Plugin Development Basics

Overview

While it's possible to develop a Yellowfin plugin using a text editor and Java command line tools, it would become frustrating if you were developing anything more complicated than a simple Hello World plugin. This guide provides essential tips and steps to help developers set up their IDE to help them code Yellowfin supported plugins, widgets, advanced functions, and more.

Tools to develop and debug your plugins

To develop and debug your plugins, you will need some tools and files.

- Java 8 JDK (or higher) on your development system;
- a Yellowfin install;
- a project that outputs to your Yellowfin folder; and,
- a META-INF/services file.

The steps below will help you locate and create each of these.

Install Java 8 JDK (or higher) on your development system

You can download Java 8 JDK here:

https://www.oracle.com/au/java/technologies/javase/javase-jdk8-downloads.html

Install Yellowfin for development work

Make sure you have a Yellowfin Install that is compatible with the Yellowfin plugin you wish to develop. This installation must have been started at least once so the WAR file can be extracted into *<Yellowfin Install directory>/appserver/webapps/ROOT/*

We recommend you use the most recent version of Yellowfin, available here.

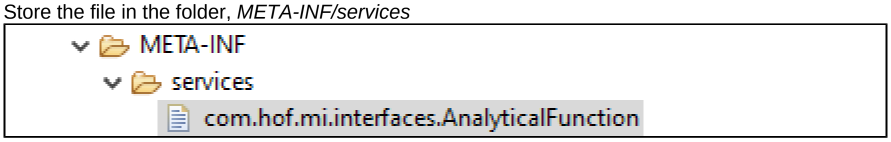
Create a project that outputs to your Yellowfin folder

You will need to create a project that outputs it's compiled classes to *<Yellowfin Install directory>/appserver/webapps/ROOT/classes*

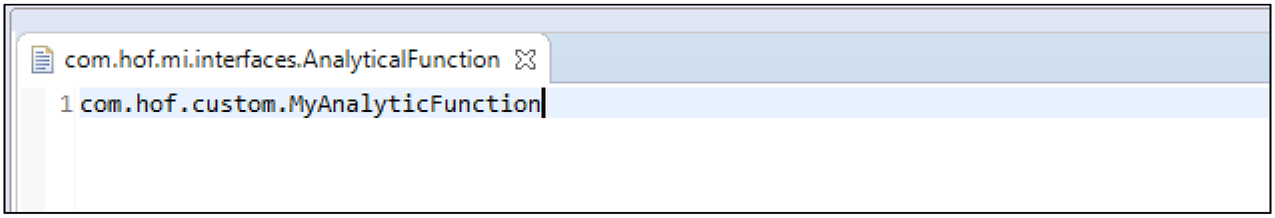
You can follow the steps below to do this in Eclipse.

Create a *META-INF/services* files

A *META-INF/services* file can be used to tell Yellowfin to load certain Plugins when it is starting up.

1. Create a file having the fully-qualified name of the plugin interface, in the services directory.
2. To create an Analytic Function, we recommend creating a file:, *com.hof.mi.interfaces.AnalyticalFunction*
3. Store the file in the folder, *META-INF/services*


4. *Store the file in the folder:*



You're now ready to set up your IDE.

