Create a New Java based KNIME Extension

Table of Contents

Introduction

This quickstart guide describes how to create a new KNIME Extension in Java, i.e. write a new node implementation to be used in KNIME Analytics Platform. You will learn how to set up a KNIME SDK, how to create a new KNIME Extension project, how to implement a simple manipulation node, how to test the node, and how to easily deploy the node in order to make it available for others. Publishing the extension is explained in detail in the following guide: Publish Your Extension on KNIME Community Hub

For this purpose, we created a reference extension you can use as orientation. This KNIME Extension project can be found in the `org.knime.examples.numberformatter` folder of the `knime-examples` GitHub repository. It contains all required project and configuration files and an implementation of a simple *Number Formatter* example node, which performs number formatting of numeric values of the input table. We will use this example implementation to guide you through all necessary steps that are involved in the creation of a new KNIME Extension.

Set up a KNIME SDK

In order to start developing KNIME source code, you need to set up a KNIME SDK. A KNIME SDK is a configured Eclipse installation which contains KNIME Analytics Platform dependencies. This is necessary as Eclipse is the underlying base of KNIME Analytics Platform i.e. KNIME Analytics Platform is a set of plug-ins that are put on top of Eclipse and the Eclipse infrastructure. Furthermore, Eclipse is an IDE, which you will use to write the actual source code of your new node implementation.

To set up your KNIME SDK, we start with an "Eclipse IDE for RCP and RAP Developers" installation (this version of Eclipse provides tools for plug-in development) and add all KNIME Analytics Platform dependencies. In order to do that, please follow the SDK Setup instructions. Apart from giving instructions on how to set up a KNIME SDK, the SDK Setup will give some background about the Eclipse infrastructure, its plug-in mechanism, and further useful topics like how to explore KNIME source code.

Create a New KNIME Extension Project

After Eclipse is set up and configured, create a new KNIME Extension project. A KNIME Extension project is an Eclipse plug-in project and contains the implementation of one or more nodes and some KNIME Analytics Platform specific configuration. The easiest way to create a KNIME Extension project, is by using the KNIME Node Wizard, which will automatically generate the project structure, the plug in manifest and all required Java classes. Furthermore, the wizard will take care of embedding the generated files in the KNIME framework.

The KNIME Node Wizard

1. **Install the KNIME Node Wizard**

   Open the Eclipse installation wizard at `Help → Install New Software…`, enter the following update site location: `https://update.knime.com/analytics-platform/5.1/` in the location box labelled `Work with:`.

   Hit the Enter key, and put `KNIME Node Wizard` in the search box. Tick the `KNIME Node Wizard` under the category `KNIME Node Development Tools`, click the **Next** button and follow the instructions. Finally, restart Eclipse.
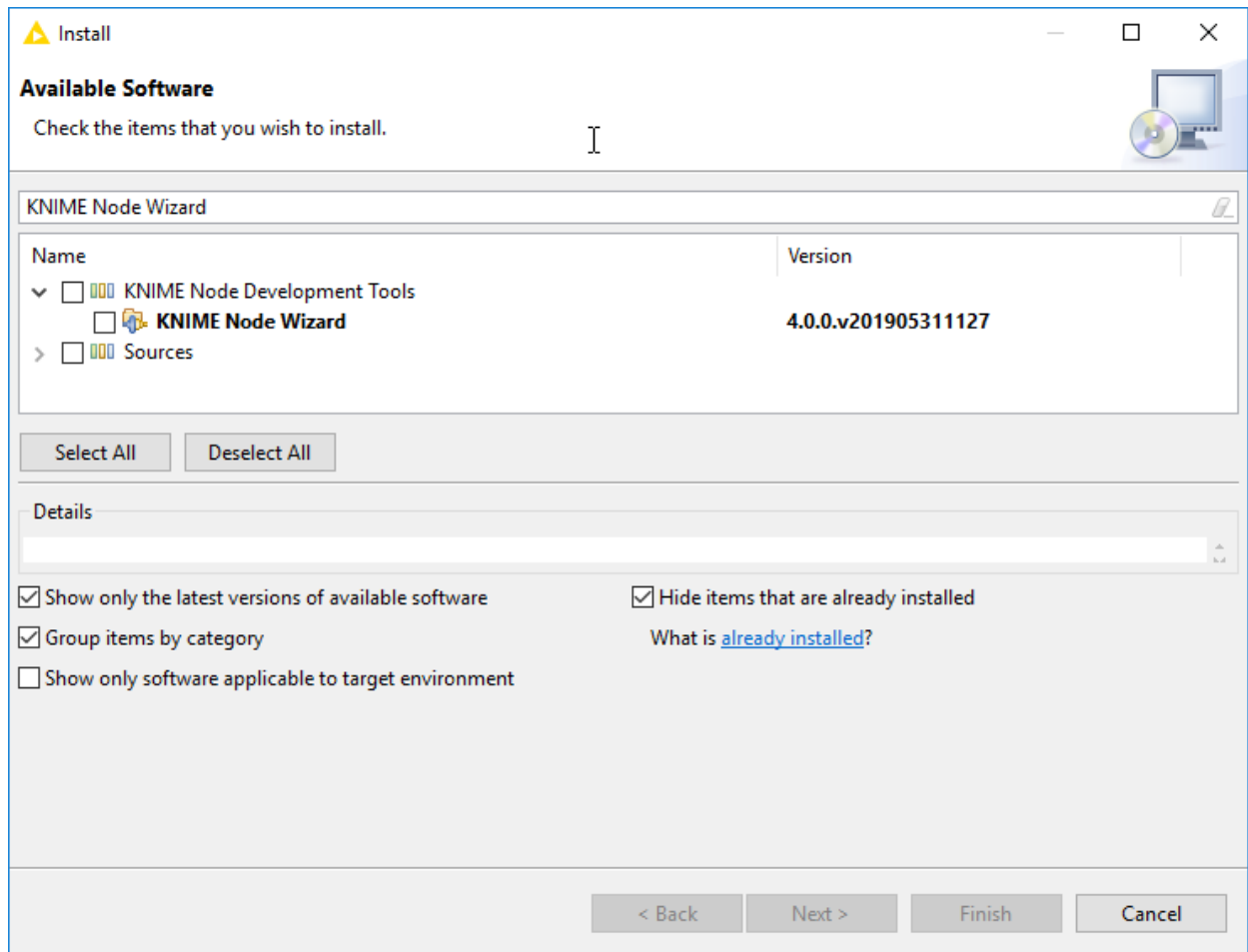
Figure 1. The KNIME Node Wizard installation dialog.

