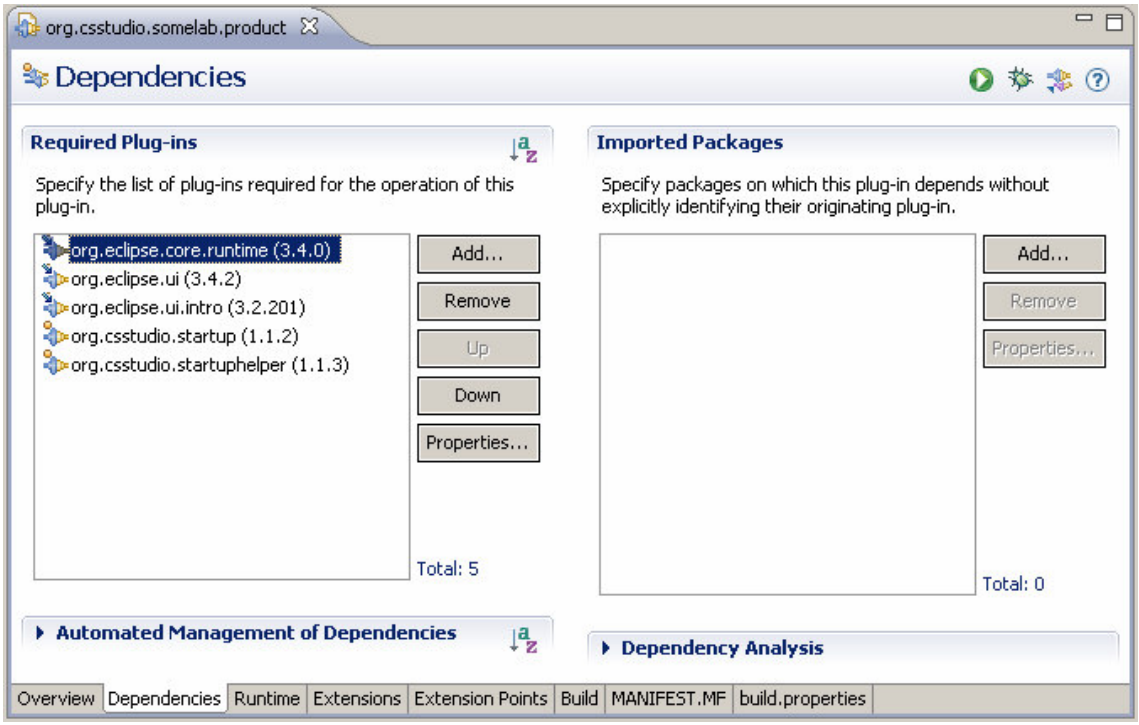
We enter the appropriate plug-in name and click *Finish*. The dialog closes and the project appears in the Eclipse's Package Explorer.

Now we have to open the project's manifest file using the plug-in manifest editor provided by Eclipse. The `MANIFEST.MF` file opens in the main Eclipse window allowing us to specify the plug-ins features, dependencies etc.

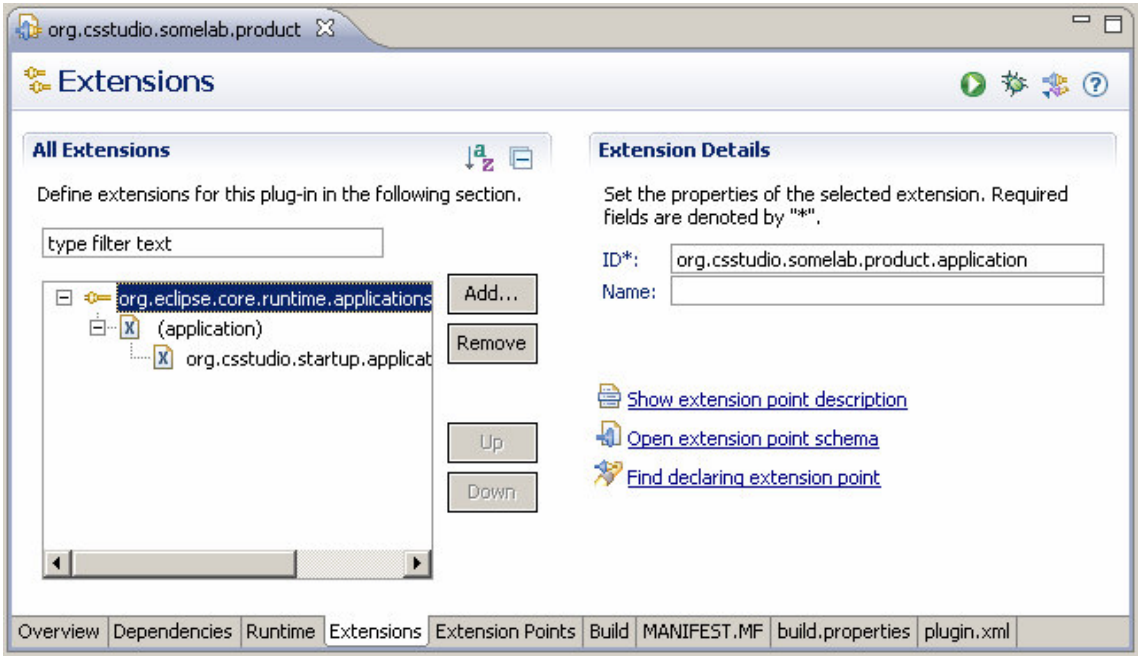
On the Overview page we define the name, provider and other general properties of the plug-in, which were not done at the creation of the plug-in.



