

minigameslib.de

MCLIB

Minigames Core Library – Eclipse plugins and Junit-Support

1 Table of contents

2	Preamble	3
3	License and usage scenarios	4
3.1	License	4
3.2	Productional Server (Spigot)	4
3.3	Developing (Eclipse-Setup)	4
3.4	Client (Forge).....	4
3.5	Alternative client mods.....	5
4	Development setup.....	6
4.1	Eclipse perquisites	6
4.2	Eclipse setup.....	6
4.3	Setup spigot servers	7
4.4	My first spigot plugin.....	10
4.5	Hot code replacement.	13
4.6	Advanced spigot topics.....	16
4.6.1	Junit support.....	16
4.6.2	Installing third party plugins.....	20
4.6.3	Server administration.....	21
4.7	Server project setup scenarios.....	21
4.7.1	Dependency background	21
4.7.2	How does eclipse minecraft plugin use the scopes?	21
4.7.3	Some word about maven-shade plugin	22
4.7.4	NMS classes.....	23
4.7.5	Multi-project setup for NMS	23
4.7.6	Plugin API vs. Implementation.....	24
4.8	Forge client	24
5	Developing against MCLIB (Spigot server).....	25
5.1	Dependencies.....	25
5.2	onEnable	25
5.3	onDisable	25
5.4	Version detection	25
5.5	Enumerations and their benefits.....	26
5.5.1	Identifying plugin from enumeration.....	27
5.5.2	Building your own enumerations	28
5.5.3	Child enumerations.....	29

5.6	Multi-language and customizable messages	29
5.6.1	Servers default locale.....	30
5.6.2	Users preferred locale.....	30
5.6.3	Declaring messages.....	31
5.6.4	Using the message	31
5.6.5	Message arguments.....	32
5.6.6	Multi line messages	32
5.6.7	Language packs.....	32
5.6.8	Exceptions with messages.....	32
5.6.9	messages.yml and Message comments	32
5.6.10	Predefined messages	32
5.7	Command handling	32
5.8	Context sensitive execution	32
5.9	Configuration Framework and DataFragments.....	32
5.10	Storages	33
5.11	Extension handling	33
5.12	Simple GUI.....	33
5.13	Permissions	33
5.14	Objects framework.....	33
5.15	Event framework.....	33
6	Developing against MCLIB (Forge clients)	34
7	BungeeCord-Clusters and Client/Server.....	35
7.1	Communication channels	35
8	Smart GUI	36
9	Advanced testing.....	37
9.1	spigot-testsupport	37
9.2	fakeplayer	37

2 Preamble

MinigamesLib was originally created by InstanceLabs. It was meant as a library to support some of the Minecraft Minigames he created for Bukkit. You can view his profile at github:

<https://github.com/instance01>

However, he decided to stop developing and supporting his work. As users, we felt sad about this and decided to ask him for overtaking the project. He agreed and so MysticCity became new project owner. I became main developer and contributor after some other guys developed only for some weeks but left the project. ***My nick name is mepeisen.***

While doing bug fixes and migration to newer Minecraft versions we decided to make a huge redesign and rework. We continued the work for version 2.0 of MinigamesLib. After doing some of the work we came up with the idea to create a common core library. It is some kind of managing different aspects of Minigames Lib that are not only related to Minigames. Did you ever ask yourself how to manage “game components”, “areas” or “multi lingual systems”?

The answer is the MCLIB (**M**inigames **C**ore Library).

Special thanks to all the users supporting the work. Every single review and feature wish was respected and hopefully will be part of the minigames system in version 2.0

3 License and usage scenarios

The library is built of multiple parts. We have client parts, support for testing and server libraries. You should read the following chapters carefully and decide which szenario fits your needs.

3.1 License

MCLIB contains a public part. It can always be used with GPLv3 in mind.

To fund the project, we decided to provide a commercial license. This is required because the commercial license may give more advanced warranty but does some restrictions to not break the warranty. The second cause are other commercial projects we might support. They are not happy with GPLs copy-left.

MCLIB and MinigamesLib itself will support some premium options that are covered by the commercial license too. More details will be part of future versions.

You are always allowed to fork the project under terms of GPL, specially while you always publish the source code. If you do not follow this license you are not allowed to modify the code in any way. But it is cleverer to ask us for feature requests directly or to contribute your work back to us. Contributing will guarantee that your feature is staying in the core lib and will be migrated to newer Minecraft versions from us.

3.2 Productional Server (Spigot)

It is fairly simple. The MCLIB is a plugin itself. Simply download the server jar from your favorite location and store the jar file into the plugins folder of your spigot server.

You need no special setup to work with MCLIB. It runs “out-of-the-box”. Some of the features may require additional setup to perform in a way you want them but this is mentioned in a detailed explanation of the features later on.

3.3 Developing (Eclipse-Setup)

We always wanted to have a smart IDE support. I decided to use eclipse for developing the minigames.

If you want to use MCLIB for developing within eclipse, we provide some detailed explanation in an extra chapter later on. There is some work to be done before using MCLIB in “eclipse way”.

We support both, client and server development in eclipse.

3.4 Client (Forge)

MCLIB has its own client mod. It is optional. You may use it for your work or simply ignore it. The MCLIB will always work without the forge mod. But there are some nice features we get with our forge mod.

One of the key features of the forge mod is called “Smart GUI”. It displays some nice GUI frontend with advanced widgets. The feature is somehow required to make administration of Minigames more intuitive. Administration via text chat console is complicated, error-prone and annoying. The GUI itself you can control via server is very limited as you know (anvil gui, containers with pre-defined icon sets etc.)

The premium part of our plugin will require the forge mod for some of the aspects.

3.5 Alternative client mods

We know that downloading and installing additional software on your PC and even on PC of your users is critical because hackers might introduce Trojan and some kind of virus. Some users do not like it and as long as MInigamesLib version 2 is new, users may be prejudiced.

MCLIB may support other clients in future versions as an alternative so that users will trust the MinigamesLib. However, the communication protocol itself and the MCLIB client mod will stay open sourced to increase trust.

4 Development setup

MCLIB includes eclipse plugins for supporting development with Minecraft and Spigot.

In this chapter I will show you how to install and use the eclipse plugins to develop your Minecraft plugins, server side and client side.

You will step per step learn how to create a simple hello world example.

