Running Cucumber From the Command Line – https://dzone.com/articles/running-cucumber-from-the-command-line

Learn the steps and conditions for running the Cucumber automated testing tool from the command line in this quick walkthrough.

In response to accelerated release cycles, a new set of testing capabilities is now required to deliver quality at speed. This is why there is a shake-up in the testing tools landscape—and a new leader has emerged in the just released Gartner Magic Quadrant for Software Test Automation.

Recently, I've been spending some time with Cucumber and joined the Cucumber Gitter channelwhen somebody pointed out that they were having trouble running Cucumber from the command line. I usually run Cucumber from Maven, so I thought it would be interesting to see what was required to run cucumber from the command line.

As described in the documentation, you can run Cucumber features from the command line by using the CLI-runner with the following command:

```
java cucumber.api.cli.Main
```

You can get help by using the --help option:

```
java cucumber.api.cli.Main --help
```

This looks pretty straightforward if you are familiar with Java, but the "hard" part comes from understanding how to use the Java command with its classpath options.

One important aspect is that the cucumber.api.cli.Main class is located in the cucumber-core jar file, so when you want to run this class, you need to provide the cucumber-core jar on your classpath. In this case, I take the jar(s) from my Maven repository and include all required dependencies:

```
$ java -cp "/Users/jreijn/.m2/repository/info/cukes/cucumber-core/1.2.5/cucumber-core-1.2.5.jar:/Users/jreijn/.m2/repository/info/cukes/gherkin/2.12.2/gherkin-2.12.2.jar:/Users/jreijn/.m2/repository/info/cukes/cucumber-java/1.2.5/cucumber-java-1.2.5.jar:/Users/jreijn/.m2/repository/info/cukes/cucumber-jvm-deps/1.0.5/cucumber-jvm-deps-1.0.5.jar" cucumber.api.cli.Main
```

If you run this it should result in the following message:

```
Got no path to feature directory or feature file
0 Scenarios
0 Steps
0m0.000s
```

Now to be able to run we need to run a feature file you will need to provide two additional arguments:

1. Your feature file(s)
2. Your glue code (step definitions, hooks, etc)

Your feature files can be added to the end of the command line:

```
$ java -cp "/Users/jreijn/.m2/repository/info/cukes/cucumber-core/1.2.5/cucumber-core-1.2.5.jar:/Users/jreijn/.m2/repository/info/cukes/gherkin/2.12.2/gherkin-2.12.2.jar:/Users/jreijn/.m2/repository/info/cukes/cucumber-java/1.2.5/cucumber-java-1.2.5.jar:/Users/jreijn/.m2/repository/info/cukes/cucumber-jvm-deps/1.0.5/cucumber-jvm-deps-1.0.5.jar" cucumber.api.cli.Main Developer/sources/github/cucumber-jvm-extentreport/src/test/resources/cucumber/feature_one.feature
```

This will probably result in the following message:

```
UUUUUU
3 Scenarios (3 undefined)
6 Steps (6 undefined)
0m0.000s
You can implement missing steps with the snippets below:
[snip]
```

This means it can't find the step definitions, hooks, etc that correspond to your feature file.
Let's add the glue code required for running the tests. In the below example I'll use my maven projects target directory, which contains my step definitions in the test-classes directory. You can do that by adding the directory to your classpath, and with the argument --glue com.sitture.definitions, provide the Java package that contains step definition Java classes.

```
$ java -cp "/Users/jreijn/.m2/repository/info/cukes/cucumber-core/1.2.5/cucumber-core-1.2.5.jar:/Users/jreijn/.m2/repository/info/cukes/gherkin/2.12.2/gherkin-2.12.2.jar:/Users/jreijn/.m2/repository/info/cukes/cucumber-java/1.2.5/cucumber-java-1.2.5.jar:/Users/jreijn/.m2/repository/info/cukes/cucumber-jvm-deps/1.0.5/cucumber-jvm-deps-1.0.5.jar:/Users/jreijn/Developer/sources/github/cucumber-jvm-extentreport/target/test-classes/" cucumber.api.cli.Main --glue com.sitture.definitions Developer/sources/github/cucumber-jvm-extentreport/src/test/resources/cucumber/feature_one.feature
```

This should result in something similar to:

```
......
....
.....
3 Scenarios (3 passed)
6 Steps (6 passed)
0m0.067s
```

