|  |
| --- |
| minigameslib.de |
| Spigot Eclipse-Plugin |
| Eclipse-Plugin and JUnit support |

|  |
| --- |
| mepeisen  12.11.2016 |

Table of contents

[Eclipse setup 2](#_Toc465320516)

[Setup spigot servers 3](#_Toc465320517)

[Installing third party plugins 6](#_Toc465320518)

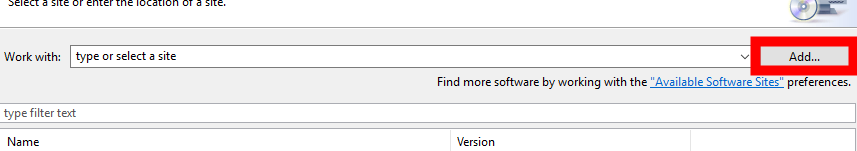
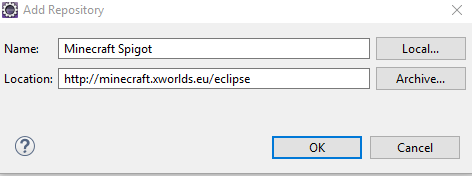
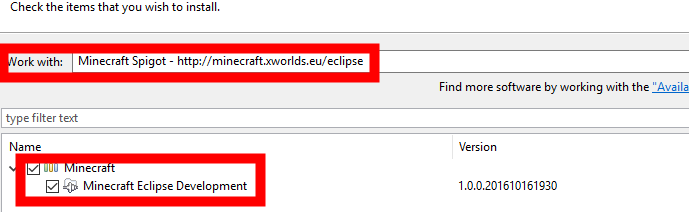
[First plugin project 6](#_Toc465320519)

[Hot code replacement. 9](#_Toc465320520)

[Junit support 11](#_Toc465320521)

# Eclipse setup

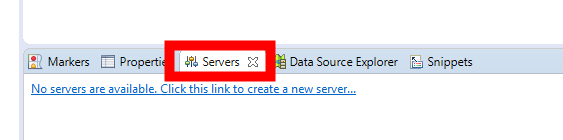
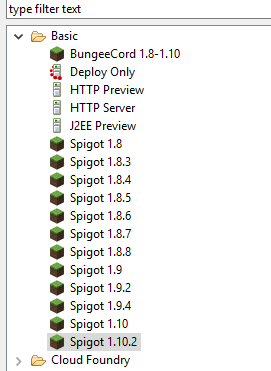
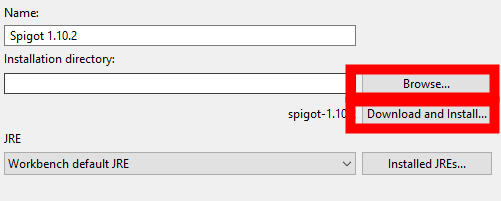
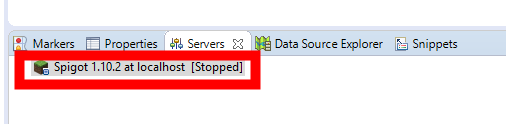
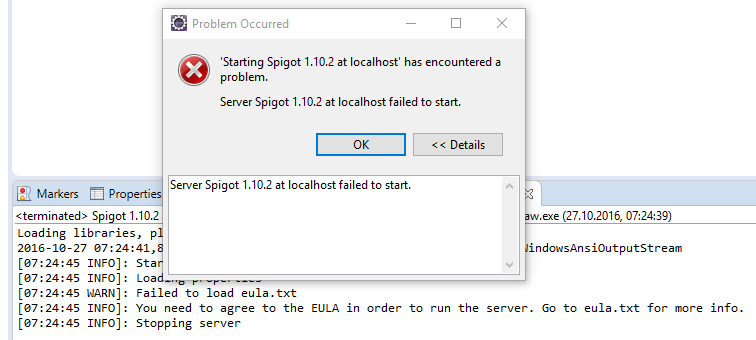
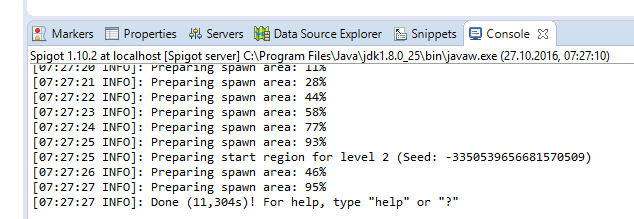
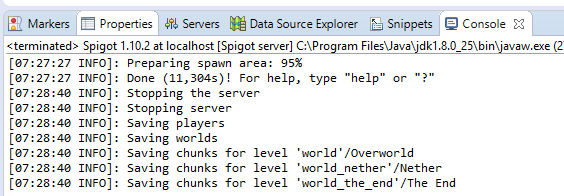
The spigot plugins were tested on Eclipse neon.