4.1 Eclipse perquisites

We use the latest eclipse. The plugins were tested in eclipse NEON. You should use Eclipse Edition for Java EE developers. You can use the installer from <http://www.eclipse.org/downloads/>

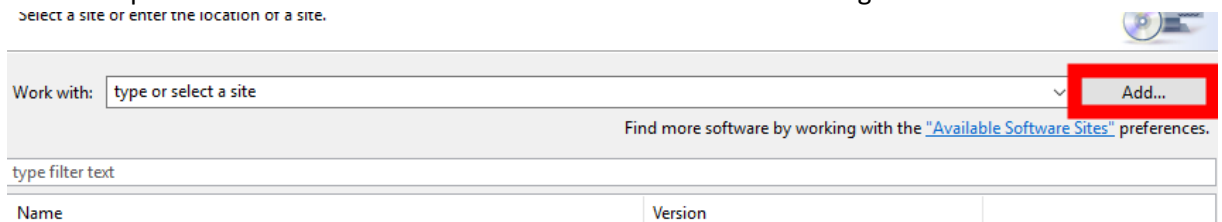
We use JDK 8 for development.

After installing eclipse, you need some more plugins and requirements. You will find them either in eclipse marketplace (see help menu) or by selecting “installing new software” from help menu and by using the eclipse NEON update site.

- “Buildship Gradle-Plugins”: used for client development with FORGE and available from marketplace.
- “Dynamic Languages Toolkit”: Available from NEON update site
- “ECLemma Java Code Coverage”: Optional for Junit Code coverage, available from marketplace
- “Java Web Developer Tools”: Available from NEON update site
- “Web developer tools”: Available from NEON update site
- “YEdit”: Available from marketplace

4.2 Eclipse setup

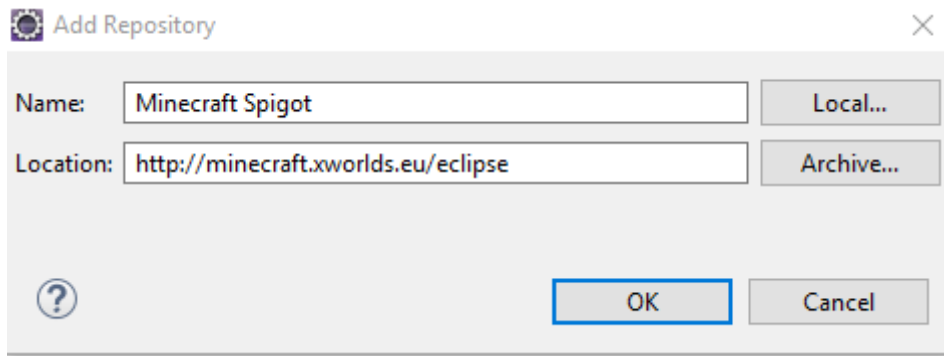
- 1) Go to “help” > “Install new software”. Select the “Add” button on the right.



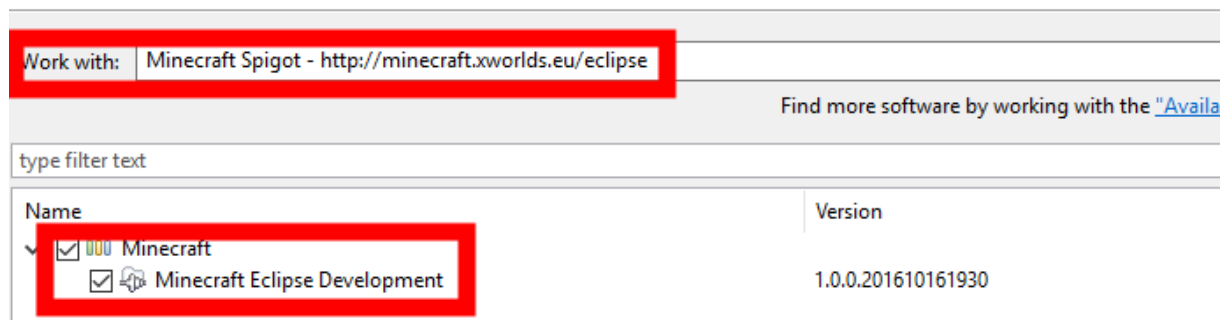
- 2) You can now enter a new update site with following values:

Name: Minecraft Spigot

Location: <http://minecraft.xworlds.eu/eclipse>



- 3) Select the site in drop down and select the minecraft spigot feature to install.
Check the items that you wish to install.



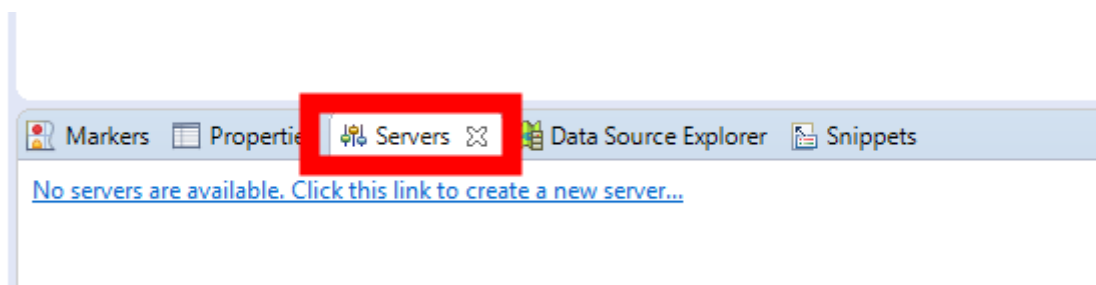
- 4) Follow the wizard steps and let eclipse install the plugins.

4.3 Setup spigot servers

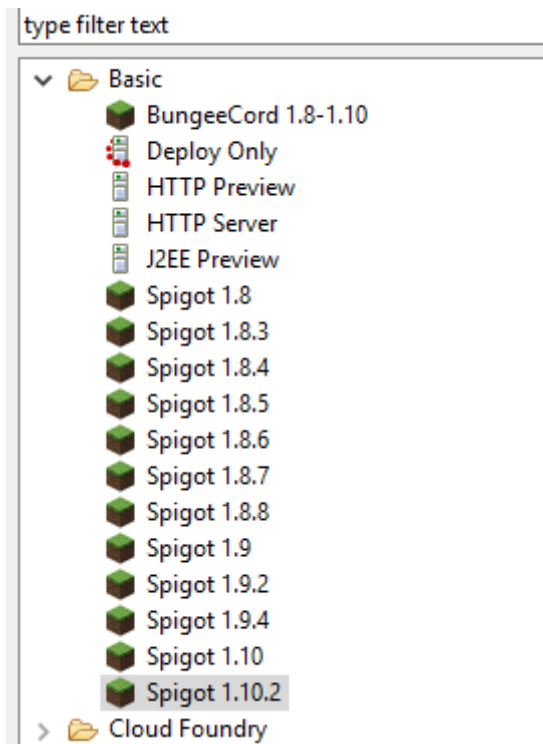
Some background information: Eclipse webtools divides servers and runtimes. A runtime is a location of your spigot server on your local hdd. A server holds all configurations, plugins and everything you need to start spigot.