2. **Start the KNIME Node Wizard**

   After Eclipse has restarted, start the KNIME Node Wizard at `File → New → Other⋯`, select `Create a new KNIME Node-Extension` (can be found in the category `Other`), and hit the **Next** button.
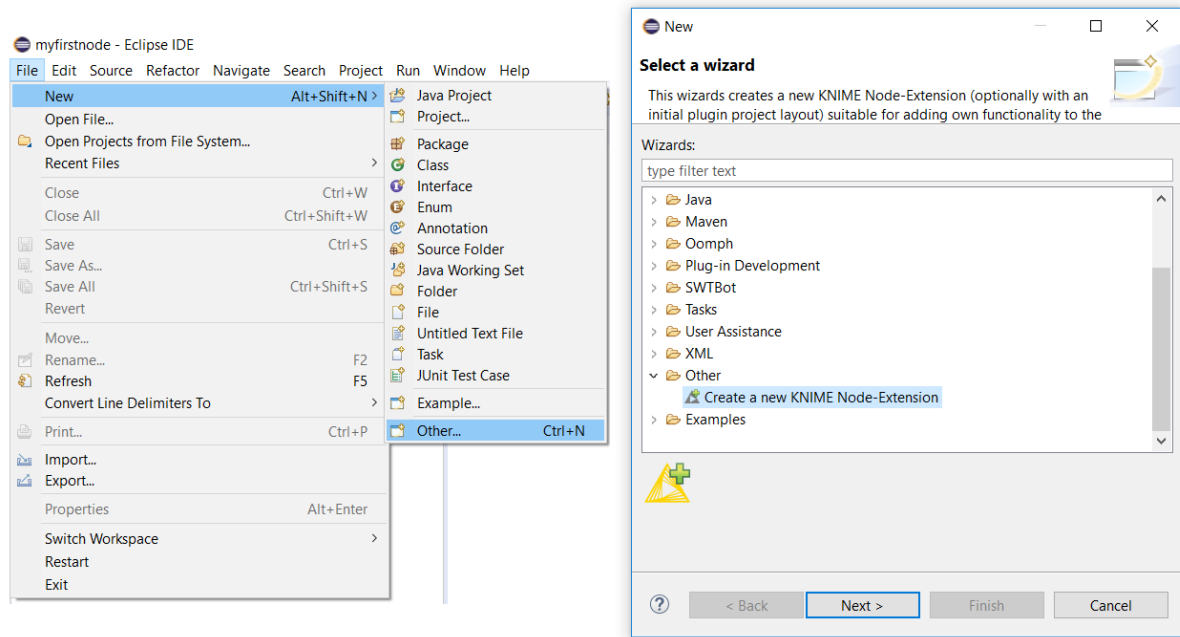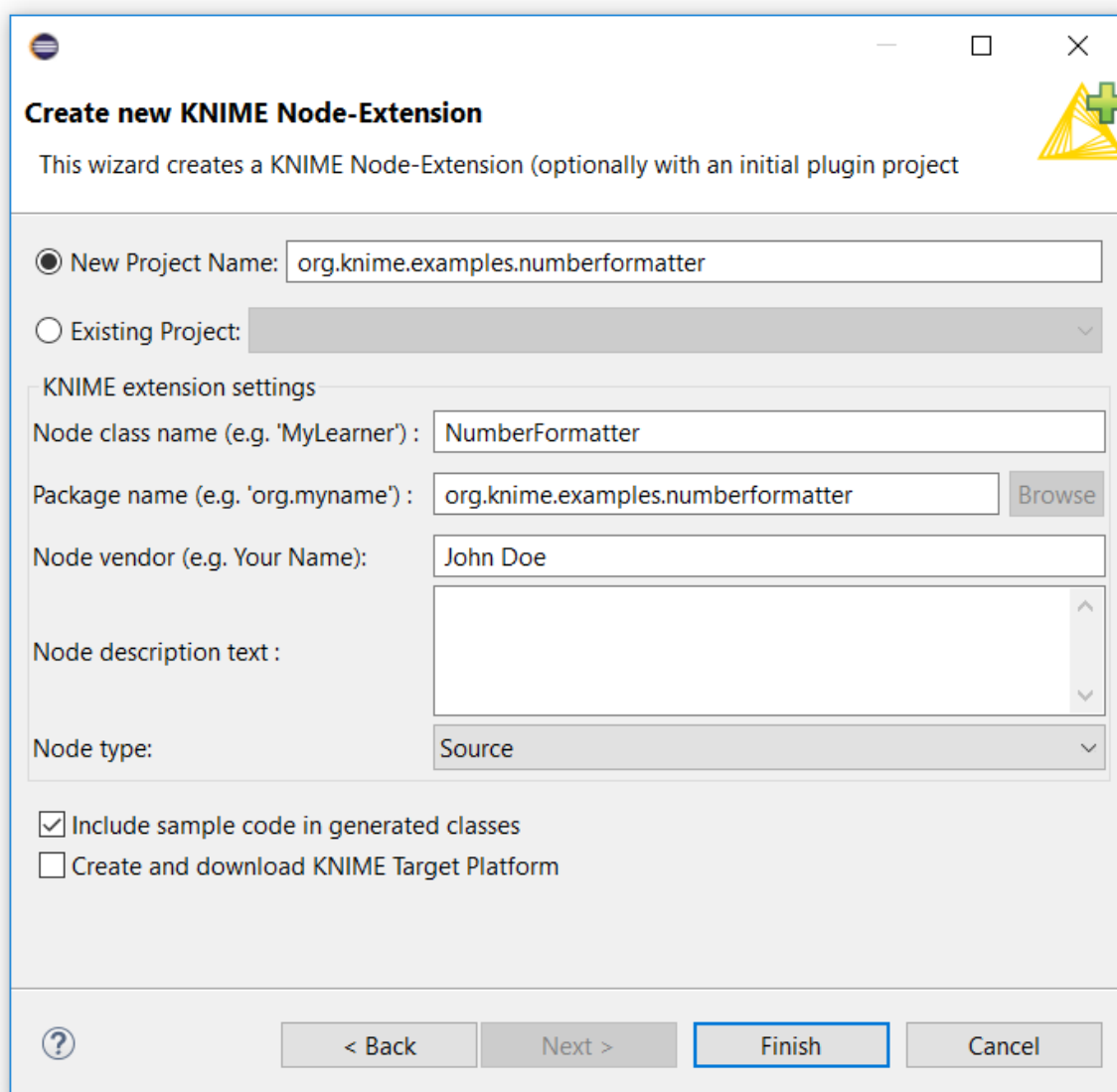
Figure 2. The KNIME Node Wizard start dialogs.

3. **Create a KNIME Extension Project**

   In the `Create new KNIME Node-Extension` dialog window enter the following values:

   - New Project Name: `org.knime.examples.numberformatter`

   - Node class name: `NumberFormatter`

   - Package name: `org.knime.examples.numberformatter`

   - Node vendor: `<your_name>`

   - Node type: Select `Manipulator` in the drop down menu.

   Replace `<your_name>` with the name that you like to be the author of the created extension. Leave all other options as is and click **Finish**.

Figure 3. The KNIME Node Wizard dialog.

After some processing, a new project will be displayed in the Package Explorer view of Eclipse with the project name you gave it in the wizard dialog.