top

Setting up your IDE

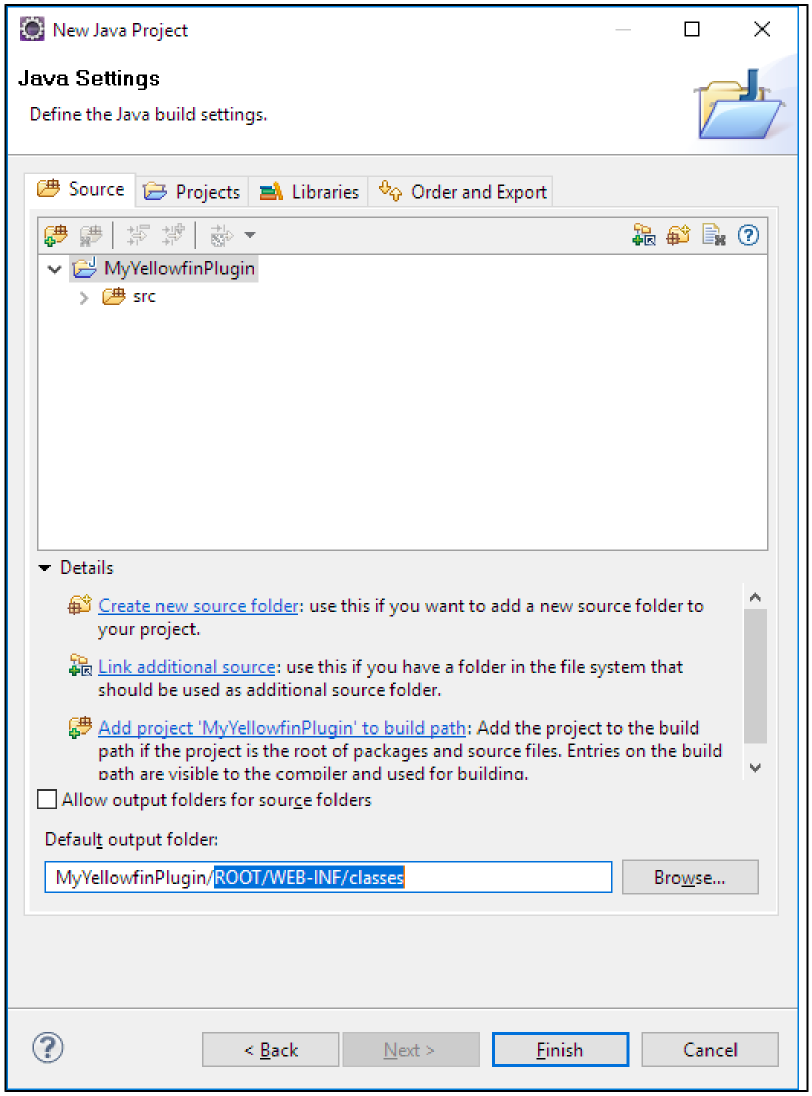
Here's a step-by-step tutorial to help you set up your development environment. We've used Tomcat as an example, which is Yellowfin's preferred IDE; however, you should be able to loosely follow the instructions below for your preferred IDE.

1. Download and set up Eclipse for EE developers.
2. Install the Tomcat plugin, if it isn't already bundled with the installation.
3. Install the version of Yellowfin to be used for development. **Note:** We suggest getting the latest version for increased productivity.
4. Start up Yellowfin to extract the WAR file.

Create a Plugin Project

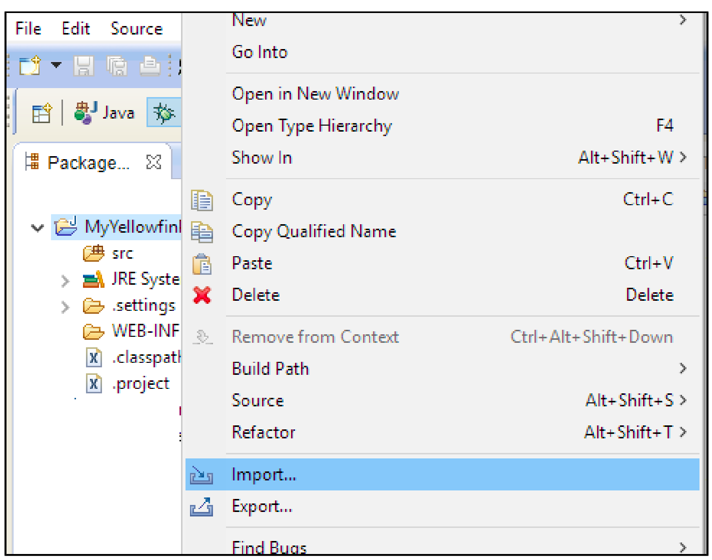
These steps will help you to create a new project for your Java plugin.

1. On starting Eclipse, create a new Java project.
    a. Enter the project name and ensure you select a JRE compatible with your version of Yellowfin.
    b. Click **Next** and change the default output folder to **<project-name>/ROOT/WEB-INF/classes**.

