

CSS Common Startup Code

Instructions How to Write Product Plug-ins

| | |
|--------------------|-------------------------------------|
| Revision: | 1.1 |
| 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: | April 28, 2009 |
| Created: | April 10, 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 |

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 | 4 |
| 2. Creating a Custom Start-up Plug-in | 5 |
| 2.1. Base Plug-in Description | 5 |
| 2.2. Extendable Features of Base Plug-in | 6 |
| 2.3. Product Plug-in..... | 7 |
| 2.3.1. Required Setup | 7 |
| 2.3.2. Optional Setup | 8 |
| 2.4. Step By Step Example..... | 9 |
| 3. Document Properties | 17 |
| 3.1. References | 17 |

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 can be used for making this customized plug-in.

2. CREATING A CUSTOM START-UP PLUG-IN

The start-up plug-in architecture consist of two (or more) CSS specific plug-ins. The first one is the base start-up plug-in which is named *org.csstudio.startuphelper* and provides the common code and resources to be used during application startup and tear down. The second plug-in is the 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*.

2.1. BASE PLUG-IN DESCRIPTION

The base plug-in for all the CSS products is the plug-in *org.csstudio.startuphelper*. 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* implementation provided within 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_customization.ini*: the customization file which defines a set of properties used by different plug-ins
- *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.)
- *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.
 - *splash.bmp*: the default splash screen for the product plug-in
- *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 four packages
 - *org.csstudio.startuphelper*: the plug-in activator which takes care of the plug-in’s lifecycle
 - *org.csstudio.startuphelper.application*: default implementation of the *IApplication* interface

- `org.csstudio.startuphelper.extensions`: the interfaces which are linked with the `org.csstudio.startuphelper.modules` extension point
- `org.csstudio.startuphelper.extensions.impl`: the default implementations of the extension point interfaces from the parent package

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.

2.2. EXTENDABLE FEATURES OF BASE PLUG-IN

The startup plug-in provides three extendable entry points.

1. The first one is the implementation of `org.eclipse.ui.plugin.AbstractUIPlugin`, which can be found in the package `org.csstudio.startuphelper`, class `Activator`. This class controls the plug-in's lifecycle. This activator is used by the startup plug-in and provides a common logger to be used during the initialization of the plug-in. If the user wants to add some specific activation for the product plug-in this class can be extended to provide the additional features. However, it is not required by the eclipse framework that the plug-in declares its activator and can therefore omit the setting of the activator if it is not needed.
2. The second entry point 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, where each of them provides a certain feature which can be loaded by the application. This class also defines the sequence how the features are loaded. For instance, firstly the startup parameters are read, which is followed by applying the locale settings etc. 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).
3. The base plug-in defines an extension point `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 log 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. However, if one decides that he wants to customize his application, there exist default implementations of some of the above extensions. They can be found in the package *org.csstudio.startuphelper.extensions.impl*. They provide some most commonly used implementations, 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 plug-in (for details and step-by-step on 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-n project creator wizard, 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.startuphelper* 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 *org.eclipse.ui.intro* also have to be provided in the dependencies tab. User is free to use other dependencies 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.

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.startuphelper.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 implementation of the extension point interfaces or he can use one of 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 reference see the extension points documentation).