1. You can download eclipse neon for java EE development at <http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/neon1a>
2. After first start go to “help” > “Install new software”. Select the “Add” button on the right.  
   
3. You can now enter a new update site with following values:  
    Name: Minecraft Spigot  
    Location: <http://minecraft.xworlds.eu/eclipse>  
   
4. Select the site in drop down and select the minecraft spigot feature to install.  
   
5. Follow the wizard steps and let eclipse install the plugins.

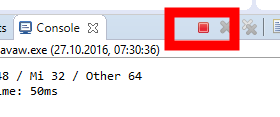
# Setup spigot servers

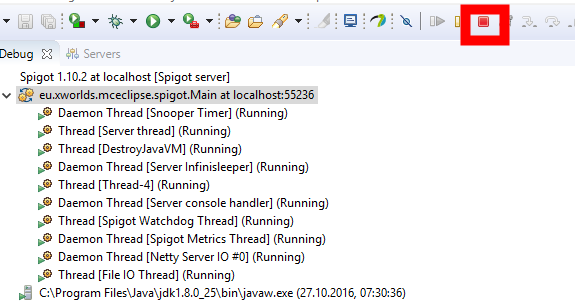
Eclipse webtools divides servers and runtimes. A runtime is a location of your spigot server on your local hdd. A server holds all configurations, plugins and everything you need to start spigot.

If you are a developer of two plugins you can use the same runtime within two servers. It is one server for each plugin. Or you can use only one server for both.

1. To create a server, go to the server view on the bottom of the java ee perspective:  
   
2. Click on the hyperlink. In the “new Server” wizard select the spigot server under category “Basic”.  
   
3. You will be forces to create a new runtime. Select either your existing installation or download it from our repository.  
   
4. On the next wizard page it will ask you which resources to add. We do not yet have any spigot plugin in our workspace. Simply ignore and click on “finish”.
5. Now you have a working spigot server. Right click or use the buttons on the right to start/debug it.  
     
   
6. As you see it will fail. However as an experienced minecraft/ spigot user you will know what to do. Have a look at your workspace and open the “Servers” projects. After doing a refresh (F5) you will see the eula.txt. Edit it, accept the eula and restart your server again.  
   
7. To shut down the server simply right click in the servers view or type “stop” in the console. You may use your minecraft to connect to the server and type “/stop” in the chat as well.  
   

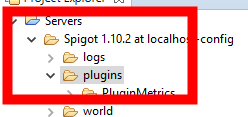
NOTICE: In the console view there is a red terminate button. Do not use it. It will shut down the java VM but does not save world data etc. You may use it if the spigot server crashed and does not respond. You should always prefer stopping the server by “stop” command or via “Servers” view. The second non-preferred terminate button is inside debug view.



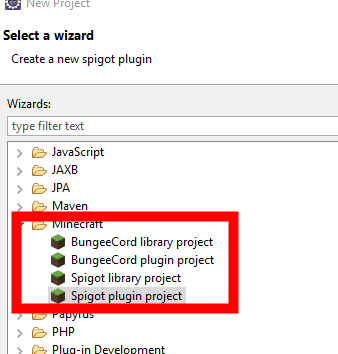
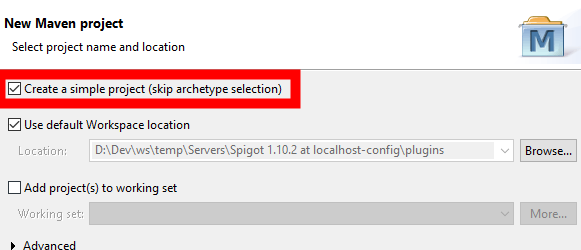
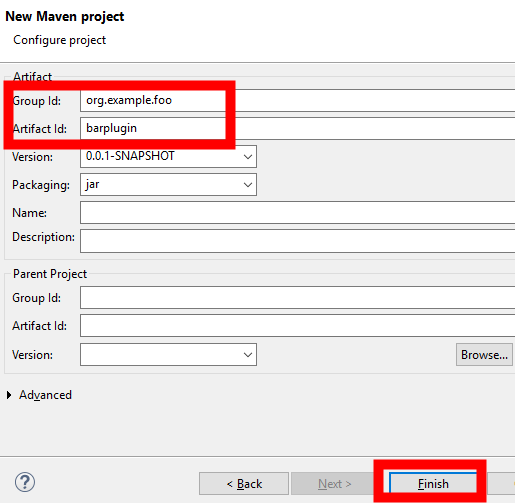
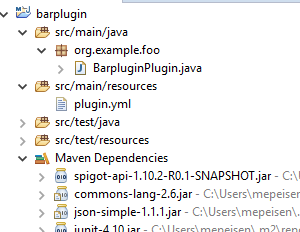
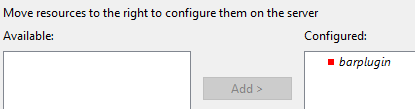
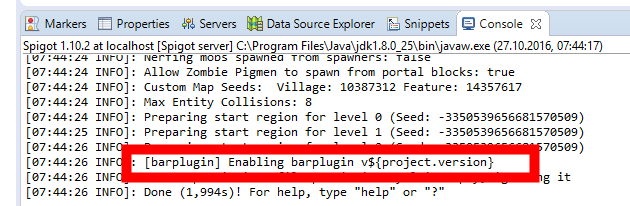