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. It is your choice.

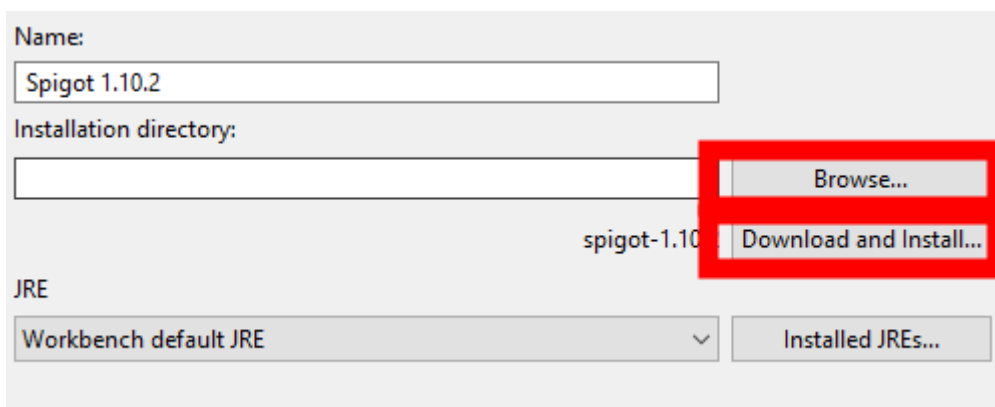
- 1) To create a server, go to the server view on the bottom of the java ee perspective or open it via "Window" -> "Show view":



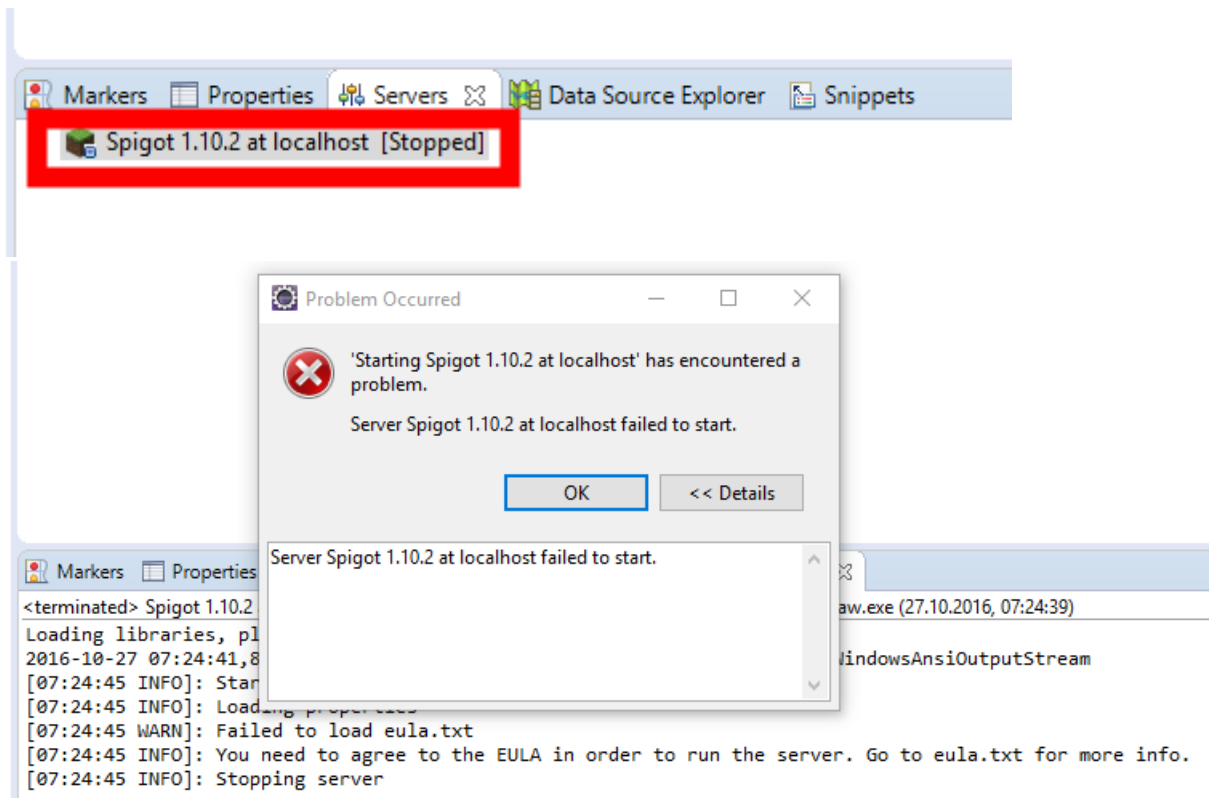
- 2) Click on the hyperlink. In the "new Server" wizard select the spigot server under category "Basic".



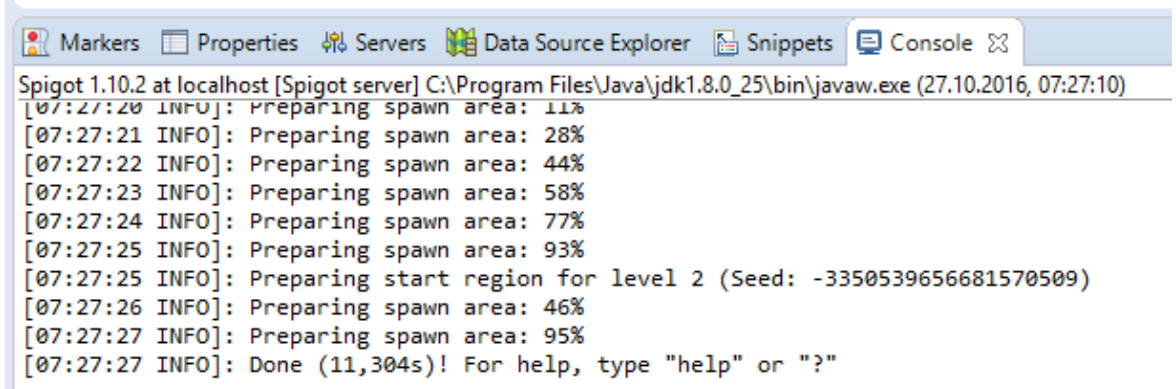
- 3) You will be forced to create a new runtime. Select either your existing installation or download it from our repository. The download versions are created using BuildTools. There is no secret within them.