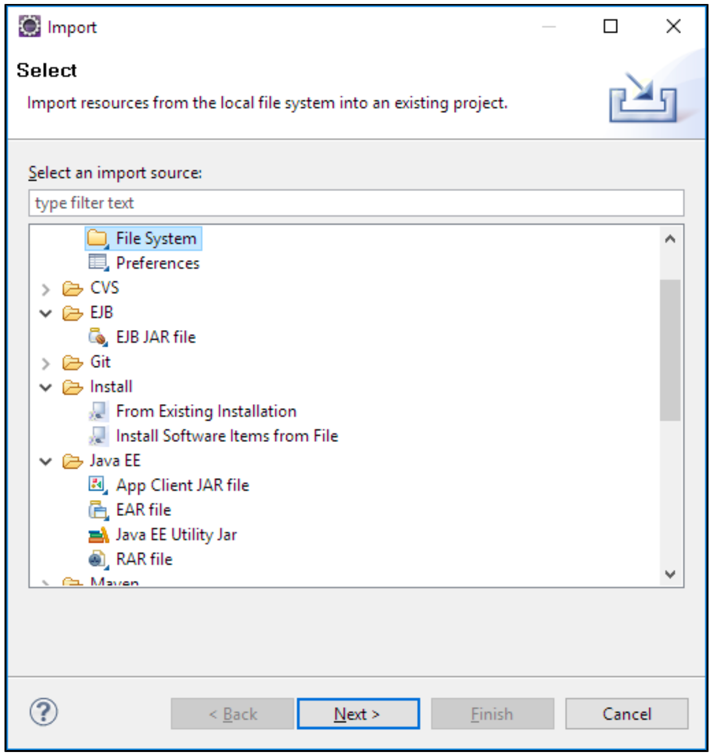


c. Click **Finish**.

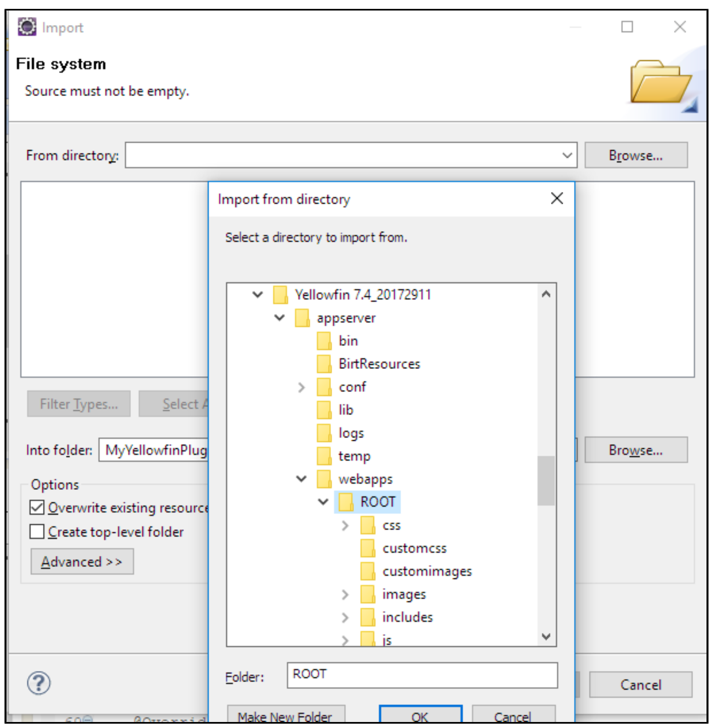
2. Import files from the installed Yellowfin instance:
   a. Right-click on the project and select **Import**.



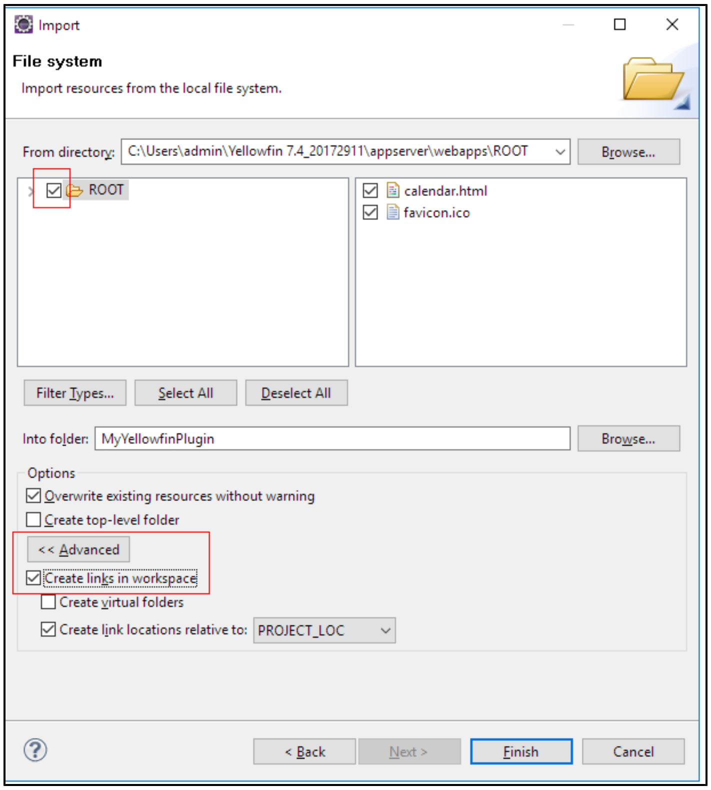
b. Select **File System** and click **Next**.



c. Navigate to **appserver/webapps/ROOT** in the Yellowfin install directory. Select **ROOT** and click **OK**.



d. Select everything under **ROOT** and in the **Advanced** section, select the **Create links in workspace** checkbox.



e. Click **Finish**. Files from the installed Yellowfin will be linked to this project.