After we define the plug-in properties we have to add the plug-in dependencies. We will use all four dependencies mentioned in the above sections: *org.eclipse.core.runtime, org.eclipse.ui, org.eclipse.ui.intro, org.csstudio.startup*, and *org.csstudio.startuphelper*. The first one will define the application entry point and the product, the second and third one will set up the welcome page and the last two are the base startup plug-ins, which will provide the startup extension and default implementation of the startup extension points.

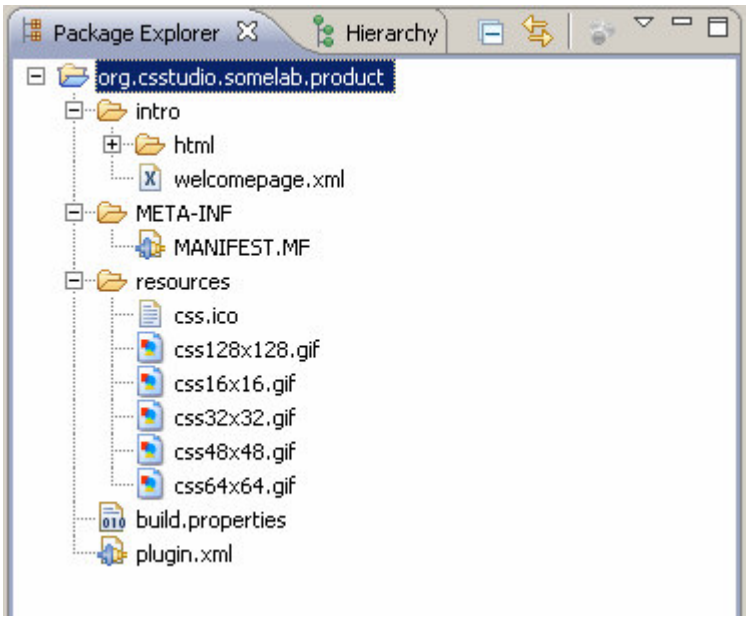


After we have set the dependencies we can start configuring the extensions. First we need to setup up the application extension. In the extension tab we add the extension point *org.eclipse.core.runtime.applications*. We set the ID of the extension and add the run node (use popup menu on the `(application)` node in the tree), which should point to the `IApplication` implementation.