> **i** Make sure that the checkbox `Include sample code in generated classes` is checked. This will include the code of the aforementioned *Number Formatter* node in the generated files.
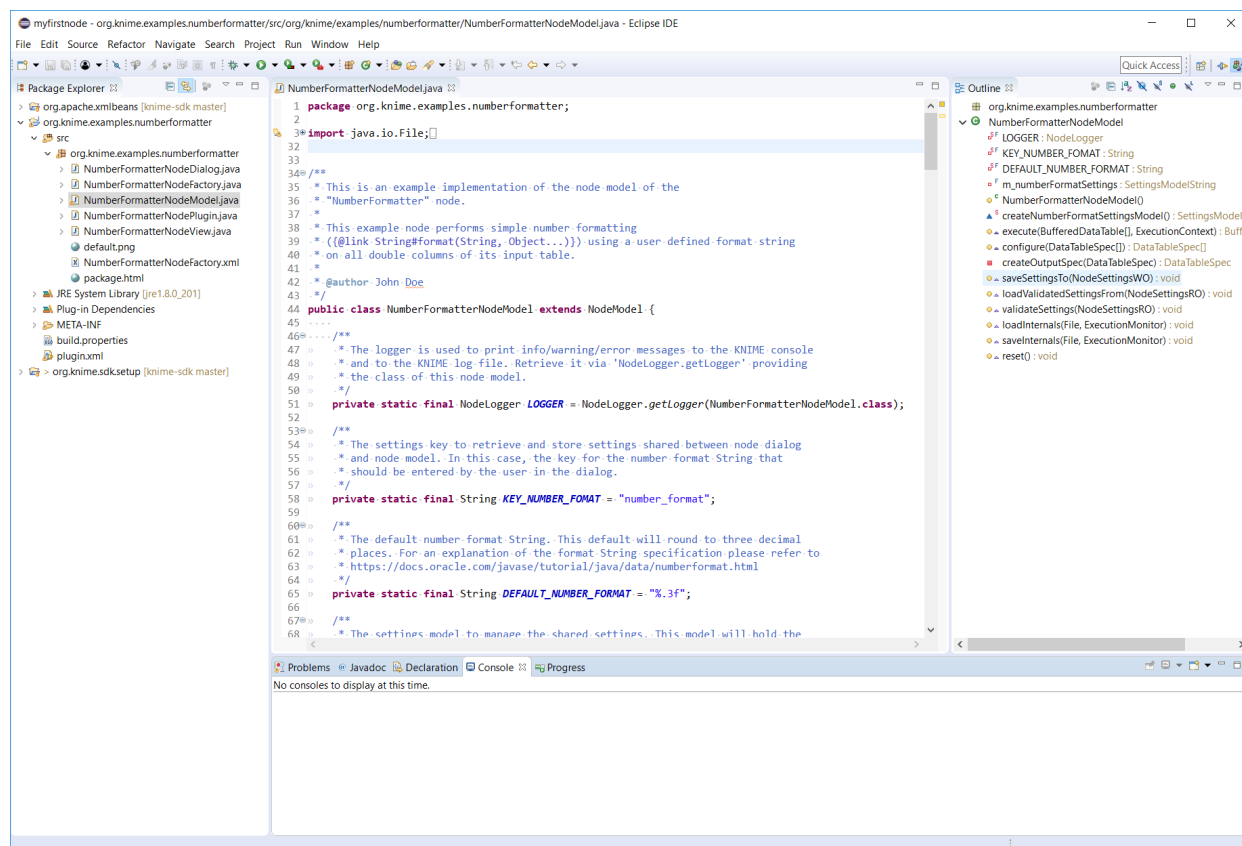
Figure 4. A view of Eclipse after the KNIME Node Wizard has run.

In the `Package Explorer` view of Eclipse (left side) you should now see three projects. The two projects `org.apache.xmlbeans` and `org.knime.sdk.setup` which you imported in the SDK Setup, and the project `org.knime.examples.numberformatter` that you just created using the KNIME Node Wizard.

Test the Example Extension

At this point, all parts that are required for a new KNIME Extension are contained in your Eclipse workspace and are ready to run. To test your node, follow the instructions provided in the **Launch KNIME Analytics Platform Section** of the SDK Setup. After you started KNIME Analytics Platform from Eclipse, the *Number Formatter* node will be available at the root level of the node repository. Create a new workflow using the node (see Figure below), inspect the input and output tables, and play around with the node.
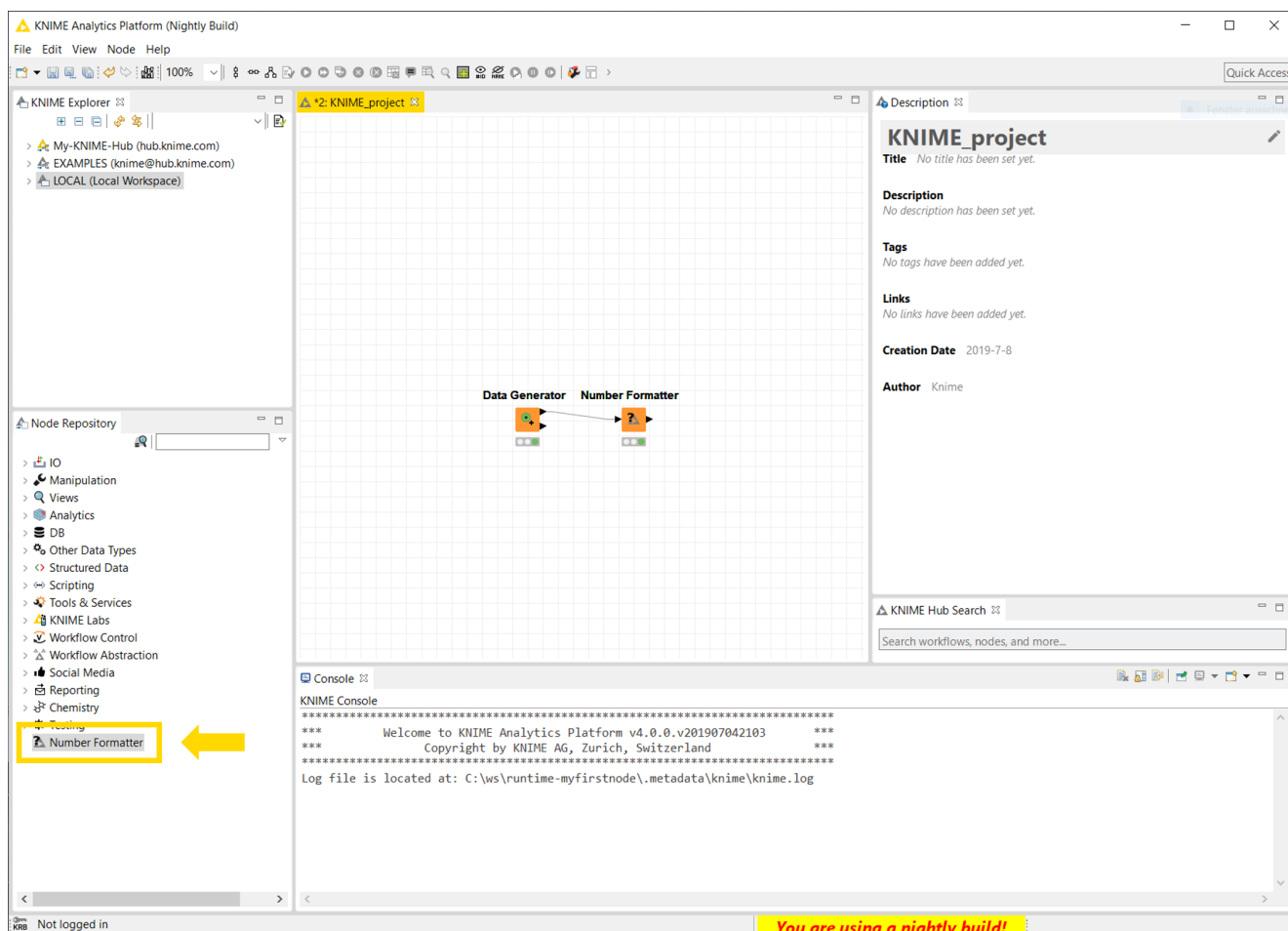

Figure 5. KNIME Analytics Platform development version started from Eclipse showing an example workflow. The Number Formatter node contained in the Eclipse workspace is displayed at the bottom of the node repository.