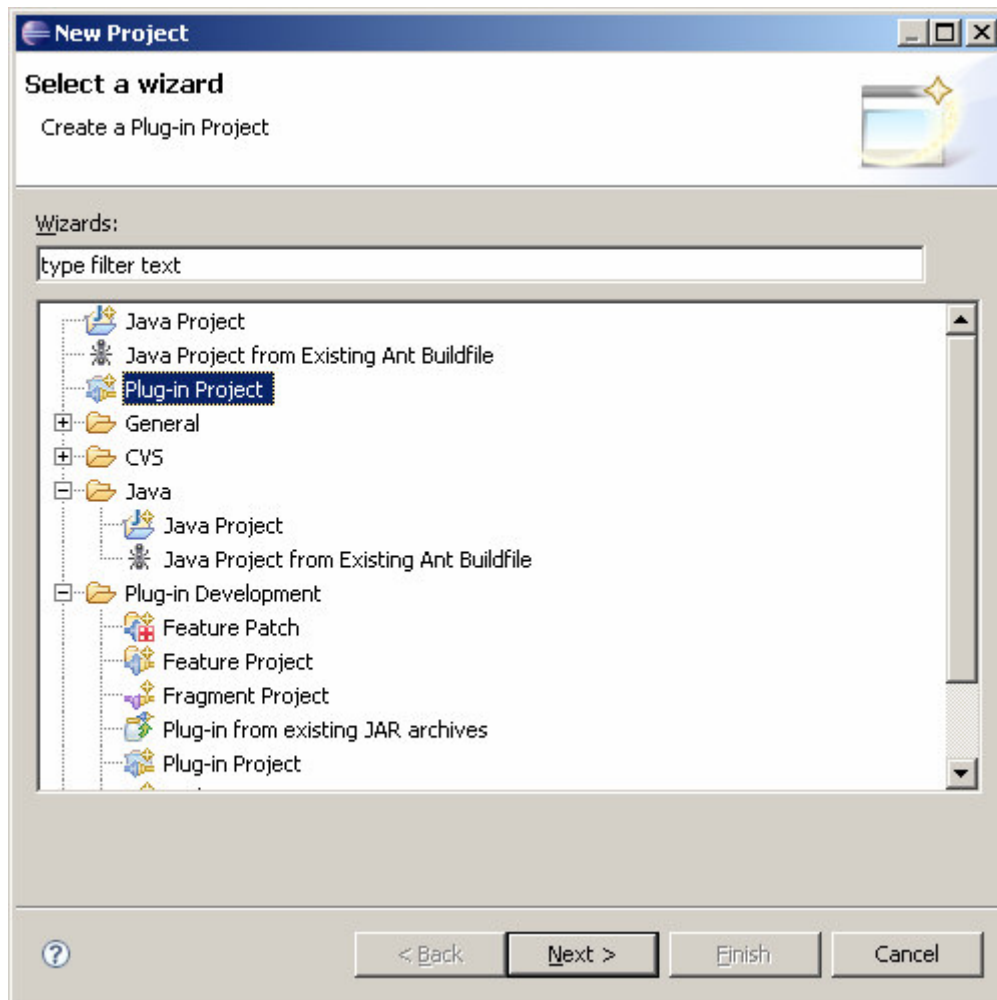
2.3.2.3. Branding

If application is required to have more professional look 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). 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 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). 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. First we create a new Eclipse plug-in project. 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.

New Plug-in Project

Plug-in Project
Create a new plug-in project

Project name:

☒ Use default location

Location:

Project Settings

☐ Create a Java project

Source folder:

Output folder:

Target Platform

This plug-in is targeted to run with:

☒ Eclipse version:

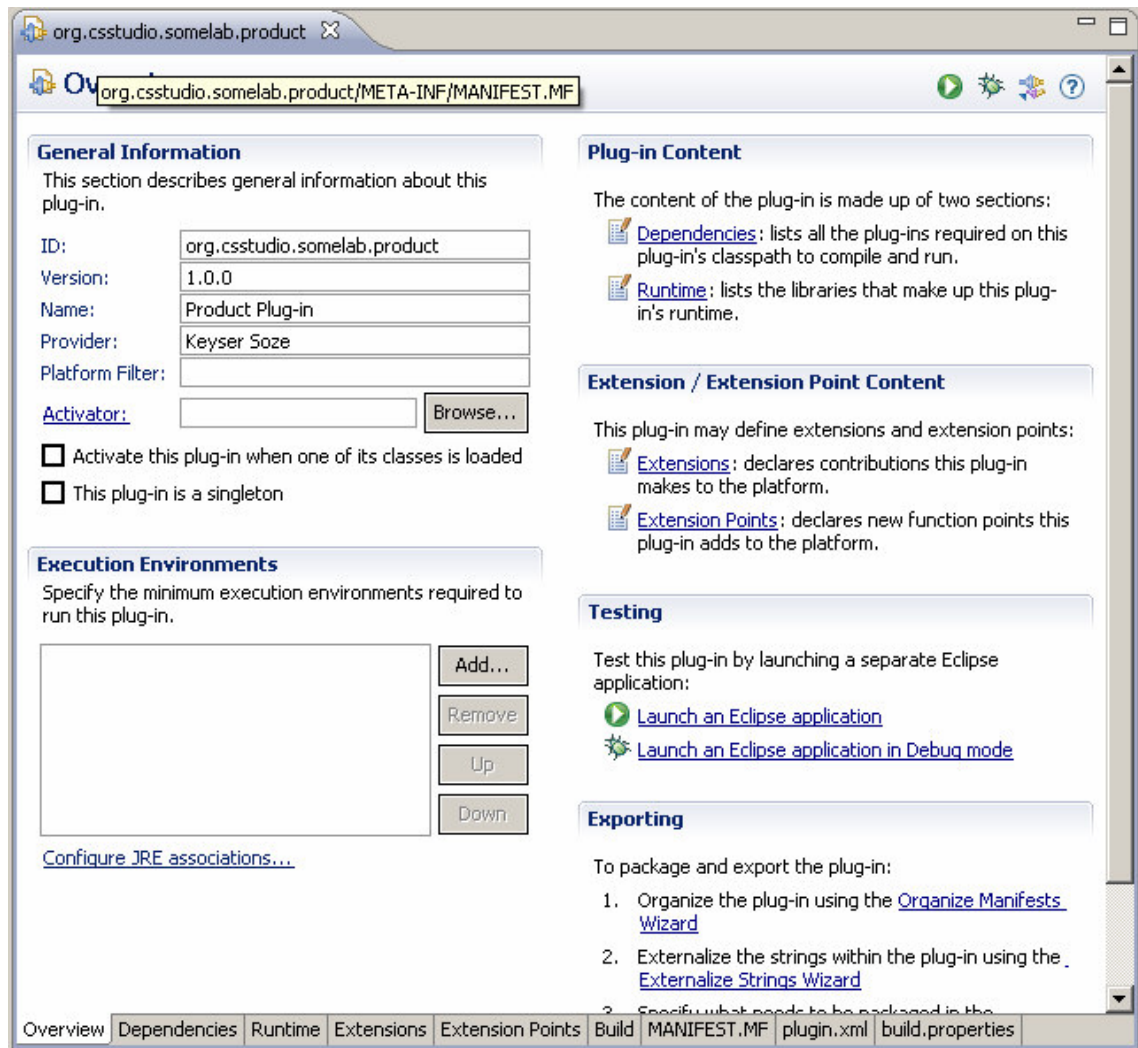
☐ an OSGi framework:

Working sets

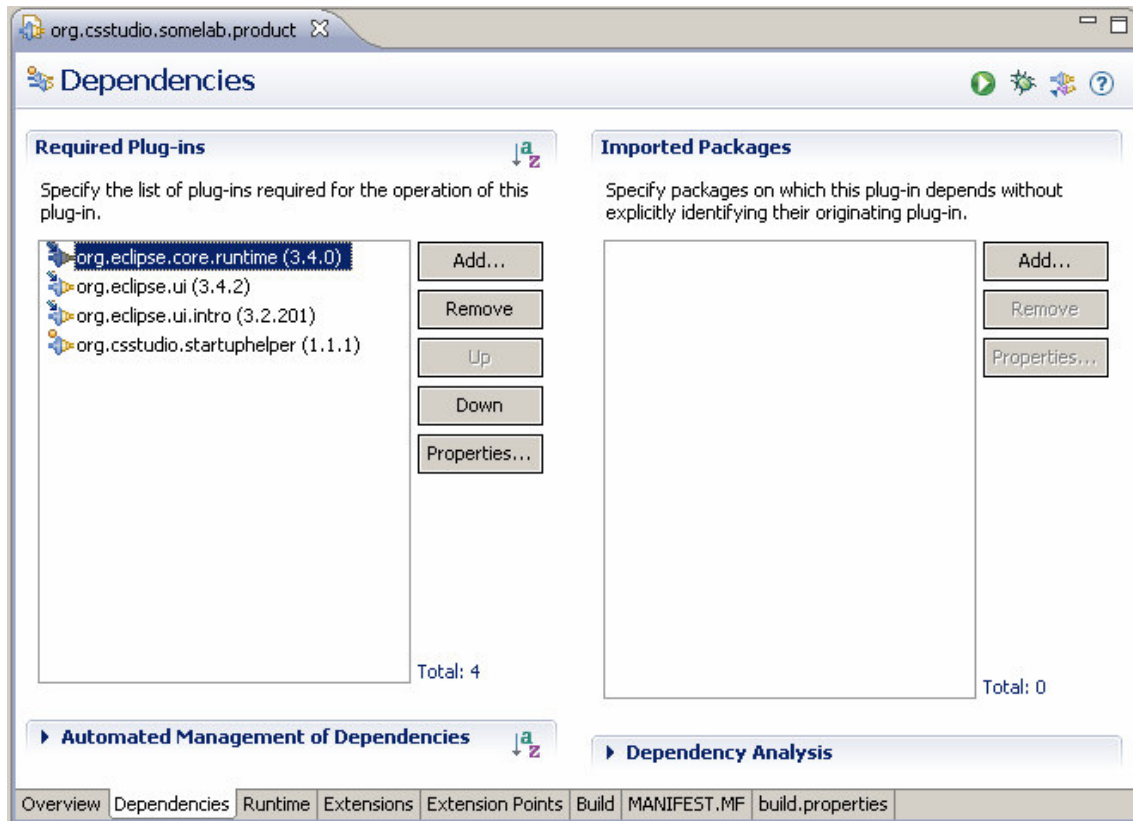
☐ Add project to working sets

Working sets:

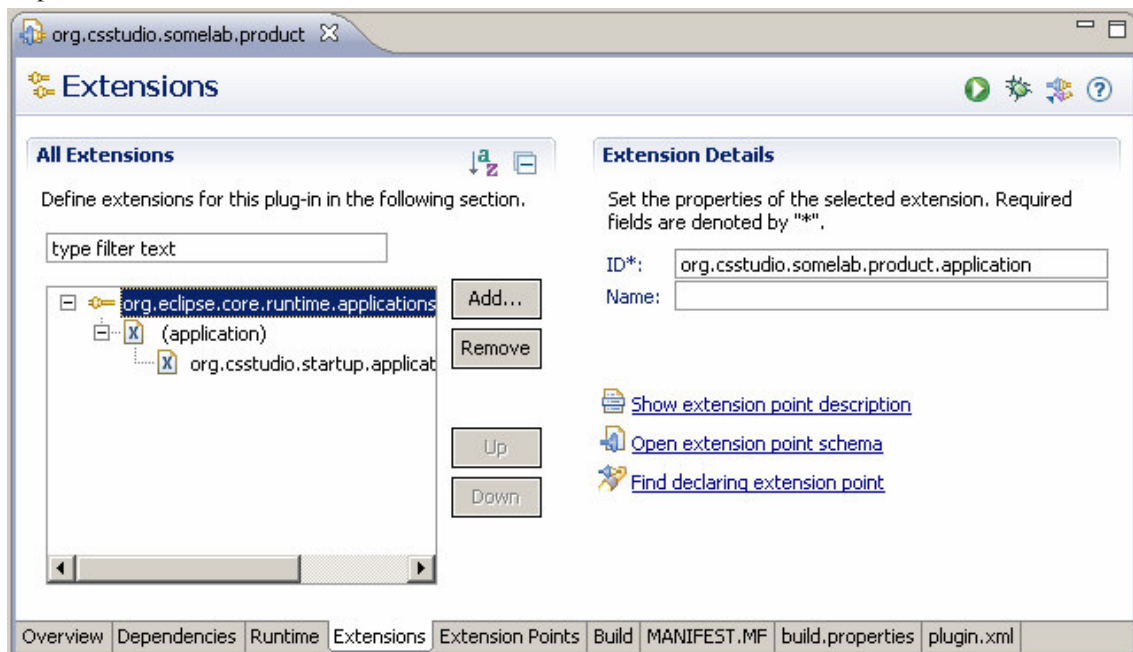
We set the appropriate project name and decide if we want to introduce some additional java code (that can also be done later, but we will have to add the newly created packages to the classpath). After the project is created we open the project's manifest file using the plug-in manifest editor provided by Eclipse. On the Overview page we define the name, provider and other general properties 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*, 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 one is the base startup plug-in.