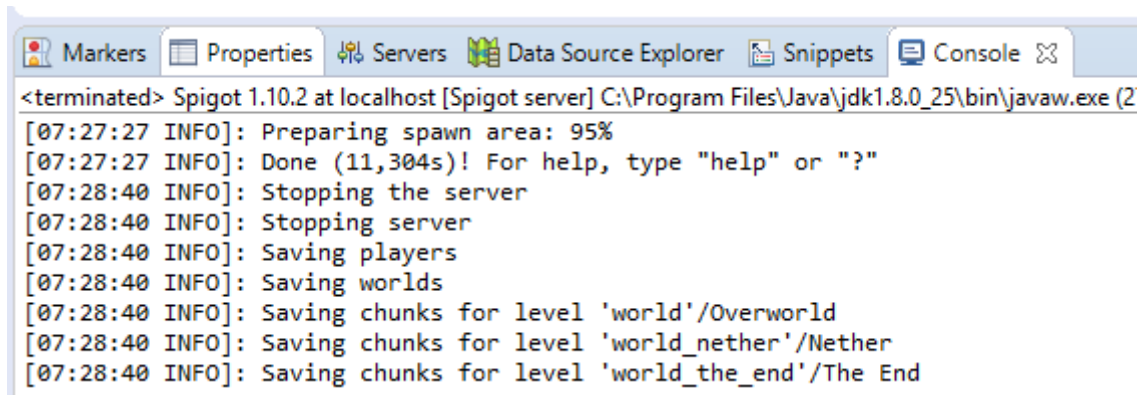
- 4) On the next wizard page, it will ask you which resources to add. The list is empty because we do not yet have any spigot plugin in our workspace. Simply ignore and click on “finish”.
- 5) Now you have a working spigot server. Right click or use the buttons on the right to start/debug it.



- 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. Now it will start without any error.



- 7) To shut down the server simply right click in the server 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.

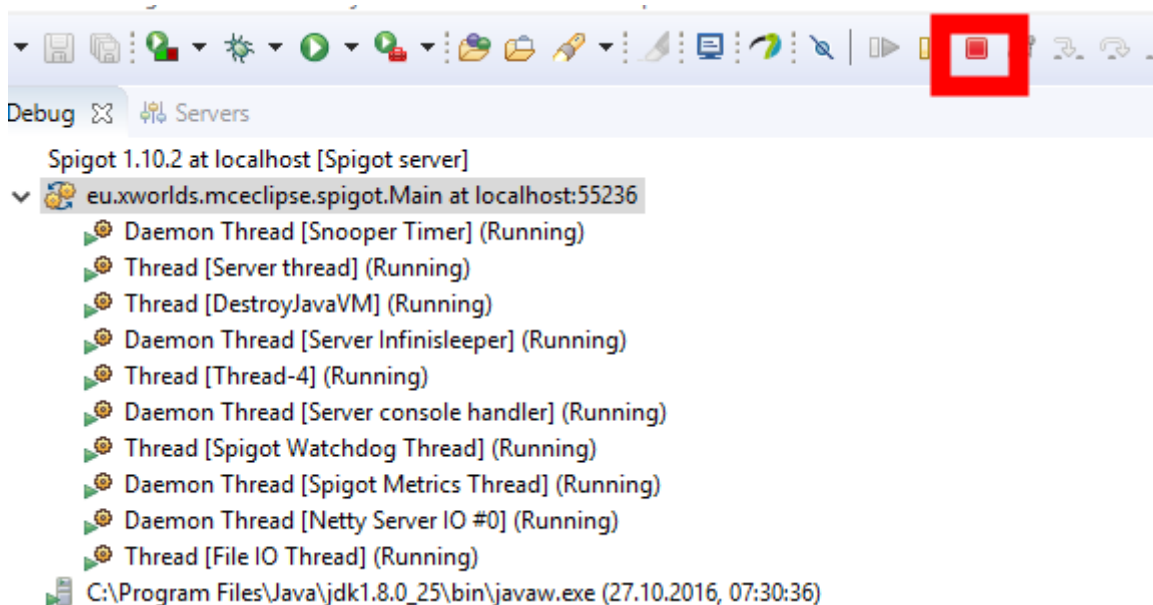
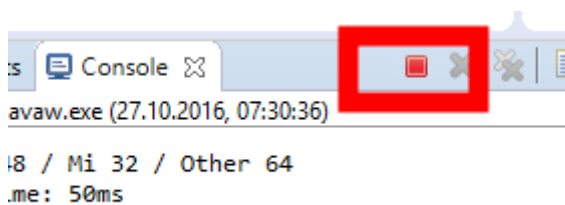


```

<terminated> Spigot 1.10.2 at localhost [Spigot server] C:\Program Files\Java\jdk1.8.0_25\bin\javaw.exe (2
[07:27:27 INFO]: Preparing spawn area: 95%
[07:27:27 INFO]: Done (11,304s)! For help, type "help" or "?"
[07:28:40 INFO]: Stopping the server
[07:28:40 INFO]: Stopping server
[07:28:40 INFO]: Saving players
[07:28:40 INFO]: Saving worlds
[07:28:40 INFO]: Saving chunks for level 'world'/Overworld
[07:28:40 INFO]: Saving chunks for level 'world_nether'/Nether
[07:28:40 INFO]: Saving chunks for level 'world_the_end'/The End


```

NOTICE: In the console view, there is a red terminate button. Do not use it. It will shut down the java VM immediately 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 “Server” view. The second non-preferred terminate button is inside debug view. Do not use the two buttons in the following screenshots.



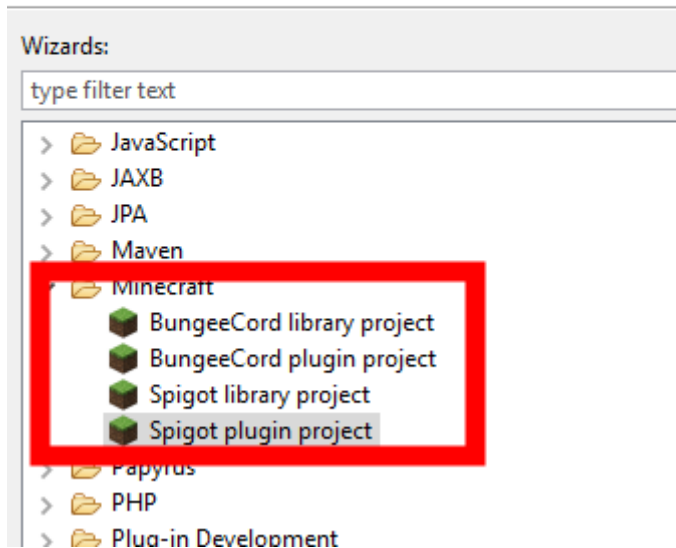
4.4 My first spigot plugin

- 1) Right click in the “project explorer” view and select “New” > “Project”. You will find a “Minecraft” category and the “Spigot plugin project” type.