# Installing third party plugins

It is fairly simple. Stop the server. Download the plugin jar files. Copy them. Open the Servers project and put them into the plugins folder.

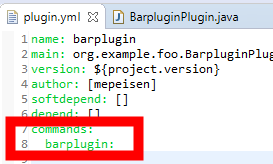


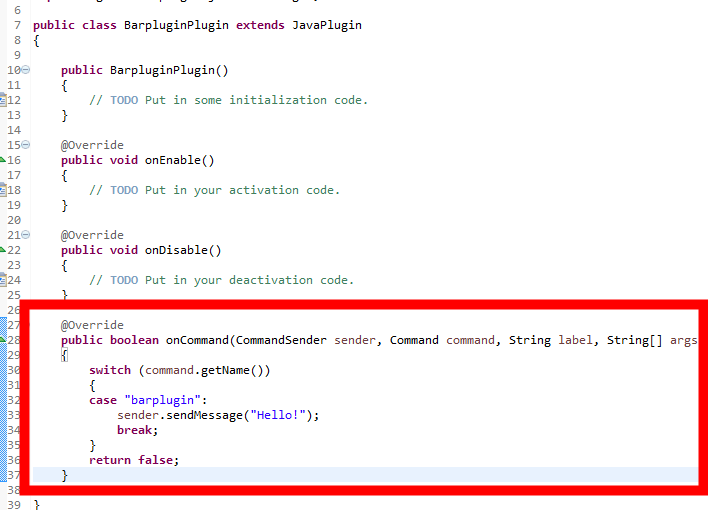
# First plugin project

1. Right click in the “project explorer” view and select “New” > “Project”. You will find a “Minecraft” category and the “Spigot plugin project” type.  
   
2. Select the “Create a simple project” checkbox.  
   
3. Type a maven group id (typically a reversed internet domain you own). Type a maven artifact id and select “Finish”.  
   
4. You now have a new spigot plugin within your workspace.  
   
5. Go back to your server and right click on it. Select the “Add and remove” command. You will now see the barplugin being available for your spigot server. Move it to the right.  
   
6. Restart the server and watch the console.  
   

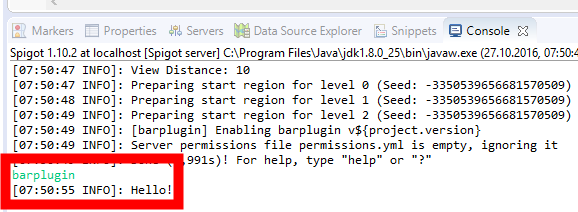
# Hot code replacement.

Eclipse supports a feature called hot code replacement. That’s useful during development. It allows you to change code without restarting the whole server. We will create a small useful example. Add a command by editing the plugin.yml and the BarpluginPlugin class.

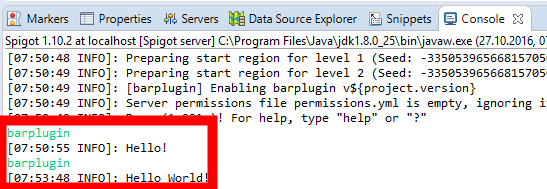




Now start the server and check your code.