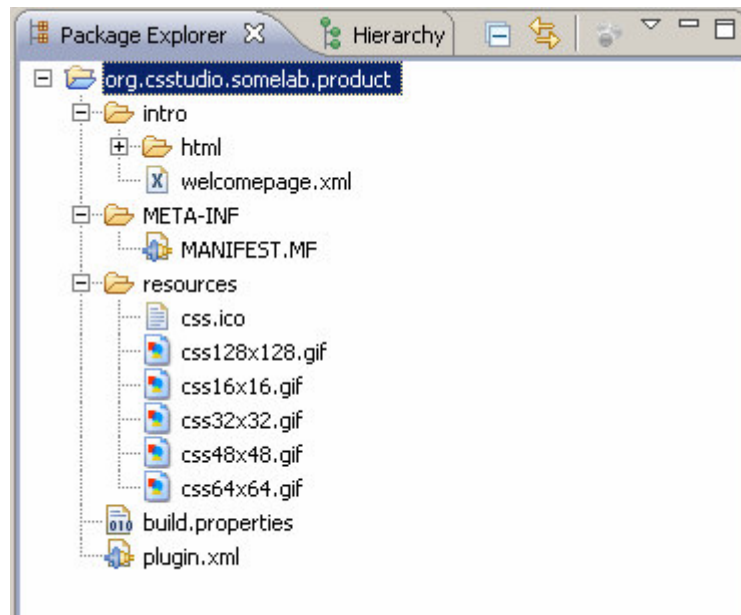


After this we can start configuring the extensions. First we need to setup up the application extension. In the extension tab we add the extension point *org.eclipse.core.runtime.applications*. We set the ID of the extension and add the run node, which should point to the *IApplication* implementation.

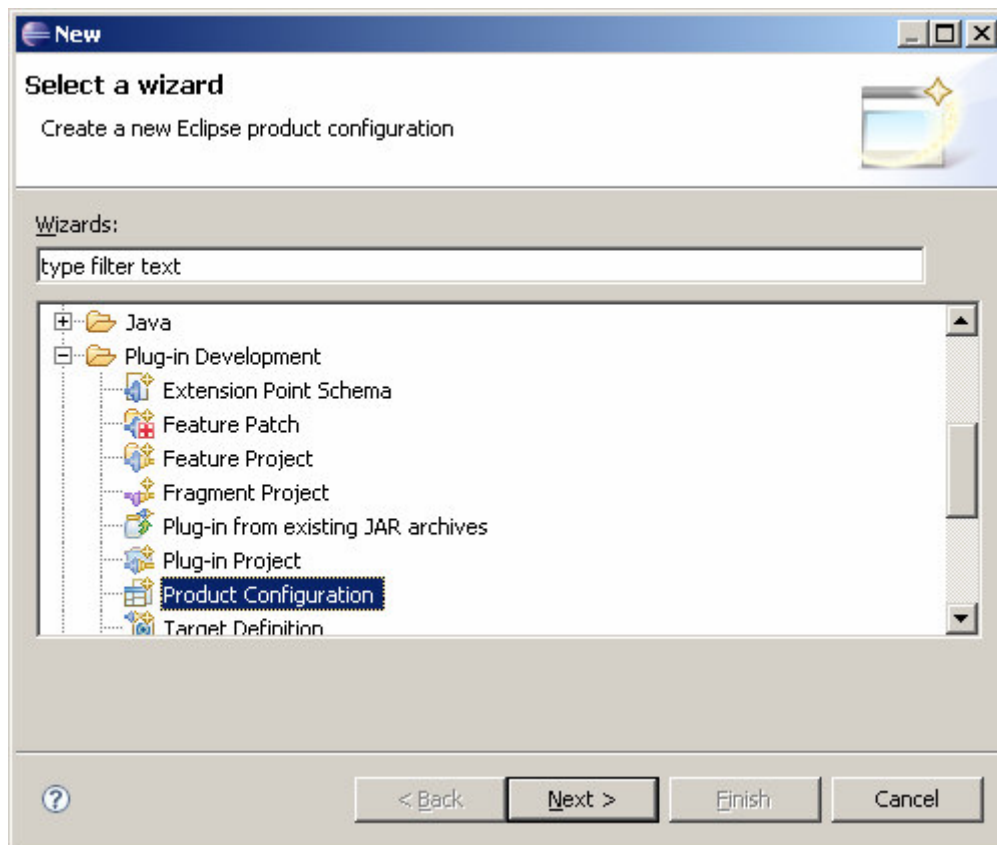


After that we have to configure the product and intro extension. Because these extensions require some icon resources as well as the welcome page we have also imported the icons for this product and the intro page. The icons were placed them into the <root>/resource folder, while the intro