 New Project

Select a wizard

Create a new spigot plugin



- 2) Select the “Create a simple project” checkbox.

New Maven project

Select project name and location



☒ Create a simple project (skip archetype selection)

☒ Use default Workspace location

Location:

☐ Add project(s) to working set

Working set:

► Advanced

- 3) Type a maven group id (typically a reversed internet domain you own). Type a maven artifact id and select “Finish”.

New Maven project

Configure project

The screenshot shows the 'New Maven project' dialog box with the following fields:

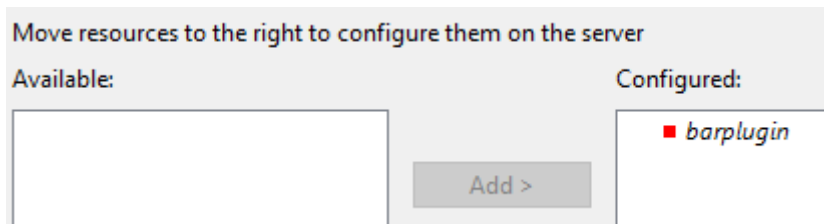
- Artifact**
 - Group Id:
 - Artifact Id:
 - Version:
 - Packaging:
 - Name:
 - Description:
- Parent Project**
 - Group Id:
 - Artifact Id:
 - Version:
 -
- Advanced** (collapsed)

At the bottom, there are three buttons: , , and . The 'Finish' button is highlighted with a red box.

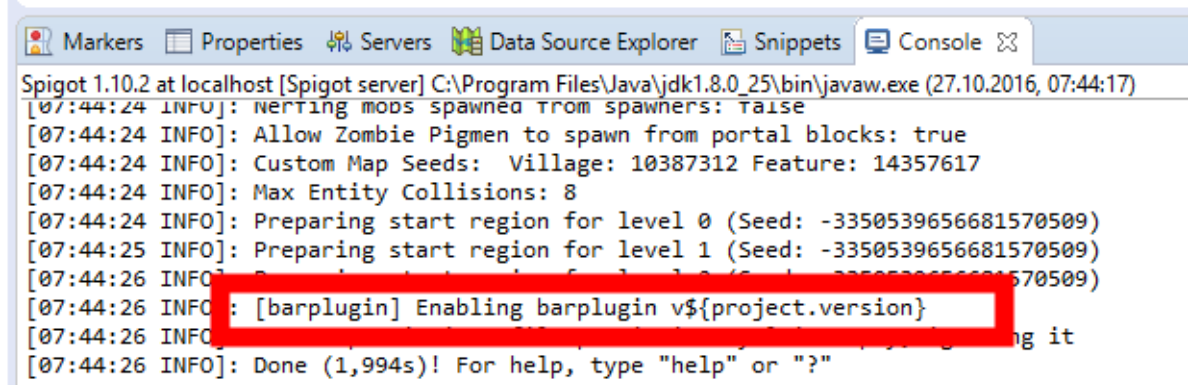
- 4) You now have a new spigot plugin within your workspace.

- ✓ barplugin
 - ✓ src/main/java
 - ✓ org.example.foo
 - > BarpluginPlugin.java
 - ✓ src/main/resources
 - plugin.yml
 - > src/test/java
 - > src/test/resources
 - ✓ Maven Dependencies
 - > spigot-api-1.10.2-R0.1-SNAPSHOT.jar - C:\
 - > commons-lang-2.6.jar - C:\Users\mepeiser
 - > json-simple-1.1.1.jar - C:\Users\mepeisen\
 - > junit-4.10.jar - C:\Users\mepeisen\m2\ren

- 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.

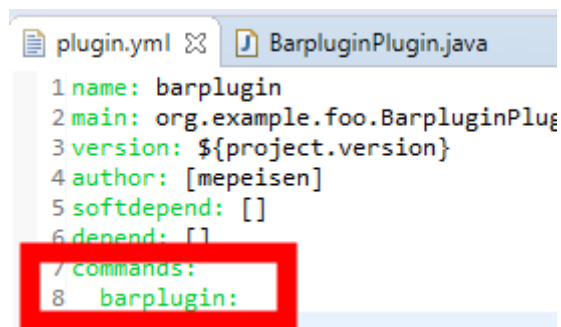


- 6) Restart the server and watch the console.



4.5 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.

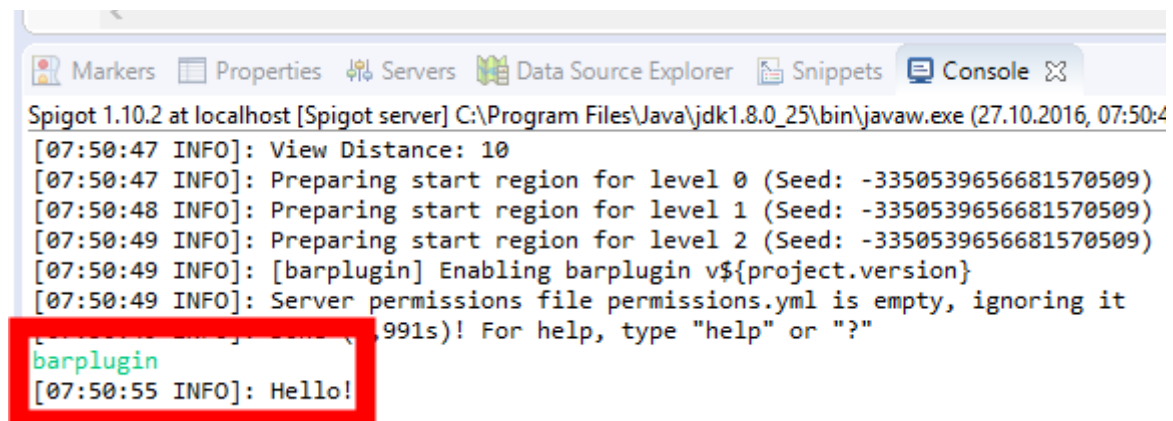


```

6
7 public class BarpluginPlugin extends JavaPlugin
8 {
9
10 public BarpluginPlugin()
11 {
12     // TODO Put in some initialization code.
13 }
14
15 @Override
16 public void onEnable()
17 {
18     // TODO Put in your activation code.
19 }
20
21 @Override
22 public void onDisable()
23 {
24     // TODO Put in your deactivation code.
25 }
26
27 @Override
28 public boolean onCommand(CommandSender sender, Command command, String label, String[] args)
29 {
30     switch (command.getName())
31     {
32     case "barplugin":
33         sender.sendMessage("Hello!");
34         break;
35     }
36     return false;
37 }
38
39 }

```

Now start the server and check your code.