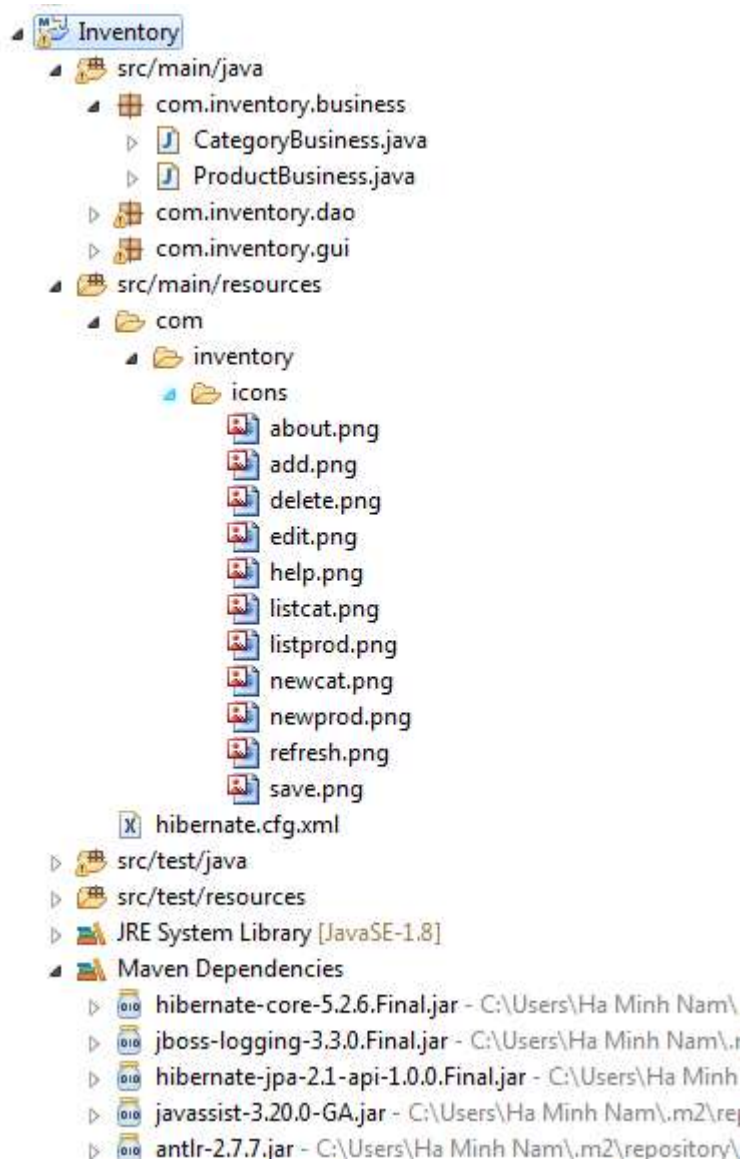
Which seems about right and shows how we can run Cucumber from the command line with the Cucumber CLI.

How to Create Executable JAR file with resources and dependencies using Maven in Eclipse

Last Updated on 21 February 2017 | Print Email

Java Spring Framework Masterclass: Beginner to Professional

In this article, we are going to guide you how to create an executable JAR file from a Java project which uses Maven build system. It's worth mentioning that the project contains resource files (XML configuration, images, etc) that are in a directory different from the Java source files directory. And the project also contains some dependencies as well. The following screenshot illustrates such a Java project:



As you can see, this Java project contains resource files like images and XML, and dependencies for Hibernate framework. So what we are going to show you is how to generate the executable JAR file of this project in a manner that the JAR file contains all the resources and dependencies (uber-JAR or fat JAR file).
And you know, creating such JAR file is not possible with Eclipse's Export function (File > Export > Java > Runnable JAR file). But the good news is that Maven provides a great plugin that is dedicated for creating executable JAR files with resources and dependencies. This plugin is called Maven Assembly Plugin.
To use the Maven Assembly Plugin, declare the following XML code in the **<plugins>** section which is under the **<build>**section of the **pom.xml** file:

```
1       <plugin>
2         <artifactId>maven-assembly-plugin</artifactId>
3         <configuration>
4           <archive>
5             <manifest>
6               <mainClass>com.inventory.gui.InventoryApp</mainClass>
7             </manifest>
8           </archive>
9           <descriptorRefs>
10            <descriptorRef>jar-with-dependencies</descriptorRef>
11          </descriptorRefs>
```

```
12        </configuration>
13    </plugin>
```

That makes the pom.xml file looks like the following:

```
      <project ....>
1       <modelVersion>4.0.0</modelVersion>
2       <groupId>com.inventory</groupId>
3       <artifactId>Inventory</artifactId>
4       <packaging>jar</packaging>
5       <version>0.0.1-SNAPSHOT</version>
6       <name>Inventory Management</name>
7
8
9       <dependencies>
10        ....
11
12      </dependencies>
13
14      <build>
15        <plugins>
16          <plugin>
17            <artifactId>maven-assembly-plugin</artifactId>
18            <configuration>
19              <archive>
20                <manifest>
21                  <mainClass>com.inventory.gui.InventoryApp</mainClass>
22                </manifest>
23              </archive>
24              <descriptorRefs>
25                <descriptorRef>jar-with-dependencies</descriptorRef>
26              </descriptorRefs>
27            </configuration>
28          </plugin>
29        </plugins>
30      </build>
31    </project>
32
```