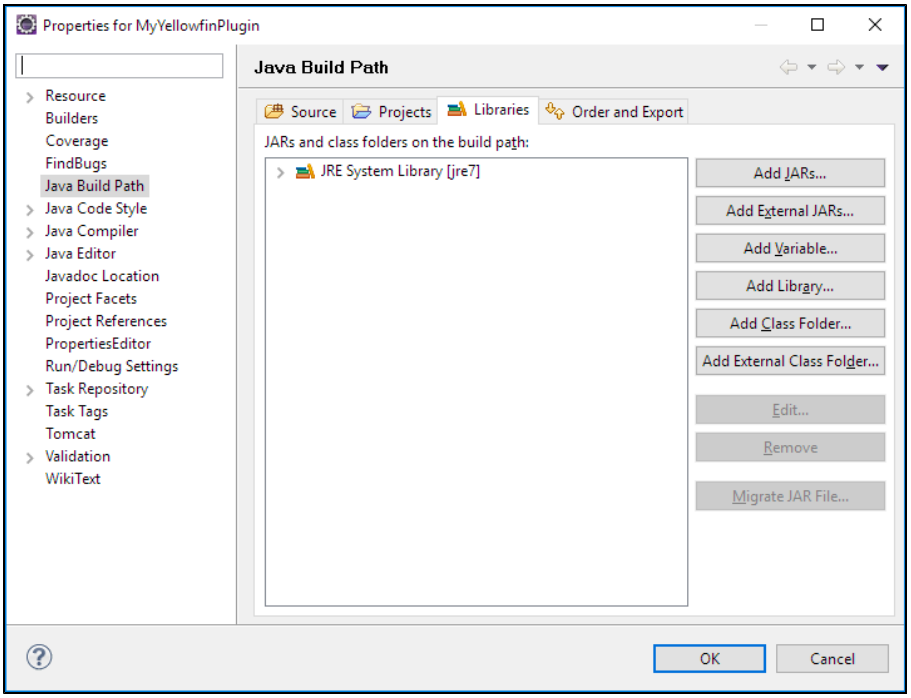
---

top

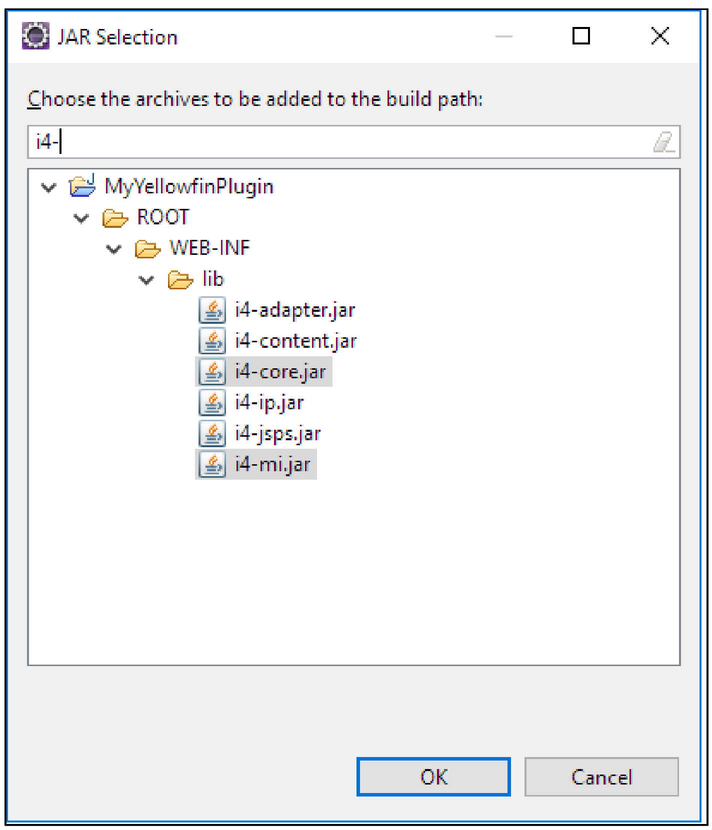
Configure the Project

Follow these steps to configure your project.

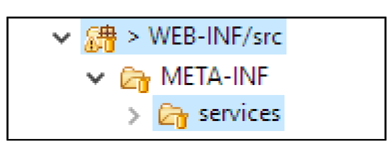
1. Right-click on the project and select **Build Path** > **Configure Build Path** from the menu. Then select the **Libraries** tab.



2. Click the **Add JARs** button and type "**i4**" into the search bar. From the results, select **i4-core.jar** and **i4-mi.jar** from your plugin project.