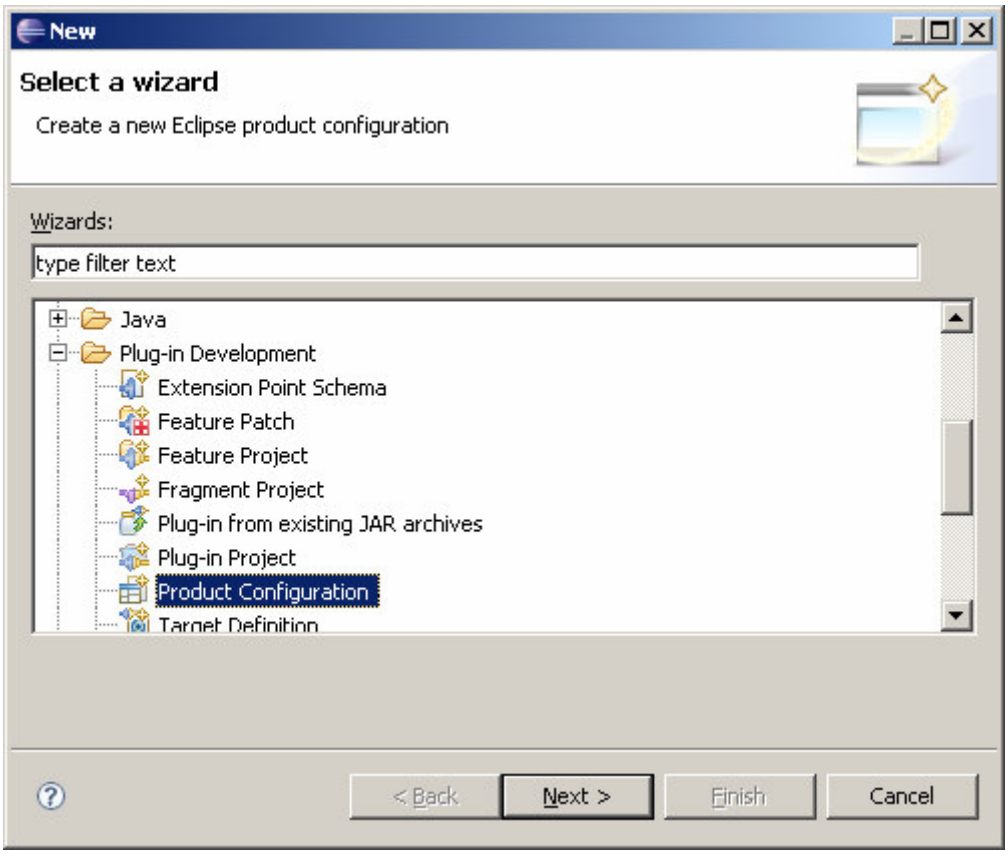


After that we have to configure the product and intro extensions. Because these extensions require some icon resources as well as the welcome page we also have to import the icons and the intro page for this product. The icons were placed into the `<root>/resource` folder, while the intro

page is placed into `the <root>/intro` of the project. The view in the package explorer should look similar to the picture below:

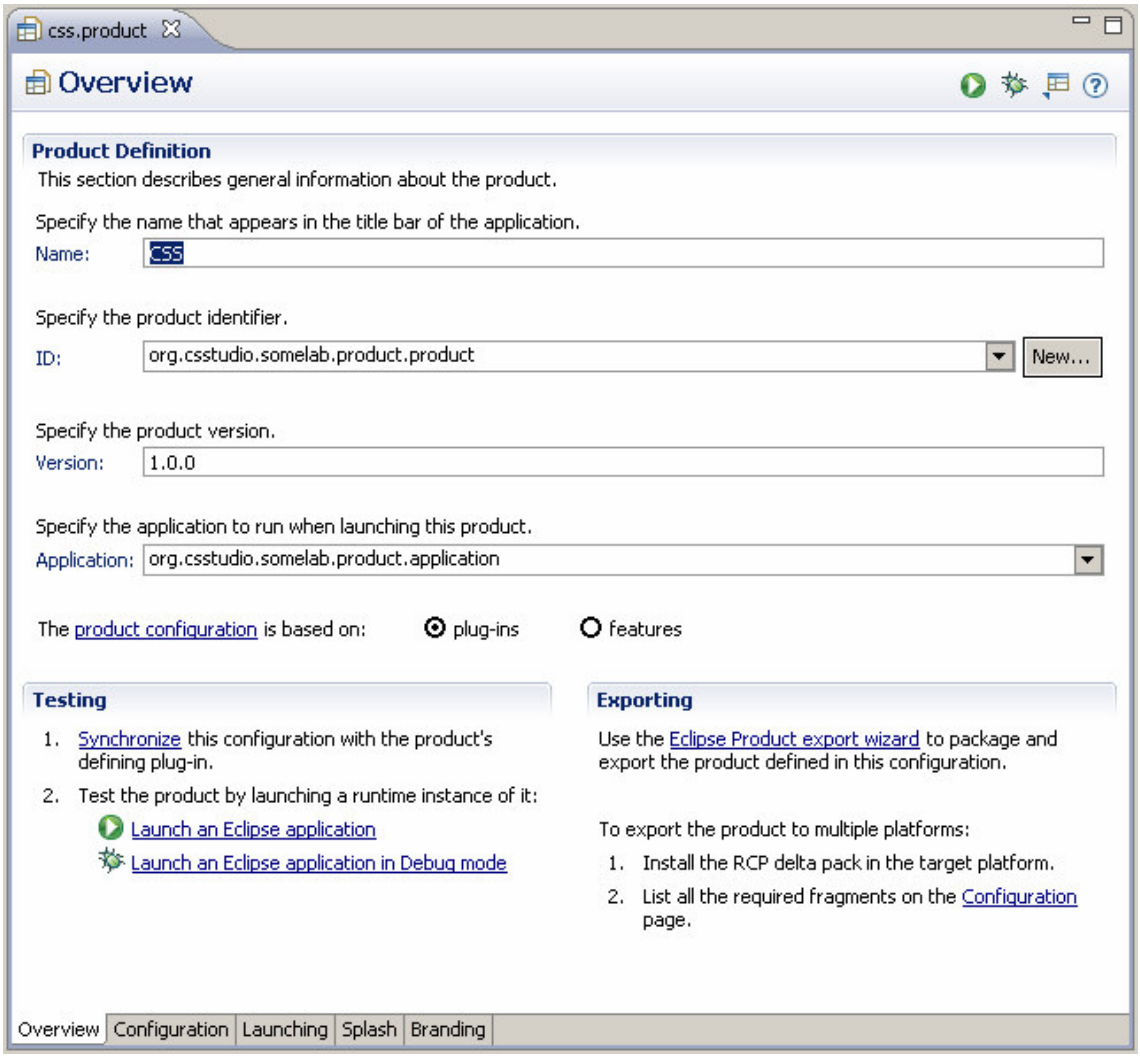


Now we have to specify the product configuration file. From the *File* menu, choose *New*, then *Other*, open *Plug-in Development* and select *Product Configuration.*

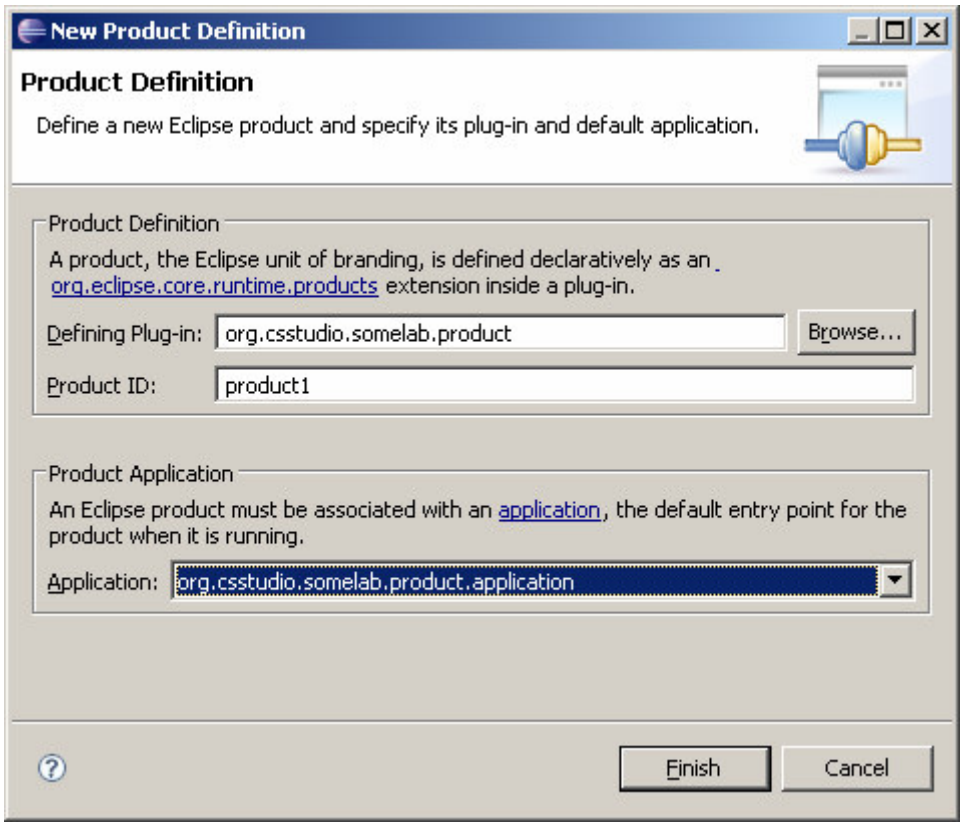


Click *Next* and set the name of the file and its location. It should be created in the root of your newly created plug-in.

We open the newly created product file in the product configuration editor and set up the name of the application.