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, New, Other we select Plug-in Development and Product Configuration



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

The screenshot shows the 'Overview' tab of the Eclipse Product Configuration Wizard. The window title is 'css.product'. The 'Product Definition' section contains the following fields: 'Name' with the value 'css', 'ID' with the value 'org.csstudio.somelab.product.product' and a 'New...' button, 'Version' with the value '1.0.0', and 'Application' with the value 'org.csstudio.somelab.product.application'. Below these fields, there are two radio buttons: 'plug-ins' (selected) and 'features'. The 'Testing' section on the left lists two steps: 'Synchronize this configuration with the product's defining plug-in.' and 'Test the product by launching a runtime instance of it:'. Under the second step, there are two links: 'Launch an Eclipse application' and 'Launch an Eclipse application in Debug mode'. The 'Exporting' section on the right explains that the 'Eclipse Product export wizard' should be used to package and export the product. It also provides instructions for exporting to multiple platforms: '1. Install the RCP delta pack in the target platform.' and '2. List all the required fragments on the Configuration page.' At the bottom, there is a tabbed interface with 'Overview' selected, and other tabs for 'Configuration', 'Launching', 'Splash', and 'Branding'.

css.product

Overview

Product Definition

This section describes general information about the product.

Specify the name that appears in the title bar of the application.

Name:

Specify the product identifier.

ID:

Specify the product version.



Version:

Specify the application to run when launching this product.

Application:

The [product configuration](#) is based on: ☒ plug-ins ☐ features

Testing

1. [Synchronize](#) this configuration with the product's defining plug-in.
2. Test the product by launching a runtime instance of it:
 -  [Launch an Eclipse application](#)
 -  [Launch an Eclipse application in Debug mode](#)

Exporting

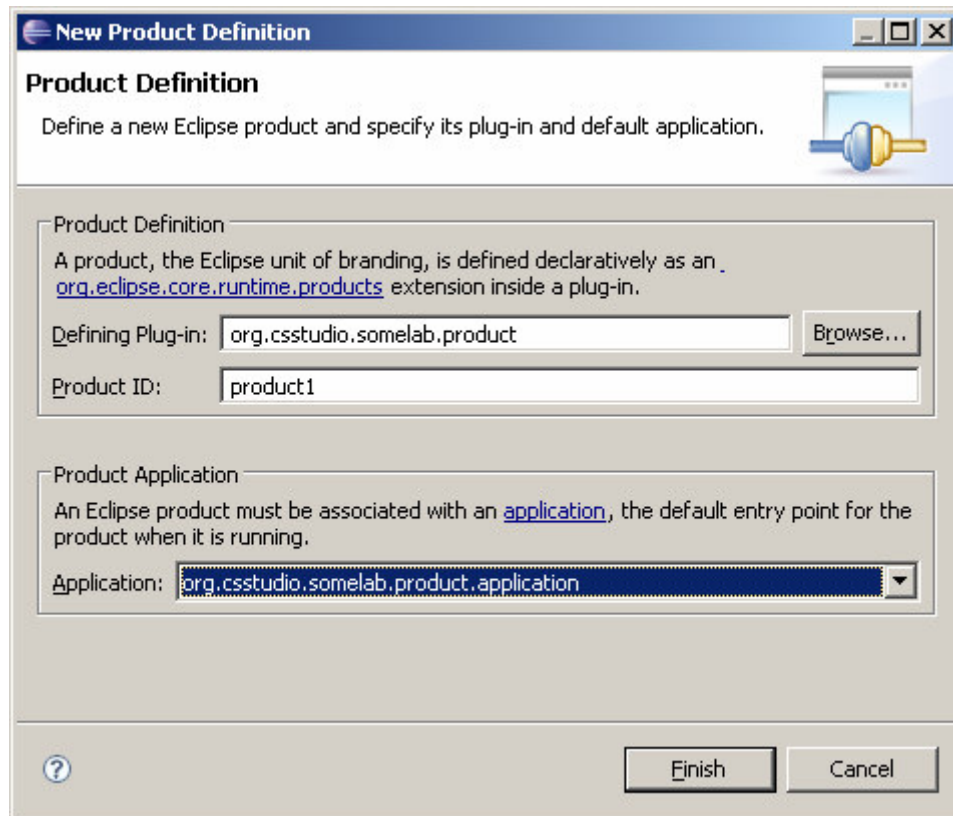
Use the [Eclipse Product export wizard](#) to package and export the product defined in this configuration.