The node will perform simple rounding of numbers from the input table. To change the number of decimal places the node should round to, change the digit contained in the format String that can be entered in the node configuration (e.g. `%.2f` will round to two decimal places,the default value is `%.3f`). After you are done, close KNIME Analytics Platform.

Project Structure

Next, let's review the important parts of the extension project you've just created. First, we'll have a look at the files located in `org.knime.examples.numberformatter`.

The files contained in this folder correspond to the actual node implementation. There are four Java classes implementing what the node should do, how the dialog and the view looks like, one XML file that contains the node description, and an image which is used as the node icon (in this case a default icon) displayed in the workflow view of KNIME Analytics Platform. Generally, a node implementation comprises of the following classes: `NodeFactory`, `NodeModel`, `NodeDialog`, `NodeView`. In our case, these classes are prefixed with the name you gave the node in the KNIME Node Wizard, i.e. NumberFormatter.

- `NumberFormatterNodeFactory.java`

  The `NodeFactory` bundles all parts that make up a node. Thus, the factory provides creation methods for the `NodeModel`, `NodeDialog`, and `NodeView`. Furthermore, the factory will be registered via a KNIME extension point such that the node is discoverable by the framework and will be displayed in the node repository view of KNIME Analytics Platform. The registration of this file happens in the `plugin.xml` (see description of the `plugin.xml` file below).

- `NumberFormatterNodeModel.java`

  The `NodeModel` contains the actual implementation of what the node is supposed to do. Furthermore, it specifies the number of inputs and outputs of a node. In this case the node model implements the actual number formatting.

- `NumberFormatterNodeDialog.java` (optional)

  The `NodeDialog` provides the dialog window that opens when you configure (double click) a node in KNIME Analytics Platform. It provides the user with a GUI to adjust node specific configuration settings. In the case of the *Number Formatter* node this is just a simple text box where the user can enter a format String. Another example would be the file path for a file reader node.

- `NumberFormatterNodeView.java` (optional)

  The NodeView provides a view of the output of the node. In the case of the *Number Formatter* node there will be no view as the output is a simple table. Generally, an example for a view could be a tree view of a node creating a decision tree model.

- `NumberFormatterNodeFactory.xml`

This XML file contains the node description and some metadata of the node. The root element must be a `<knimeNode>` ⋯ `</knimeNode>` tag. The attributes of this tag further specify the location of the node icon (`icon="⋯"`) and the type of the node (`type="⋯"`). Note that this is the type you selected in the dialog of the Node Wizard earlier. The most common types are `Source`, `Manipulator`, `Predictor`, `Learner`, `Sink`, `Viewer`, and `Loop`. The description of the node is specified in the children of the root tag. Have a look at the contents of the file for some examples. The `.xml` must be located in the same package as the `NodeFactory` and it has to have the same name (only the file ending differs).

- `default.png`

  This is the icon of the node displayed in the workflow editor. The path to the node icon is specified in the `NumberFormatterNodeFactory.xml` (`icon` attribute of the `knimeNode` tag). In this case the icon is just a placeholder displaying a question mark. For your own node, replace it with an appropriate image representative of what the node does. It should have a resolution of 16x16 pixels.

Apart from the Java classes and the factory `.xml`, which define the node implementation, there are two files that specify the project configuration:

- `plugin.xml` and `META-INF/MANIFEST.MF`

  These files contain important configuration data about the extension project, like dependencies to other plug-ins and the aforementioned extension points. You can double click on the `plugin.xml` to open an Eclipse overview and review some of the configuration options (e.g. the values we entered in KNIME Node Wizard are shown on the overview page under `General Information` on the left). However, you do not have to change any values at the moment.

Overview

General Information

This section describes general information about this plug-in.

ID: `org.knime.examples.numberformatter`

Version: `1.0.0.qualifier`

Name: `NumberFormatter-Node extension for KNIME Workbench`

Vendor: `John Doe`

Platform Filter:

Activator: `org.knime.examples.numberformatter.NumberFormatterNodePlugin` [Browse...]

☑ Activate this plug-in when one of its classes is loaded
☑ This plug-in is a singleton

Execution Environments

Specify the minimum execution environments required to run this plug-in.

[Add...] [Remove] [Up] [Down]

Configure JRE associations...
Update the classpath settings

Plug-in Content

The content of the plug-in is made up of two sections:

- Dependencies: lists all the plug-ins required on this plug-in's classpath to compile and run.
- Runtime: lists the libraries that make up this plug-in's runtime.

Extension / Extension Point Content

This plug-in may define extensions and extension points:

- Extensions: declares contributions this plug-in makes to the platform.
- Extension Points: declares new function points this plug-in adds to the platform.

Testing

Test this plug-in by launching a separate Eclipse application:

- ▶ Launch an Eclipse application
- ▶ Launch a RAP Application
- ✳ Launch an Eclipse application in Debug mode
- ✳ Launch a RAP Application in Debug mode

Exporting

To package and export the plug-in:

1. Organize the plug-in using the Organize Manifests Wizard
2. Externalize the strings within the plug-in using the Externalize Strings Wizard
3. Specify what needs to be packaged in the deployable plug-in on the Build Configuration page
4. Export the plug-in in a format suitable for deployment using the Export Wizard

Figure 6. Eclipse overview of the `plugin.xml` *and* `MANIFEST.MF`.

Number Formatter Node Implementation

Once you have reviewed the project structure, we have a look at some implementation details. We will cover the most important parts as the example code in the project you created earlier already contains detailed comments in the code of the implemented methods (also have a look at the reference implementation in the `org.knime.examples.numberformatter` folder of the `knime-examples` repository).

Generally, the *Number Formatter* node takes a data table as input and applies a user specified format String to each `Double` column of the input table. For simplicity, the output table only contains the formatted numeric columns as String columns. This basically wraps the functionality of the Java `String.format(…)` function applied to a list of `Double` values into a node usable in KNIME Analytics Platform.