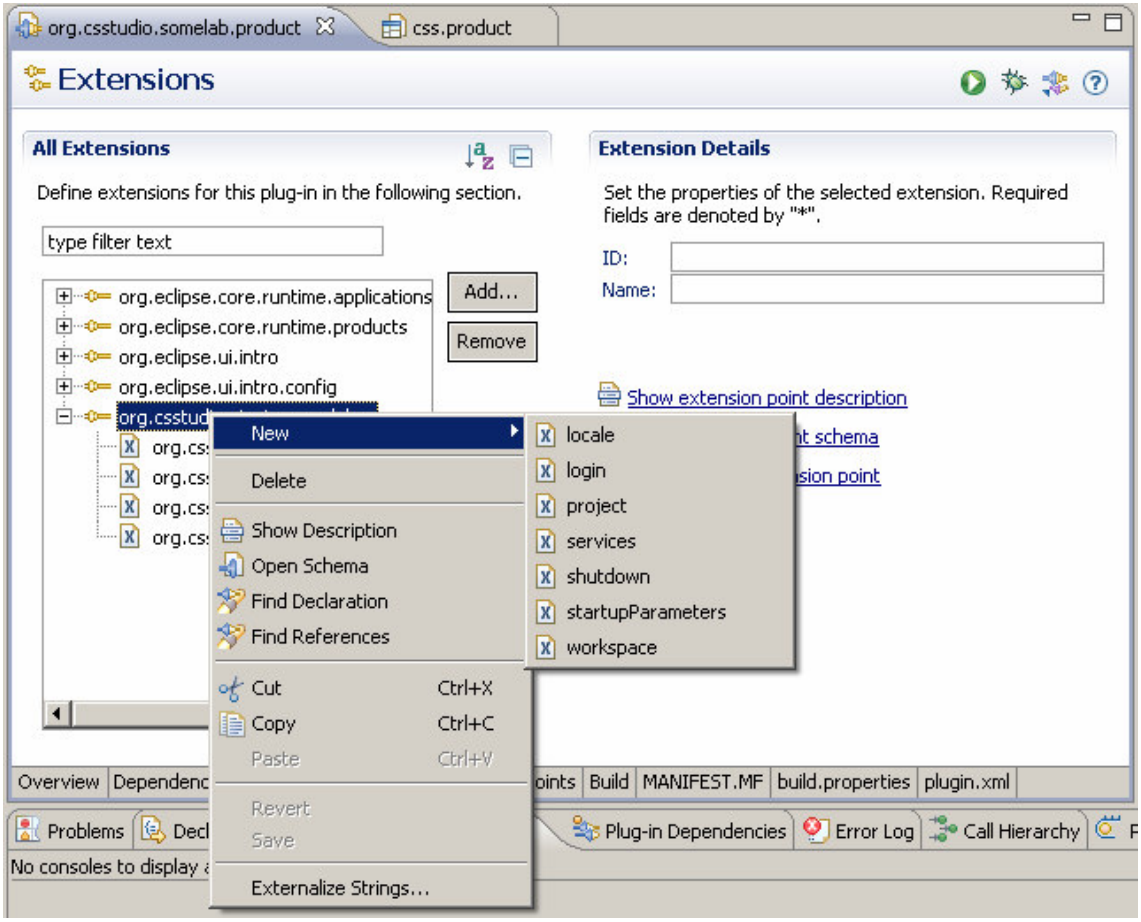


After that we have to specify a new product ID and link it to the application id which we defined in the *org.eclipse.core.runtim.applications* extension:



This will add an additional extension to our plugin.xml. Now we can start configuring the branding, splash, and launching properties of the product. Select the appropriate tab at the bottom of the project configuration editor and define those properties that you wish to use.

After we are finished with product configuring there is only one more extension point that we need to 'implement'. This is the extension point provided by the base plug-in *org.csstudio.startup.modules*. We add the extension and add the options that we want to use. In our case these are locale, login, services, and workspace:



For each of these extensions we define the implementation of the interface that is coupled with that particular extension. After the extension is added, click on the node in the tree and on the right side specify the appropriate class. If you do not require any extra functionality, you can use the default implementations, which are located in the base plug-in.

When all the files are saved you are prepared to run the plug-in as an eclipse product. You can run it from eclipse by choosing *Run*, *Run As, Eclipse Application* or you can make a plug-in binary release and use it with you stand alone deployment.

3. DOCUMENT PROPERTIES

3.1. REFERENCES

[1] K. Kasemir, Common CSS Product Code, 22-10-2008

[2] J. Bobnar, CSS Common Startup Code: Study and Proposition, 23-02-2009

[3] http://www.eclipse.org

[4] http://www.vogella.de/articles/EclipseExtensionPoint/article.html