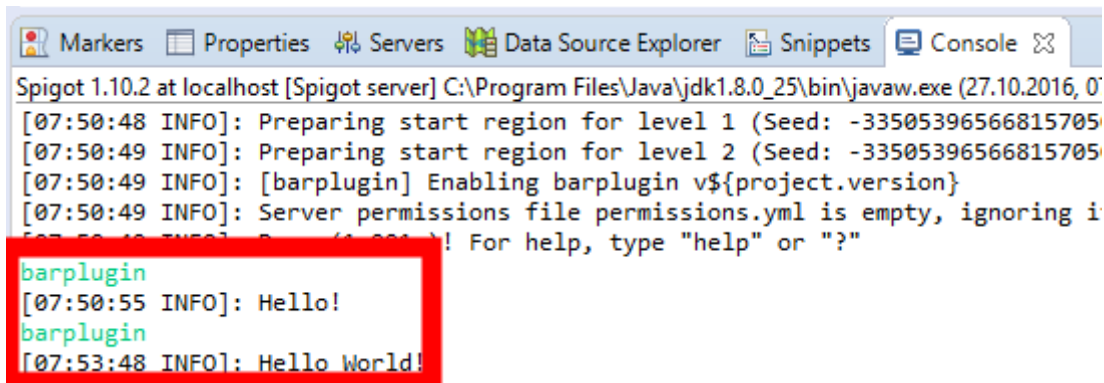
The screenshot shows the Spigot server console with the following output:

```

Spigot 1.10.2 at localhost [Spigot server] C:\Program Files\Java\jdk1.8.0_25\bin\javaw.exe (27.10.2016, 07:50:4
[07:50:47 INFO]: View Distance: 10
[07:50:47 INFO]: Preparing start region for level 0 (Seed: -3350539656681570509)
[07:50:48 INFO]: Preparing start region for level 1 (Seed: -3350539656681570509)
[07:50:49 INFO]: Preparing start region for level 2 (Seed: -3350539656681570509)
[07:50:49 INFO]: [barplugin] Enabling barplugin v${project.version}
[07:50:49 INFO]: Server permissions file permissions.yml is empty, ignoring it
[07:50:50 INFO]: [barplugin] (1.991s)! For help, type "help" or "?"
barplugin
[07:50:55 INFO]: Hello!

```

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.



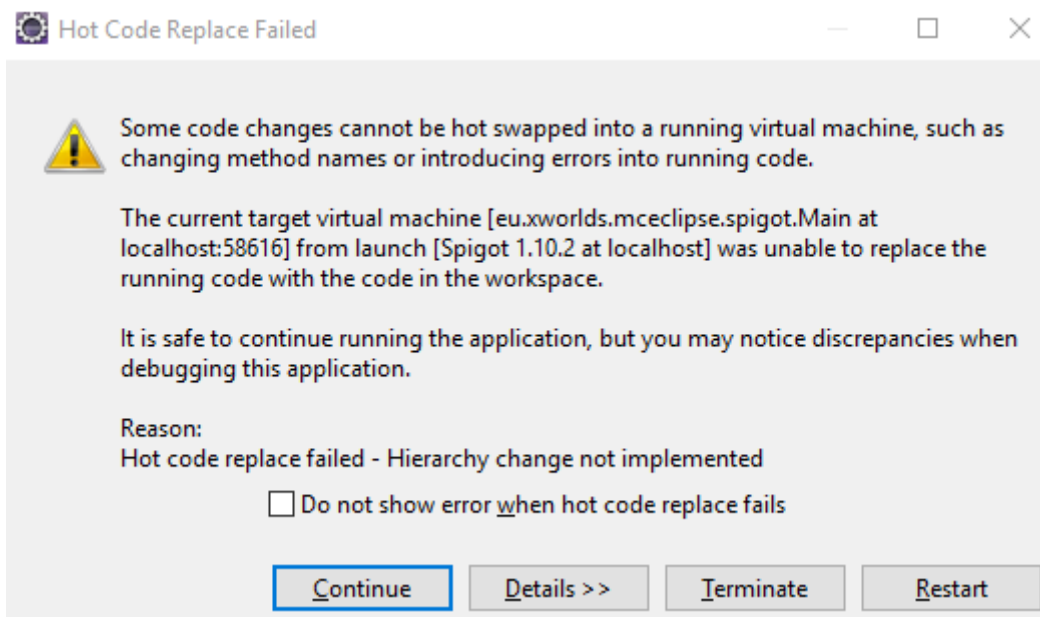
```

Markers Properties Servers Data Source Explorer Snippets Console
Spigot 1.10.2 at localhost [Spigot server] C:\Program Files\Java\jdk1.8.0_25\bin\javaw.exe (27.10.2016, 0'
[07:50:48 INFO]: Preparing start region for level 1 (Seed: -33505396566815705)
[07:50:49 INFO]: Preparing start region for level 2 (Seed: -33505396566815705)
[07:50:49 INFO]: [barplugin] Enabling barplugin v${project.version}
[07:50:49 INFO]: Server permissions file permissions.yml is empty, ignoring it
[07:50:49 INFO]: [barplugin] Hello!
[07:53:48 INFO]: [barplugin] Hello World!

```

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.

4.6 Advanced spigot topics

4.6.1 Junit support

- 1) Edit your pom.xml and add the following elements to dependencies and repositories section:

```
<dependency>
  <groupId>de.minigameslib.mclib</groupId>
  <artifactId>spigot-testsupport</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <scope>test</scope>
</dependency>

</dependencies>
<repositories>
  <repository>
    <id>mce-repo</id>
    <url>http://nexus.xworlds.eu/nexus/content/groups/mce</url>
  </repository>
```

- 2) Right click on “src/test/java” and create a JUnit class for testing (“New “ > “Other” > “Java” > “JUnit” > “JUnit Test Case”).

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.