Let's work through the most important methods that each node has to implement. The functionality of the node is implemented in the `NumberFormatterNodeModel.java` class:

```
protected NumberFormatterNodeModel() {
    super(1, 1);
}
```

The `super(1, 1)` call in the constructor of the node model specifies the number of output and input tables the node should have. In this case it is one input and one output table.

```
BufferedDataTable[] execute(final BufferedDataTable[] inData, final ExecutionContext exec)
```

The actual algorithm of the node is implemented in the `execute` method. The method is invoked only after all preceding nodes have been successfully executed and all data is therefore available at the input ports. The input table will be available in the given array `inData` which contains as many data tables as specified in the constructor. Hence, the index of the array corresponds to the port index of the node. The type of the input is `BufferedDataTable`, which is the standard type of all tabular data in KNIME Analytics Platform. The persistence of the table (e.g. when the workflow is saved) is automatically handled by the framework. Furthermore, a `BufferedDataTable` is able to handle data larger than the size of the main memory as the data will be automatically flushed to disk if necessary. A table contains `DataRow` objects, which in turn contain `DataCell` objects. `DataCell`s provide the actual access to the data. There are a lot of `DataCell` implementation for all types of data, e.g. a `DoubleCell` containing a floating point number in double precision (for a list of implementations have a look at the type hierarchy of the `DataCell` class). Additionally, each `DataCell` implements one or multiple `DataValue` interfaces. These define

which access methods the cell has i.e. which types it can be represented as. For example, a `BooleanCell` implements `IntValue` as a `Boolean` can be easily represented as 0 and 1. Hence, for each `DataValue` there could be several compatible `DataCell` classes.

The second argument `exec` of the method is the `ExecutionContext` which provides means to create/modify `BufferedDataTable` objects and report the execution status to the user. The most straightforward way to create a new `DataTable` is via the `createDataContainer(final DataTableSpec spec)` method of the `ExecutionContext`. This will create an empty container where you can add rows to. The added rows must comply with the `DataTableSpec` the data container was created with. E.g. if the container was created with a table specification containing two `Double` columns, each row that is added to the container must contain two `DoubleCells`. After you are finished adding rows to the container close it via the `close()` method and retrieve the `BufferedDataTable` with `getTable()`. This way of creating tables is also used in the example code (see `NumberFormatterNodeModel.java`). Apart from creating a new data container, there are more powerful ways to modify already existing input tables. However, these are not in the scope of this quickstart guide, but you can have a look at the methods of the `ExecutionContext`.

The `execute` method should return an array of output `BufferedDataTable` objects with the length of the number of tables as specified in the constructor. These tables contain the output of the node.

```
DataTableSpec[] configure(final DataTableSpec[] inSpecs)
```

The `configure` method has two responsibilities. First, it has to check if the incoming data table specification is suitable for the node to execute with respect to the user supplied settings. For example, a user may disallow a certain column type in the node dialog, then we need to check if there are still applicable columns in the input table according to this setting. Second, to calculate the table specification of the output of the node based on the inputs. For example: imagine the *Number Formatter* node gets a table containing two `Double` columns and one `String` column as input. Then this method should return a `DataTableSpec` (do not forget to wrap it in an array) containing two `DataColumnSpec` of type `String` (the `Double` columns will be formatted to `String`, all other columns are ignored). Analogously to the `execute` method, the `configure` method is called with an array of input `DataTableSpec` objects and outputs an array of output `DataTableSpec` objects containing the calculated table specification. If the incoming table specification is not suitable for the node to execute or does not fit the user provided configuration, throw an `InvalidSettingsException` with an informative message for the user.

```
saveSettingsTo(final NodeSettingsWO settings)
```

and

```
loadValidatedSettingsFrom(final NodeSettingsRO settings)
```

These methods handle the loading and saving of settings that control the behaviour of the node, i.e. the settings entered by the user in the node dialog. This is used for communication between the node model and the node dialog and to persist the user settings when the workflow is saved. Both methods are called with a `NodeSettings` object (in a read only (RO) and write only (WO) version) that stores the settings and manages writing or reading them to or from a file. The `NodeSettings` object is a key-value storage, hence it is easy to write or read to or from the settings object. Just have a look at the provided methods of the `NodeSettings` object in your Eclipse editor. In our example, we do not write settings directly to the `NodeSettings` object as we are using a `SettingsModel` object to store the user defined format String. `SettingsModel` objects already know how to write and read settings from the `NodeSettings` (via methods that accept `NodeSettings`) and help to keep settings synchronization between the model and dialog simple. Furthermore, they can be used to create simple dialogs where the loading and saving of settings is already taken care of.

You can find the actual algorithm of the *Number Formatter* node in the `execute` method in the `NumberFormatterNodeModel.java` class. We encourage you to read through the code of the above mentioned classes to get a deeper understanding of all parts of a node. For a more thorough explanation about how a node should behave consult the KNIME Noding Guidelines.

Deploy your Extension

This section describes how to manually deploy your Extension after you have finished the implementation using the *Number Formatter* Extension as example. There are two options:

Option 1: Local Update Site (recommended)

The first option is to create a local Update Site build, which can be installed using the standard KNIME Analytics Platform update mechanism.

To create a local Update Site build, you need to create a `Feature` project that includes your extension. A Feature is used to package a group of plug-ins together into a single installable and updatable unit. To do so, go to `File → New → Other⋯`, open the `Plug-in Development` category, select `Feature Project` and click the **Next** button.
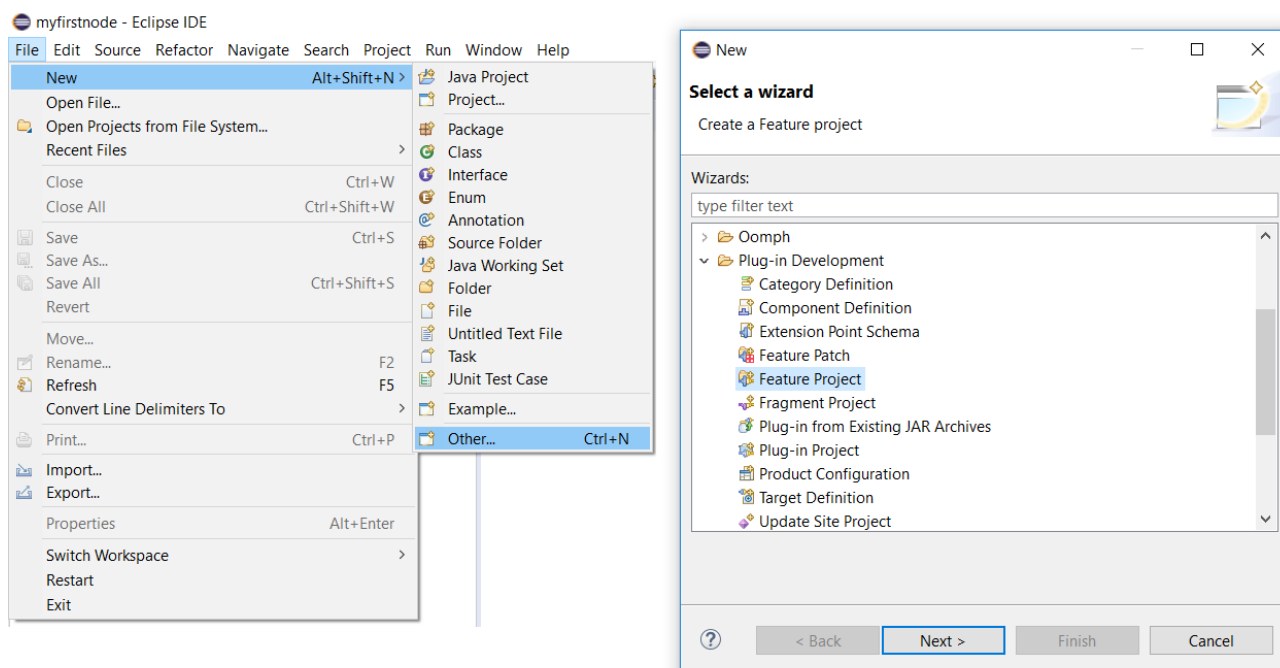

Figure 7. The Feature Project Wizard start dialogs.