To export the product to multiple platforms:

1. Install the RCP delta pack in the target platform.
2. List all the required fragments on the [Configuration](#) page.

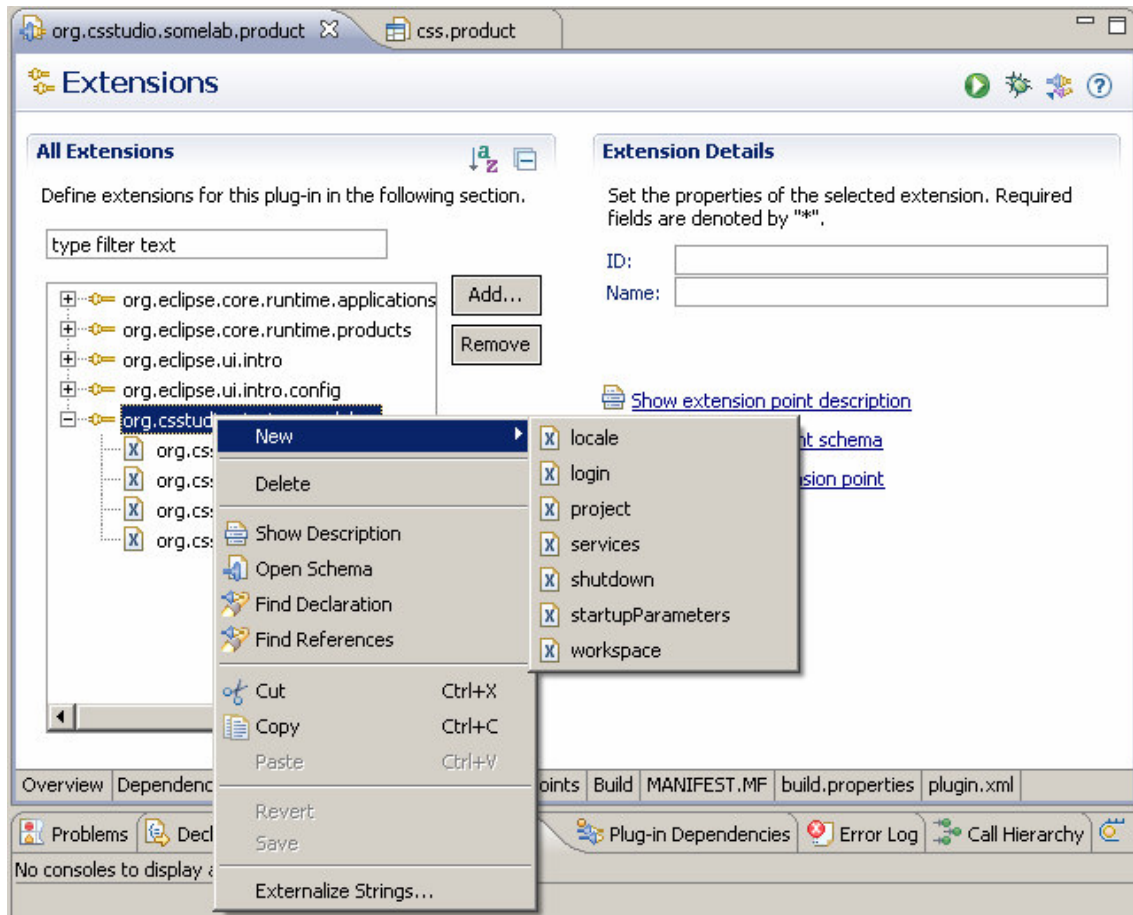
Overview Configuration Launching Splash Branding

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.runtime.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 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.startuphelper.modules*. We add the extension and add the option 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. If we do not require any extra functionality, we can set the default implementations, which are located in the base plug-in.

We are now prepared to run the plug-in as an eclipse product.

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>