☐ New JUnit 3 test ☒ New JUnit 4 test

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

- ☐ setUpBeforeClass() ☐ tearDownAfterClass()
☐ setUp() ☐ tearDown()
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

☒ Generate comments

Class under test:



< Back

Next >

Finish

Cancel

- 3) Add the annotations “RunWith” and “SpigotTest” at class level.

```

*/
@RunWith(SpigotJUnit4Runner.class)
@SpigotTest(versions = {SpigotVersion.V1_10_2})
public class BarpluginTest {

```

- 4) To access our spigot server during junit tests we need to add a variable annotated with “SpigotInject”.

```

@SpigotInject
private SpigotServer server;

```

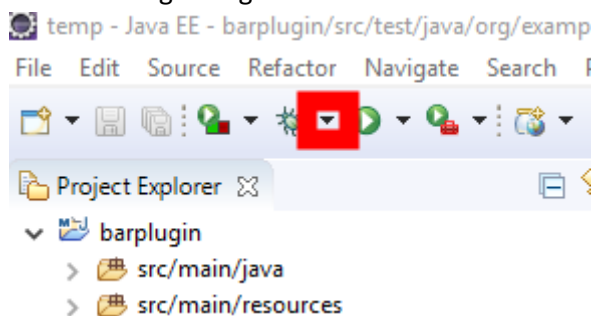
- 5) We are now ready to perform our test. As we already have a command we will simply test it:

```

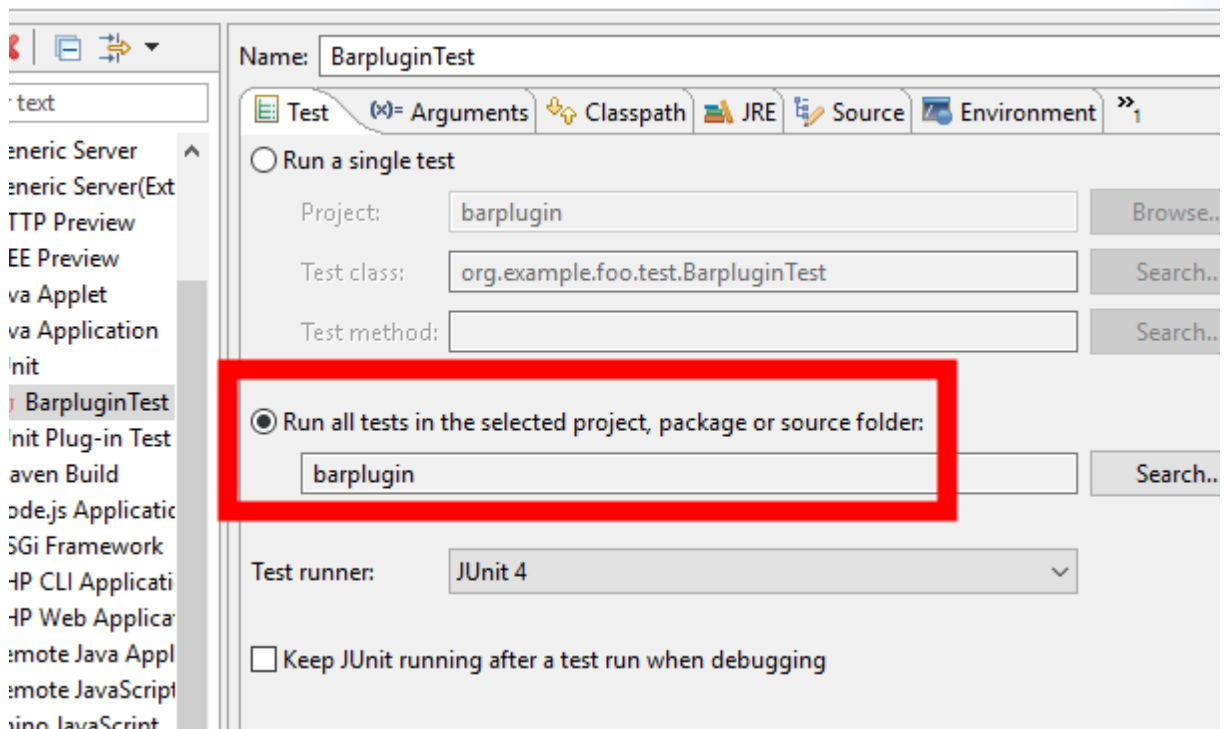
@Test
public void testBarpluginCommand() throws IOException
{
    this.server.clearConsole();
    this.server.sendCommand("barplugin");
    assertTrue(this.server.waitForConsole(".*Hello.*", 5000));
}

```

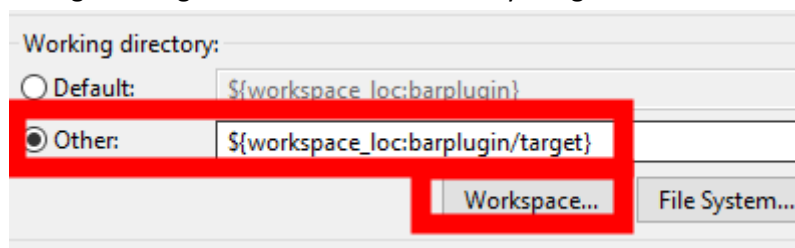
- 6) We now create a new debug profile via toolbar. Click the small arrow right from the bug and select “debug configurations...”.



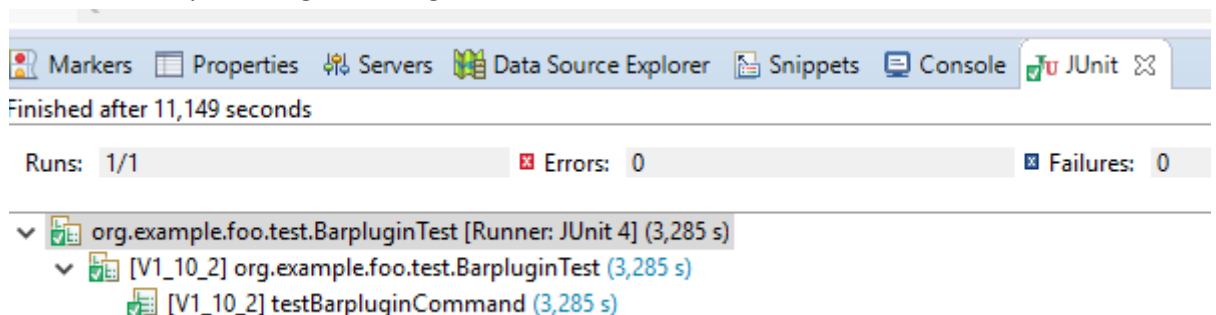
- 7) Right click on “JUnit” and select “New”. Change the properties to run all tests from your project.



- 8) Change the arguments to start in directory “target”.



- 9) Start the test by selecting the debug button.