Enter the following values in the `Feature Properties` dialog window:

- Project ID: `org.knime.examples.numberformatter.feature`

- Feature Name: `Number Formatter`

- Feature Version: *leave as is*

- Feature Vendor: `<your_name>`

- Install Handler Library: *leave empty*

Replace `<your_name>` with the name that you like to be the author of the created extension. Additionally, choose a location for the new Feature Project (e.g. next to the *Number Formatter* Extension) and click the **Next** button. On the next dialog choose `Initialize from the plug-ins list:` and select the `org.knime.examples.numberformatter` plug-in (you can use the search bar to easily find the plug-in). The plug-ins selected here are the ones that will be bundled into an installable unit by the Feature. Of course, you can edit that list later on. Finally, hit the **Finish** button.
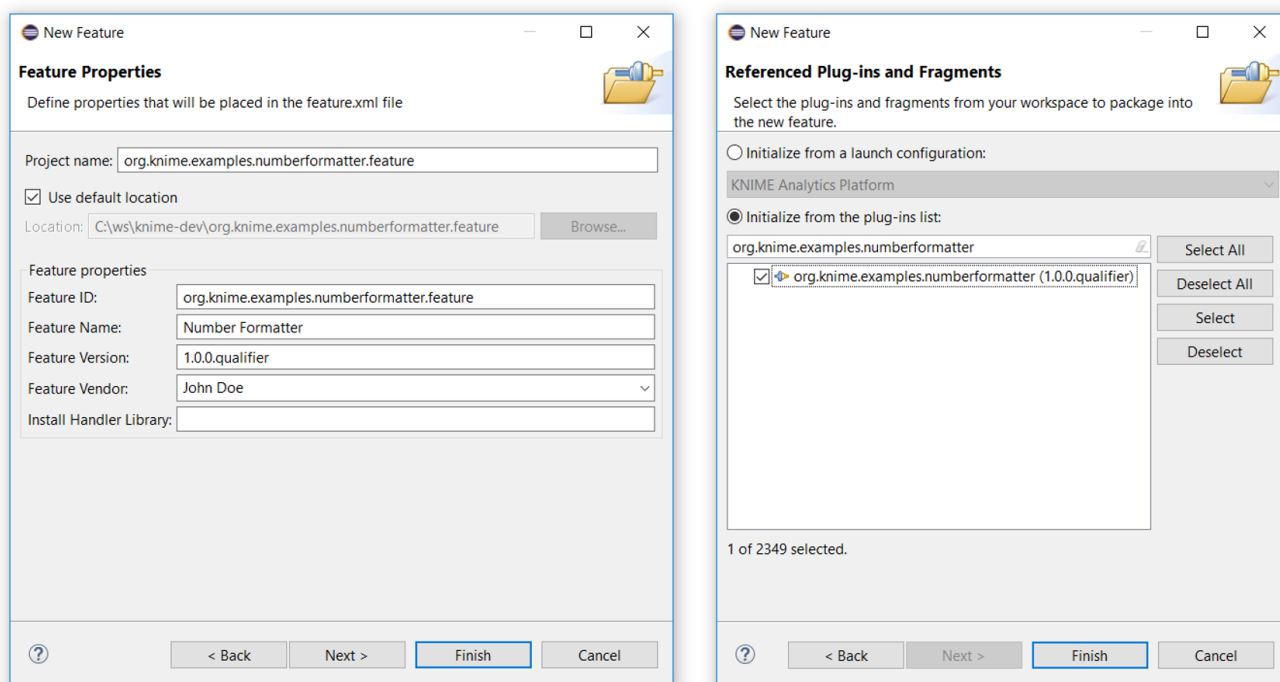

Figure 8. The Feature Project Wizard dialogs.

After the wizard has finished, you will see a new project in your Eclipse Package Explorer with the `Project ID` you gave it earlier and Eclipse will automatically open an overview of the `feature.xml` (you can also open this view by double clicking on the `feature.xml` file located in the Feature Project). The Feature overview looks similar to the `plugin.xml` overview, be careful not to confuse them. You can view/modify the list of included plug-ins by selecting the `Included Plug-ins` tab at the bottom of the overview dialog.

> ℹ️ Additionally to the information you entered in the Feature Project Wizard, you should provide a detailed Feature description, license and copyright information in the Feature meta data. This can be done by selecting the `Information` tab at the bottom of the overview dialog. This information will be displayed to the user during installation of the Feature.
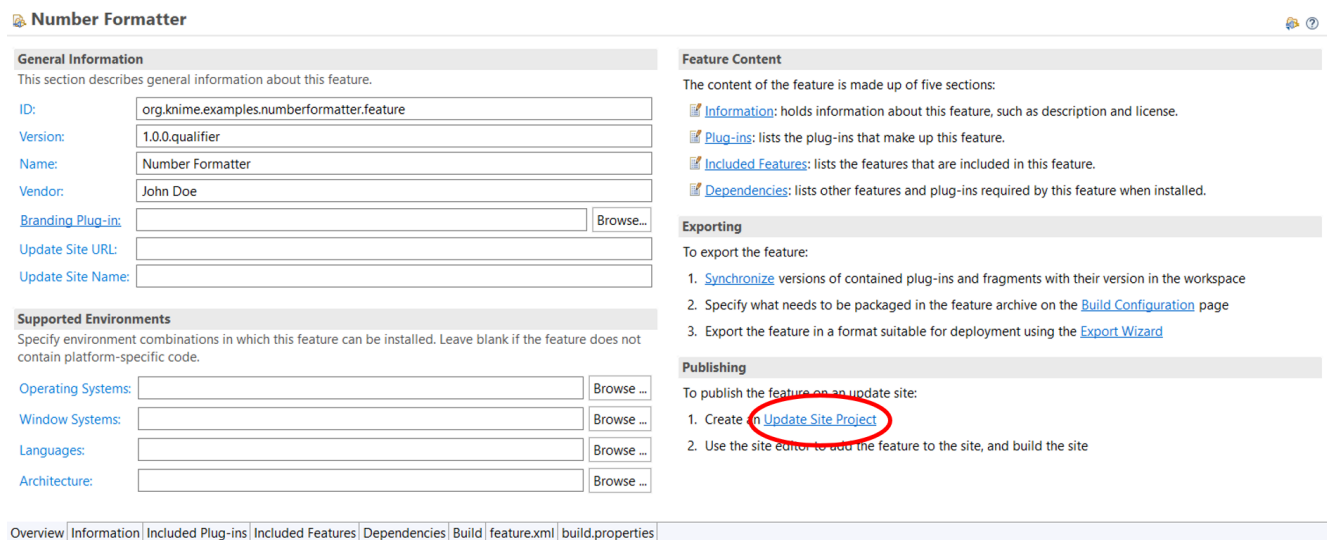
Figure 9. Eclipse overview of the `feature.xml`. *The link to create an Update Site Project is marked in red.*