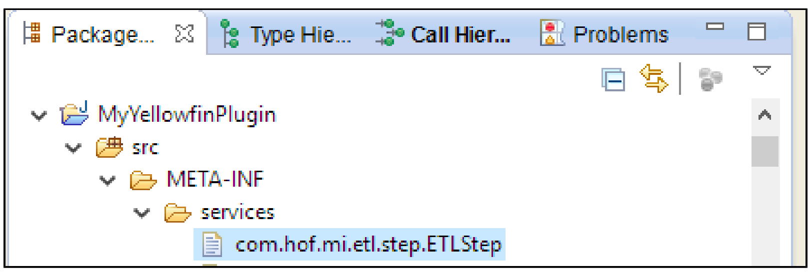
3. Click **OK** to save this and **OK** again in the build path config window.
4. Under the **WEB-INF/src** folder, create a new folder and call it **META-INF**. Create a new folder called **services** within this one.



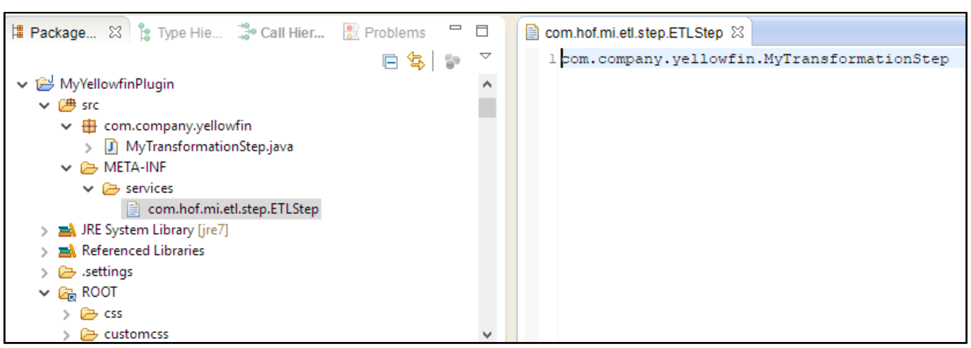
5. Depending on which plugin is being developed, create a file with the fully qualified name of the plugin interface in the **services** directory. See the table below for all available plugin options:

Yellowin Plugin	Interface	Description
Transformation Step	com.hof.mi.etl.step.ETLStep	A step which may be used in the Data Transformation module.
Advanced Function	com.hof.mi.interfaces.AnalyticalFunction	Advanced functions used in Reports.
Data Type Converter	com.hof.mi.interfaces.Converter	Conversion of data types, done at the View Level and in the Data Transformation module.
Custom Formatter	com.hof.mi.interfaces.CustomFormatter	Custom formatting used in Reports.
Data Profiler	com.hof.mi.interfaces. DataSuggestionPlugin	Profile data for a field. Contains functionality to determine whether the implemented suggestion is applicable.
Third-Party Connector	com.hof.mi.thirdparty.interfaces. AbstractDataSource	Connectors to create connections to external API data sources.
Canvas Widget	com.hof.mi.widgetcanvas.interfaces. CanvasObjectTemplate	Custom widgets used in canvases in the Dashboard, Storyboard and Report Design modules.
Source Platform	com.hof.sources.SourcePlatform	Define source types, such as JDBC, JNDI, OLAP etc.

a. For instance, if creating a Data Transformation Step, name your file '**com.hof.mi.etl.step.ETLStep**'.



6. Create the plugin class by implementing one of the interfaces given above. The fully qualified classname should be added to the services file corresponding to the interface.
So, for our Data Transformation Step example, add its fully qualified classname to **META-INF/services/com.hof.etl.step.ETLStep**.



You can add further transformation steps below this line, if required.