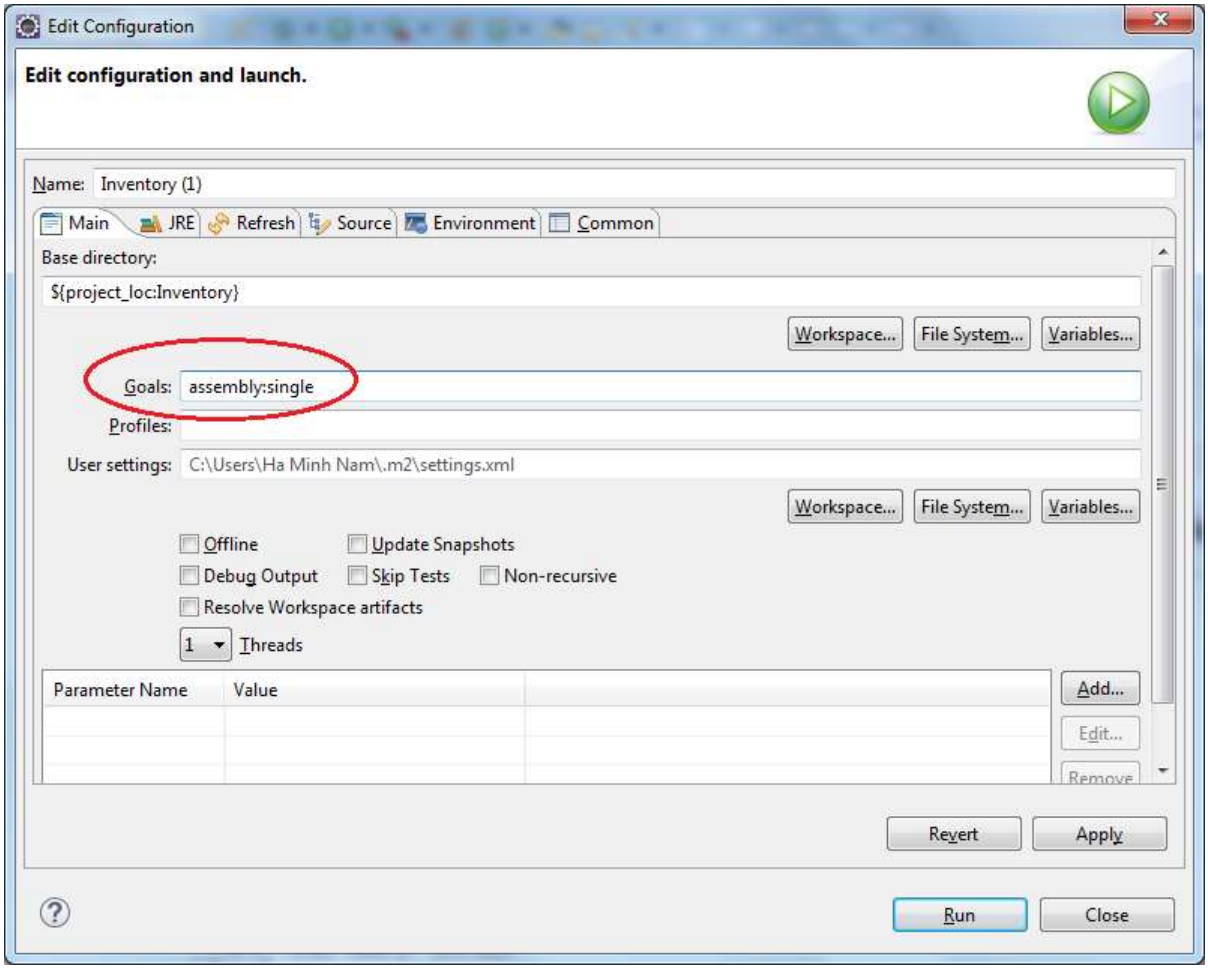
Here, there are two noteworthy points:

- The packaging type of the project must be jar: **<packaging>jar</packaging>**
- The **<mainClass>** element specifies the main class of your application in form of fully qualified name. For example, in the above XML we specify the main class is com.inventory.gui.InventoryApp

And to generate the JAR file from the project, run Maven with the goal **assembly:single**. For example, in the command line:

```
mvn clean install assembly:single
```

In Eclipse, right click on the project and select **Run As > Maven build**, and specify the goal **assembly:single** in the dialog, then click **Run**:



Wait a moment for the build to complete, and then check the JAR file generated under the project's **target** directory. Note that the JAR file contains all the resources and dependencies (fat JAR or uber-JAR).

References:

Maven Assembly Plugin