Next, you need to publish the Feature on a local Update Site. For this, first create an `Update Site Project` by clicking on the `Update Site Project` link on bottom right corner of the Eclipse overview dialog of the `feature.xml` (see figure above). This will start the Update Site Project Wizard.
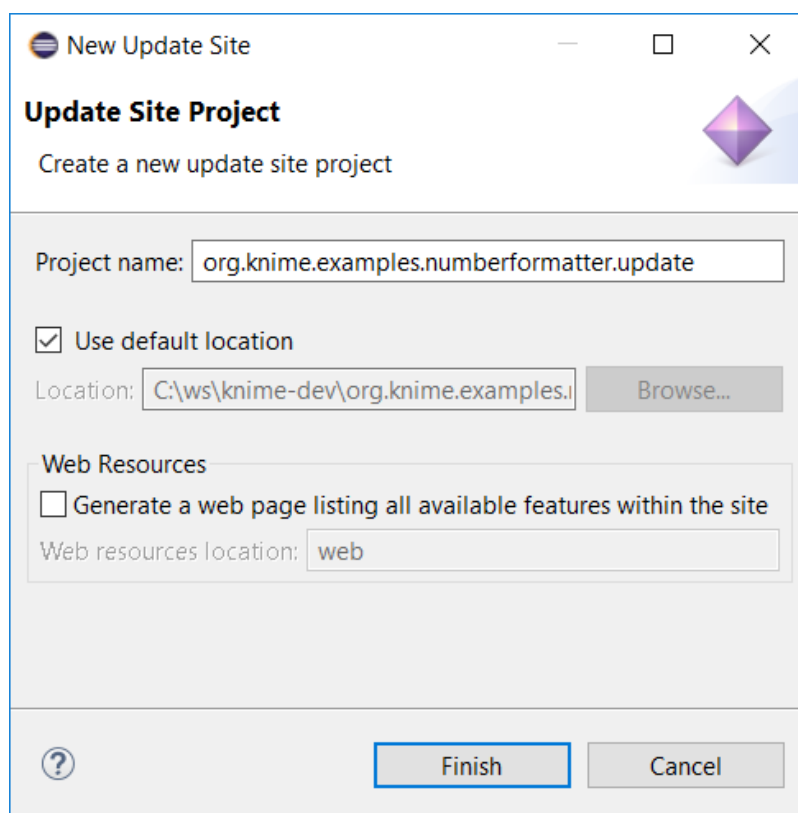

Figure 10. The Update Site Project Wizard dialog.

On the shown dialog, enter the following:

- Project name: `org.knime.examples.numberformatter.update`

Again, choose a location for the new Update Site Project and click the **Finish** button. Similar to the Feature Project Wizard, you will see a new project in your Eclipse Package Explorer with the `Project name` you gave it in the wizard dialog and Eclipse will automatically open an overview of the `site.xml` called `Update Site Map`. Again similar to a Feature, an Update Site bundles one or several Features that can be installed by the Eclipse update mechanism.
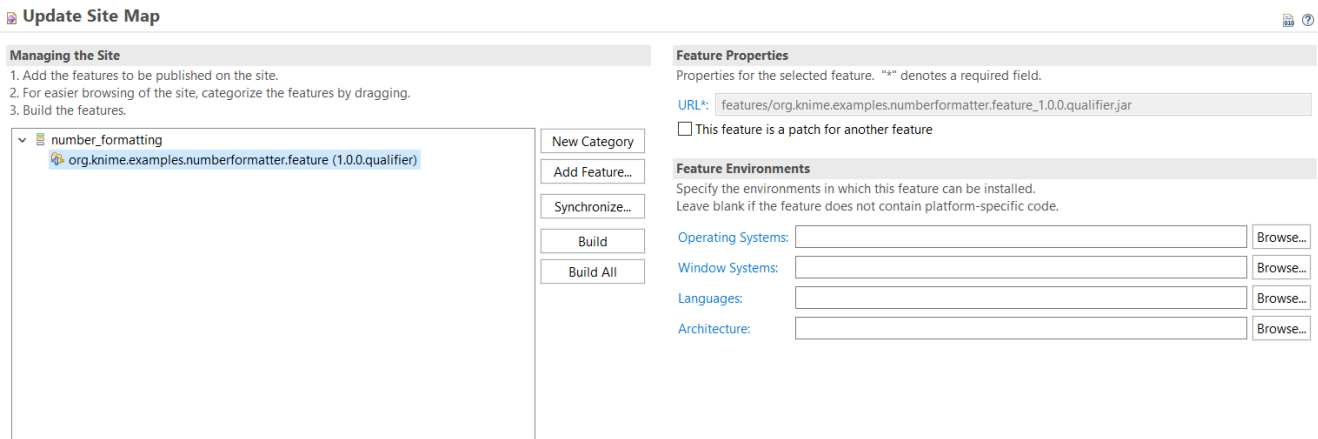

Figure 11. Eclipse overview of the `site.xml`*. The Update Site in this image already contains one category called* `number_formatting` *where the* `org.knime.examples.numberformatter.feature` *was added to. This way the Number Formatter Extension will be listed under this category during installation.*

On the Eclipse overview of the `site.xml`, first create a new category by clicking on the **New Category** button. This will create a new default category shown in the tree view on the left. On the right, enter an `ID` like `number_formatting` and a `Name` like `Number Formatting`. This name will be displayed as a category and used for searching when the Feature is installed. Also, provide a short description of the category.