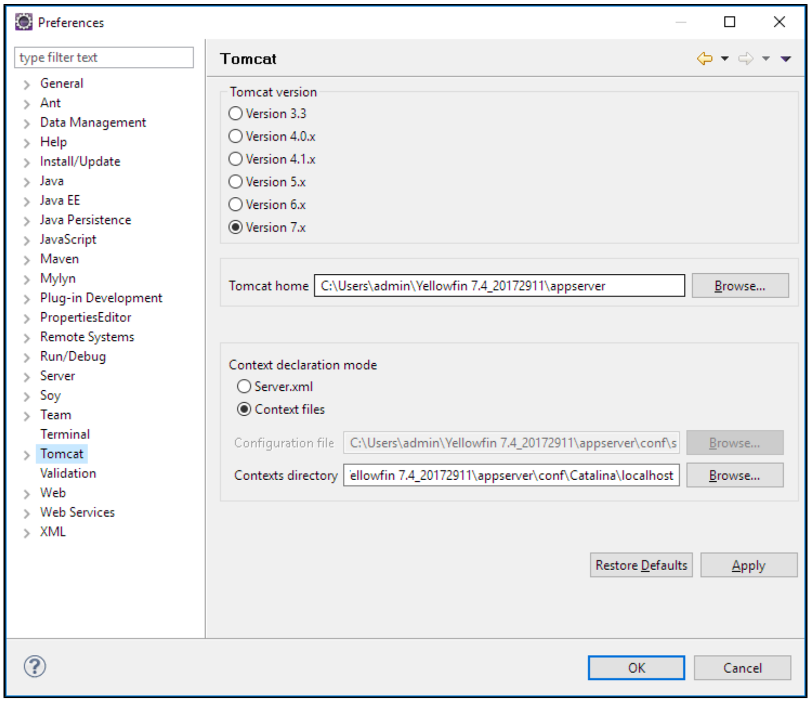
---

top

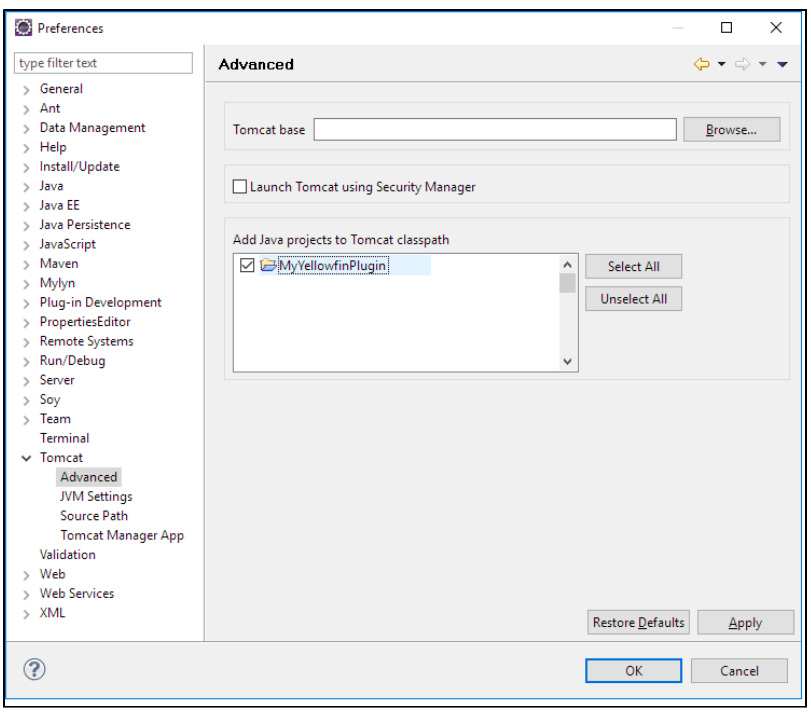
Configure Tomcat

The next step is to set up your Tomcat configuration.

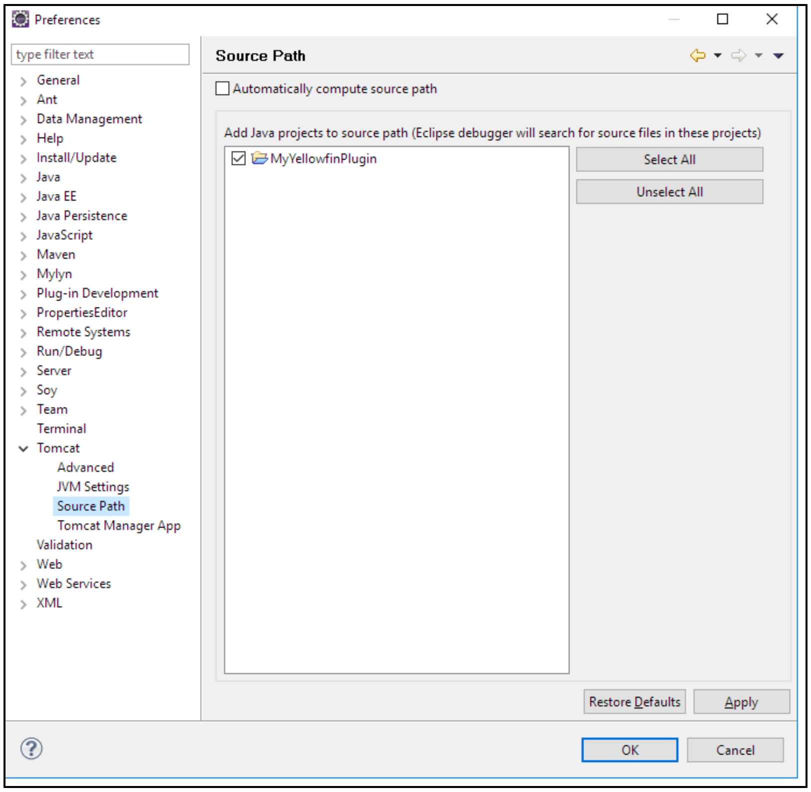
1. Select **Window > Preferences** and go to the section for **Tomcat**.