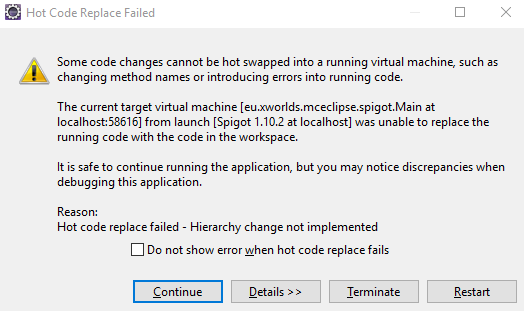


It is working. But let us now use the hot code replacement. WIHTOUT restarting the server change the message in your code to “Hello World!”. Save the java file and check the command again.



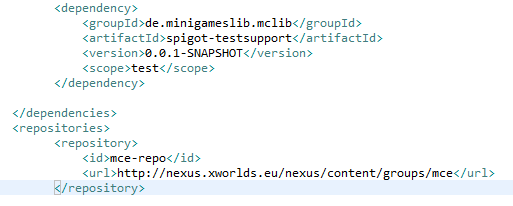
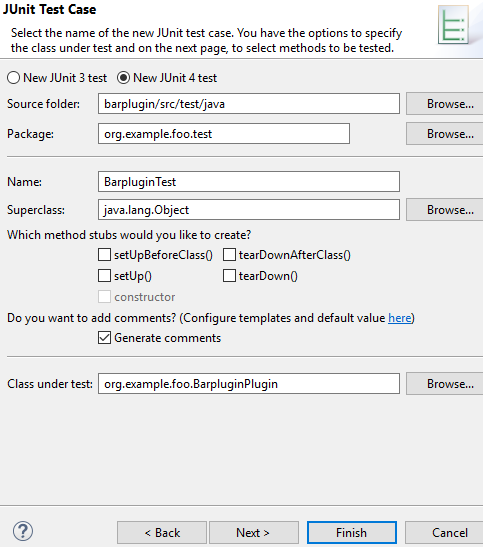
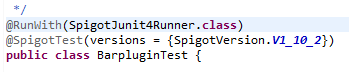
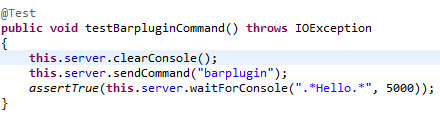
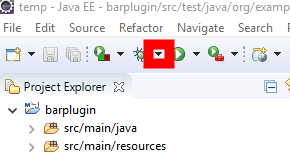
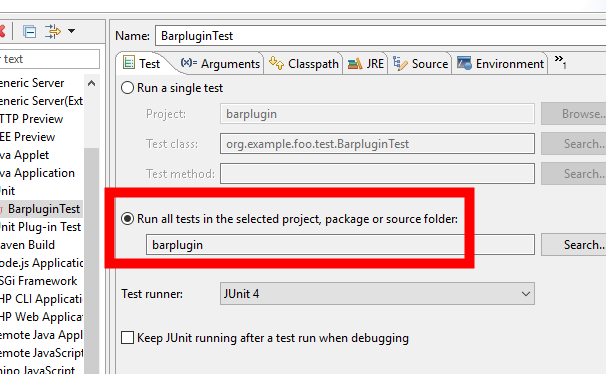
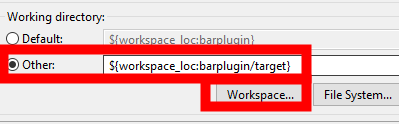
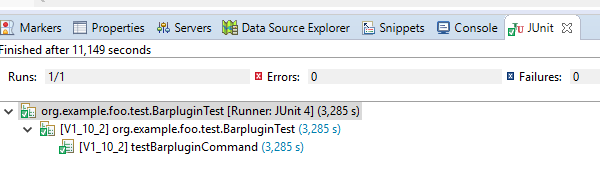
The message changed. Without building jar files, without redeploying to server and without restarting the server.

The hot code replacement is limited. Changing class structures often result in warnings to restart the server. For example let your plugin implement the bukkit listener interface. You will get the following warning.



Your spigot server is now out of sync. Restart it to get test the new code.

# Junit support

1. Edit your pom.xml and add the following elements:  
   
2. Right click on “src/test/java” and create a JUnit class for testing (“New “ > “Other” > “Java” > “Junit” > “Junit Test Case”).  
   
3. Add the annotations “RunsWith” and “SpigotTest” at class level.  
   
4. To access our spigot server during junit tests we need to add a variable annotated with “SpigotInject”.  
   
5. We are now ready to perform our test. As we already have a command we will simply test it:  
   
6. We now create a new debug profile via toolbar. Click the small arrow right from the bug and select “debug configurations…”.  
   