Second, select the newly created category from the tree view and click the **Add Feature…** button. On the shown dialog, search for `org.knime.examples.numberformatter.feature` and click the **Add** button.
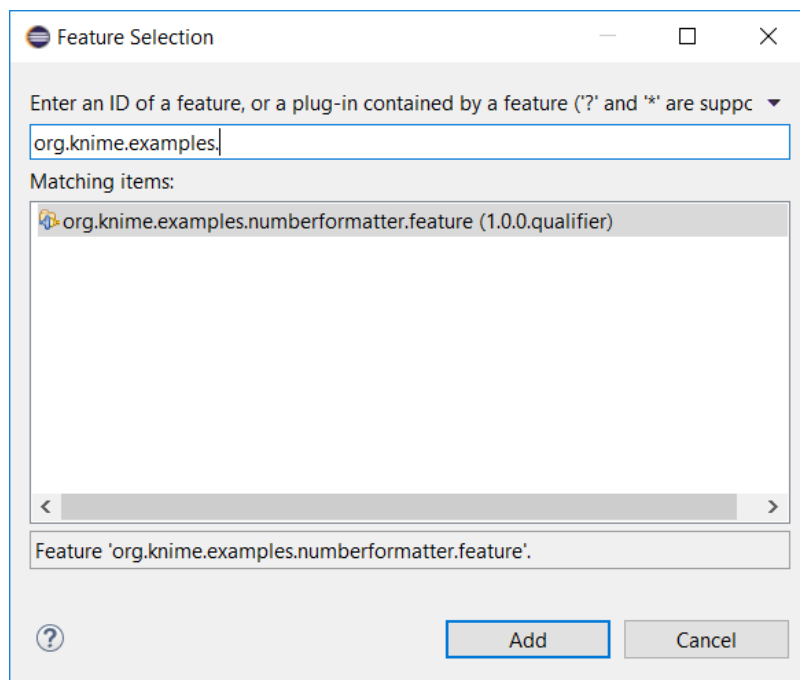
Figure 12. The Feature Selection dialog.

At last, click the **Build All** button. This will build all Features added to the update site and create an installable unit that can be used to install the *Number Formatter* Extension into an KNIME Analytics Platform instance.

> **i** The building of the Update Site might take some time. You can review the progress in the bottom right corner of Eclipse.

After building has finished, you can now point KNIME Analytics Platform to this folder (which now contains a local Update Site) to install the Extension. To do so, in KNIME Analytics Platform open the `Install New Software…` dialog, click on the **Add** button next to the update site location, on the opening dialog click on **Local…**, and choose the folder containing the Update Site. At last, give the local Update Site a name and click **OK**. Now, you can install the *Number Formatter* Extension like any other plug-in.

> **i** Now that you have a working extension, why not sharing it with the community? Take a look at the following guide: Publish Your Extension on KNIME Community Hub

Option 2: dropin

The second option is to create a `dropin` using the `Deployable plug-ins and fragments` Wizard from within Eclipse. A `dropin` is just a `.jar` file containing your Extension that is simply put into the Eclipse `dropins` folder to install it.

To create a `dropin` containing your Extension, go to `File` → `Export` → `Plug-in Development` → `Deployable plug-ins and fragments` and click **Next**. The dialog that opens will show a list of deployable plug-ins from your workspace. Check the checkbox next to `org.knime.examples.numberformatter`. At the bottom of the dialog you are able to select the export method. Choose `Directory` and supply a path to a folder where you want to export your plugin to. At last click **Finish**.
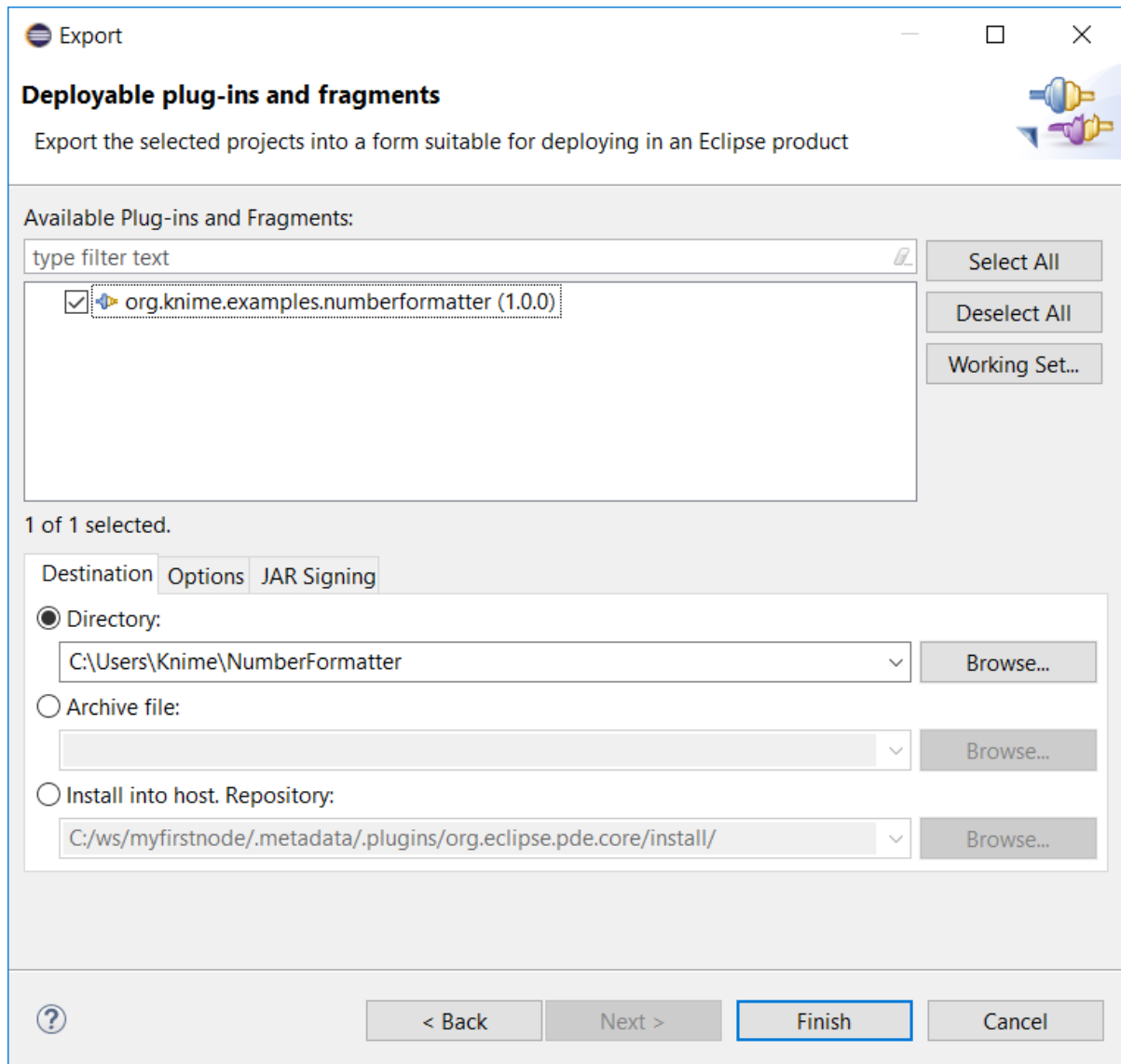

Figure 13. The dialog of the deploy wizard.

After the export has finished, the selected folder will contain a `.jar` file containing your plugin. To install it into any Eclipse or KNIME Analytics Platform installation, place the `.jar` file in the `dropins` folder of the KNIME/Eclipse installation folder. Note that you have to restart KNIME/Eclipse for the new plugin to be discovered. In this example, the node is then displayed at the top level of the node repository in KNIME Analytics Platform.

Further Reading

- For more information on development see the Developers Section of the KNIME website.

- KNIME source code

- If you have questions regarding development, reach out to us in the KNIME Development category of our forum.

Open for Innovation

KNIME

KNIME AG
Talacker 50
8001 Zurich, Switzerland
www.knime.com
info@knime.com