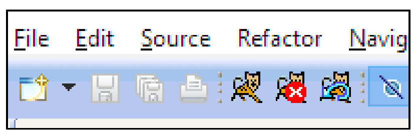
2. Set **Tomcat home** to *<Yellowfin Install directory>/appserver* and **Contexts directory** to *<Yellowfin Install directory>/appserver/conf/Catalina /localhost.*
3. Expand the **Tomcat** section in the left-side menu and click on **Advanced**. Add the plugin project to Tomcat's classpath.



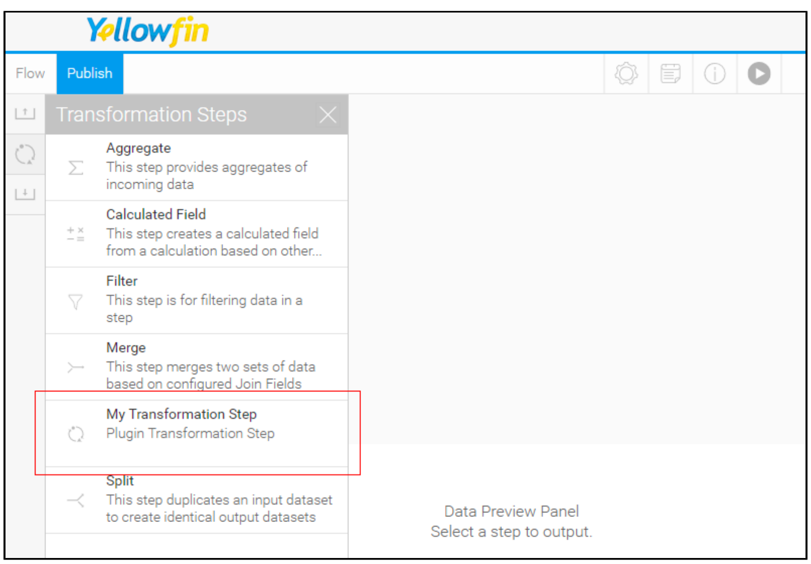
4. Adjust JVM Settings, if necessary (through the **JVM Settings** option on the left side). **Tip:** You could use this to increase the memory available for Tomcat.
5. Select **Source Path** (in the left side menu) and add the plugin project.



6. Click **OK** to save.
7. Start up Tomcat from Eclipse using the buttons in the toolbar.



8. The plugin will be now be available in Yellowfin.



Changes to code get reflected instantly, except when:

- a method's signature is changed,
- new methods/members are are added to the class,

- new classes are added to the plugin package.

In these cases, Tomcat must be restarted to apply changes.