7. Right click on “Junit” and select “New”. Change the properties to run all tests from your project.  
   
8. Change the arguments to start in directory “target”.  
   
9. Start the test by selecting the debug button.  
   

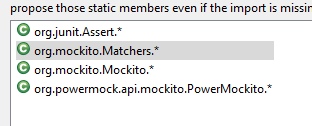
***Hint***: We do not really need our own debug configuration for the whole project and with folder “target”.

You can simply run your tests by right clicking the class and select “Run as Junit test”.

However eclipse always chooses to use the projects root directory. You may be messed up with log files, configuration files etc.

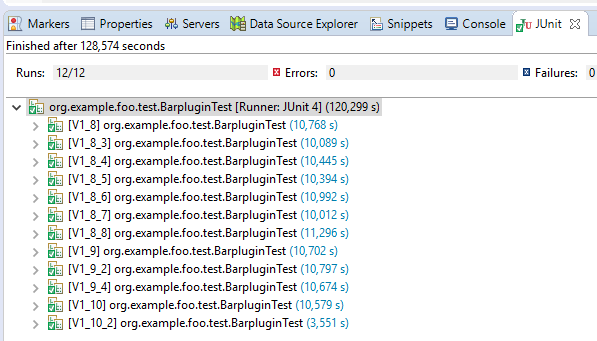
Typically in scm (SVN or GIT) you will never commit the target directory. Choosing the target directory as test working directory will automatically hide all the test files from scm.

***Hint:*** We use a static import “assertTrue” in our sample code. Eclipse is able to automatically import statics but the feature is somehow hard to find. Go the “Window” > “Preferences” > “Java” > “Editor” > “Content Assist” > “Favorites”. Add the following by selecting the “New Type” button:



Now you can type “assertTrue” in the java code editor and press “Ctrl+Space” for content assist and creating a static import.

***Hint:*** As you see the SpigotTest annotation controls the servers you want to test. You can add more servers or even simply set “all” to true. See what happens if we test all the versions:



# Dependencies

First of all some theory of how maven manages dependencies. The most important thing is the dependency scope.

You can read details on the following web page: <https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>

We are focusing on three scopes: compile, provided, runtime.

The compile scope is the default. We use it for all non-minecraft jar files. For example you can add apaches commons-mail. Adding it with compile scope (default) will cause the eclipse plugin to add it to the plugin classpath.

The provided classpath is more special. We use it for all plugins and plugin apis that are minecraft related. Provided classpath causes maven to add it during compile time (you can use it) but not to the classpath itself. We use this scope for spigot-api itself too. Evenrything that is not part or your plugin will at least use “provided” or “runtime”.

The runtime classpath is only a hint for maven that this dependency must be used while running the server.

## How does eclipse minecraft plugin use the scopes?

It is really simple:

1. All external jar files are using no scope/ “compile” scope.
2. All Plugin APIs or Plugin Jars you need during compile will use “provided” scope.
3. All Plugin Jars implementing a plugin API will use “runtime” scope.

Some simple example:

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-email</artifactId>

<version>1.4</version>

</dependency>

<dependency>

<groupId>de.minigameslib.mclib</groupId>

<artifactId>mclib-api</artifactId>

<version>0.0.1-SNAPSHOT</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>de.minigameslib.mclib</groupId>

<artifactId>mclib</artifactId>

<version>0.0.1-SNAPSHOT</version>

<scope>runtime</scope>

</dependency>

What do we do here?

First we add the apache mailing library. It is a compile library and we like to embed it into our plugin.

As second dependency we add minigames mclib. It uses provided scope. We will see it during compile time so we can develop against it. But we do not want it to be part of our plugin. Instead the api will be shipped with mclib jar and the user of our plugin will have to install both, our plugin and mclib.

At third position we give maven and the eclipse plugin some hint. We require the presence of “mclib” during runtme. We do not compile against it but we need it to work.

Let us view the result:



The eclipse plugin will automatically detect the dependencies. It will see that you use some other plugin in the dependencies and it will automatically install both plugins in your server.

## Some word about maven-shade plugin

For spigot it is recommended to distribute a single jar file to your users containing everything that is relevant.

Given our previous example we want maven to build a jar file containing everything but mclib.

Add the following section to your pom.xml:

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<version>2.4.3</version>

<executions>

<execution>

<phase>package</phase>

<goals><goal>shade</goal></goals>

<configuration>

<artifactSet>

<excludes>

<exclude>de.minigameslib.mclib:\*</exclude>

</excludes>

</artifactSet>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

If you now invoke the maven build it will create a single jar file containing all the needed dependencies. But it excludes the mclib plugin.