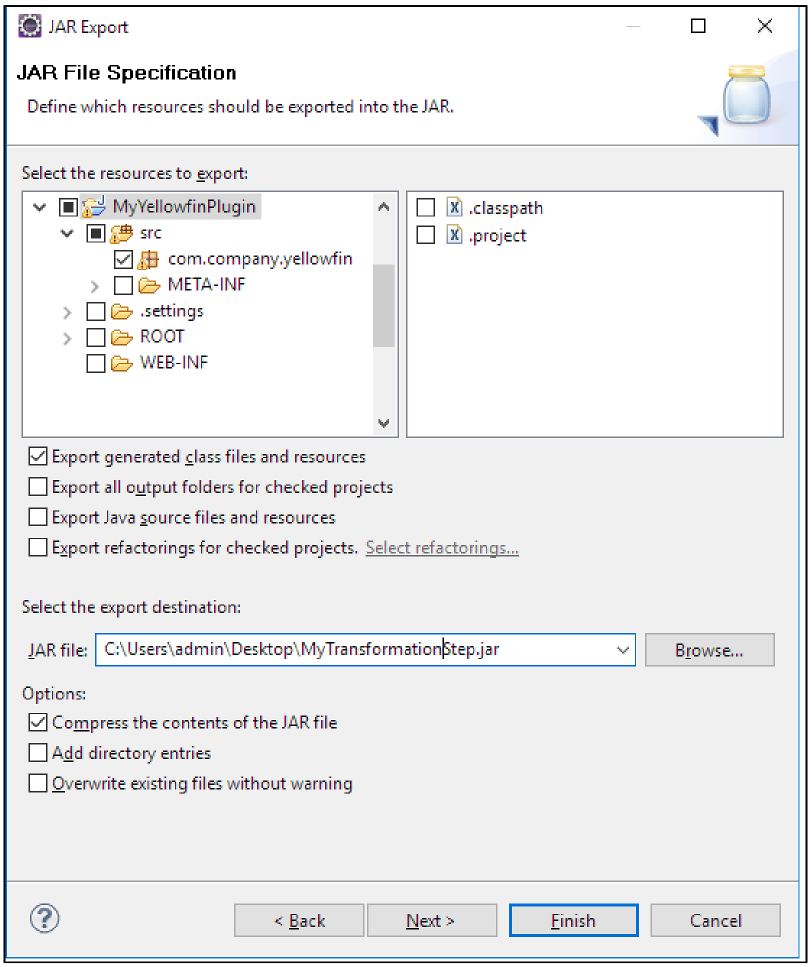
---

top

Packaging the Plugin

Once you've created your plugin, you will need to package it will all of its dependencies. The file extension should be in a specific file format that is supported in Yellowfin.

1. Right-click on the project and select **Export > JAR file**.
2. Select only the package(s) to be exported and nothing else.



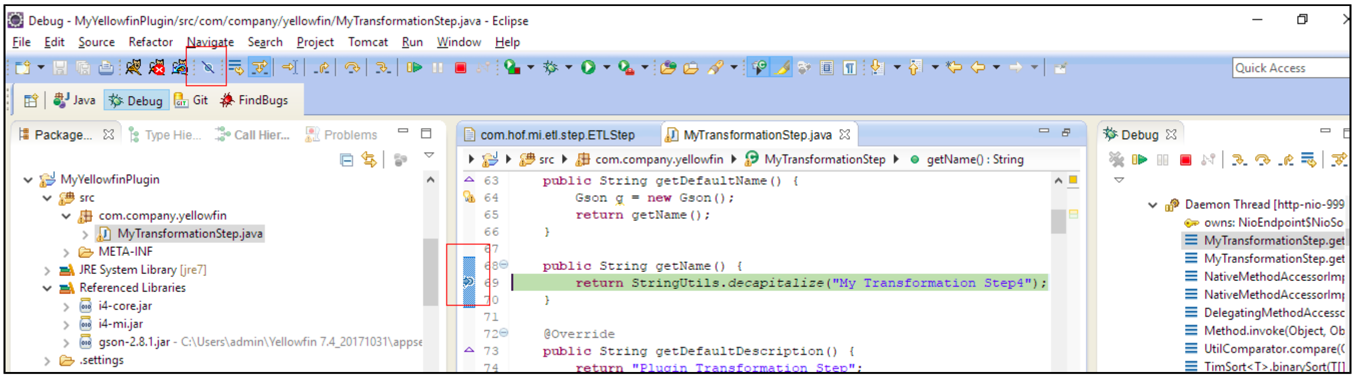
3. If the project has dependent JARs, put all of them and the Plugin JAR into one directory, zip into one archive, and give it the extension "**yfp**".

---

top

Debugging

Debugging is easy as the Eclipse Tomcat plugin starts Tomcat in the debug mode. Simply add breakpoints in code and ensure they are active.


Debug - MyYellowfinPlugin/src/com/company/yellowfin/MyTransformationStep.java - Eclipse
File Edit Source Refactor Navigate Search Project Tomcat Run Window Help
Quick Access
Java Debug Git FindBugs
Package... Type Hie... Call Hier... Problems
MyYellowfinPlugin
src
com.company.yellowfin
MyTransformationStep.java
META-INF
JRE System Library [jre7]
Referenced Libraries
i4-core.jar
i4-mi.jar
gson-2.8.1.jar - C:\Users\admin\Yellowfin 7.4_20171031\appse
.settings
com.hof.mi.etl.step.ETLStep
MyTransformationStep.java
src com.company.yellowfin MyTransformationStep getName() : String
public String getDefaultName() {
Gson g = new Gson();
return getName();
}
public String getName() {
return StringUtils.decapitalize("My Transformation Step4");
}
@Override
public String getDefaultDescription() {
return "Plugin Transformation Step";
Debug
Daemon Thread [http-nio-999
owns: NioEndpoint$NioSo
MyTransformationStep.get
MyTransformationStep.get
NativeMethodAccessorImp
NativeMethodAccessorImp
DelegatingMethodAccessc
Method.invoke(Object, Ob
UtilComparator.compare(
TimSort<T>.binarySort(T[]