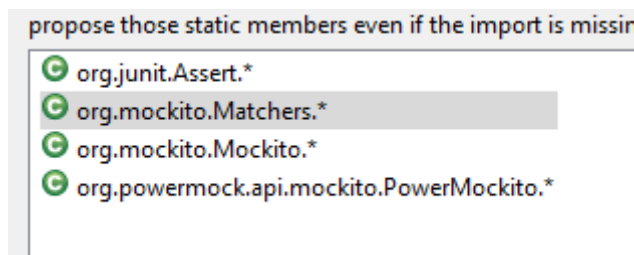
Hint: We do not really need our own debug configuration for the whole project and with folder “target”.

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.

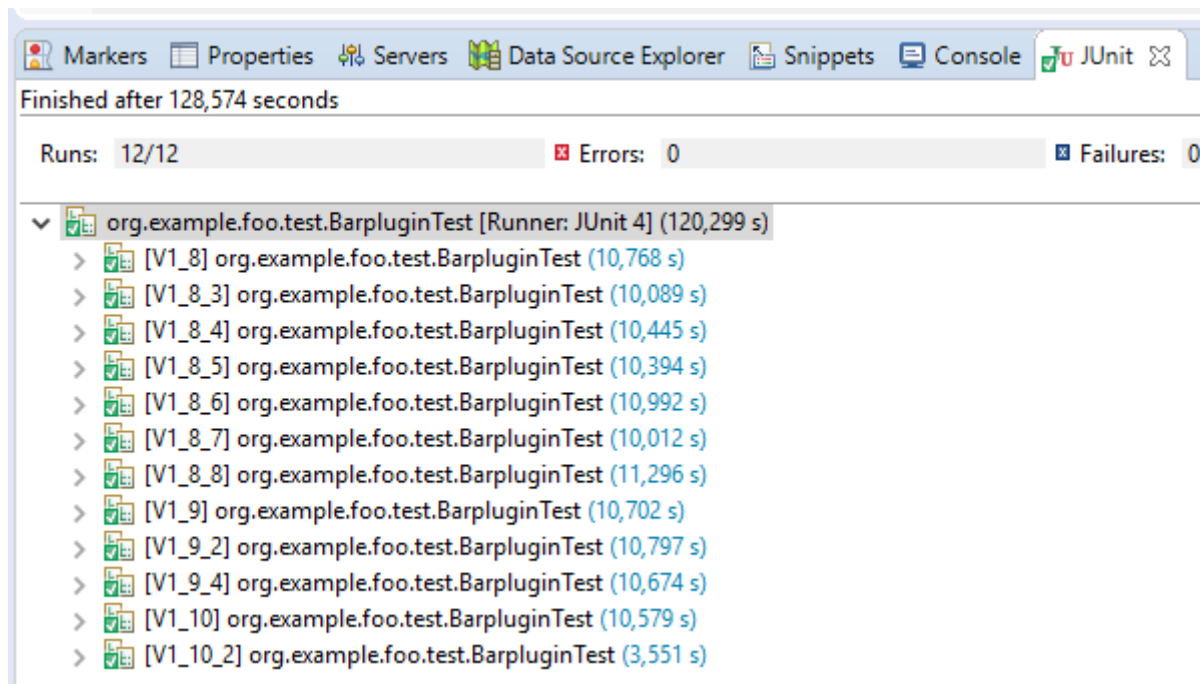
Typically, in scm (SVN or GIT) you will never commit the target directory because it is known as maven build directory. Choosing the target directory as test working directory will automatically hide all the test files from scm.

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:



4.6.2 Installing third party plugins

You might try to install third party plugins directly by adding them into the plugins folder. Like you do in productional environments. The plugin does not support it this way.

But there is an alternative. You may add them to your own plugin as dependency or you may create your own java project.

4.6.2.1 *Java project for dependencies*

- 1) Create a simple standard java project.
- 2) In project properties go to “Project facets” and click on the link “Convert to faceted project”.
- 3) After the project was converted activate the “Spigot library” facet.
- 4) Now you can add this library project to the server.
- 5) Every jar file you add to the classpath of this project will now be loaded by eclipse plugins. The eclipse plugin automatically detect that you added a plugin jar file and performs the setup.

4.6.2.2 *Plugin dependencies*

Sometimes you want to invoke code on other plugins.

Doing this will be simple if you are familiar with maven. All you have to do is adding a dependency to the other plugin. Edit your pom.xml and add the following code to the dependencies section:

```
<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>
```

You will notice that there are some things going on and that eclipse tries to download MCLIB. There is a complete chapter explaining the background later on.

For the moment simply notice that eclipse makes MCLIB available from your project.

Restarting the server will show you that it now loads the MCLIB because you declared a dependency on it. Very magical and fairly simple if you understand it.

4.6.3 Server administration

You already found the `eula.txt` on the first start. If you take a closer look you will see a complete `spigot` folder with all the files you already know from productive servers.

There is a `plugin` folder. Do not mind the `“*.eclipseproject”` or `“*.jar”` files within this folder. But you will find all the sub folders holding the data files from your plugins. Everything you are knowing from productive servers.

4.7 Server project setup scenarios

I showed you how to write a sample plugin. Some kind of Hello-World including JUnit tests.

Within this chapter, you will learn some more detailed background on dependencies. I will give you some suggestion on a favorite maven setup for NMS projects and API-aware projects, such as MCLIB.

4.7.1 Dependency background

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. Everything that is not part of 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.

4.7.2 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>

```

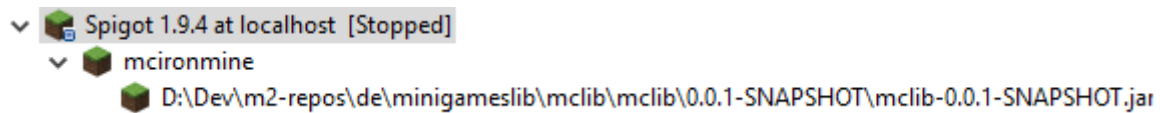
What do we do here?

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 runtime. 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.

4.7.3 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.

4.7.4 NMS classes

I have seen many plugins from users of different skill levels. I do not want to flame other developers. They are doing a nice job. As long as they are active.

But there are two problems:

- 1) Developers may become inactive
- 2) Developers may come back months later

Every time those two things happen we experience a problem: No one understands the original code. Even the original developer is unfamiliar with their own code.

Seeing MinigamesLib code from InstanceLabs and seeing other plugins found out that plugin developers tend to use Reflection to call the vanilla code. I really hate reflection. Yes, I hate it and you will learn to hate it too.

At first everything seems to be OK. And yeah, it seems to be clever that you do not need multiple projects and dependencies. It seems to be clever that your code works on different Minecraft versions without any change. But hey, vanilla is obfuscated. You know about methods with Names “a”, “b” or “A_”.

If you use reflection your java compiler will have no chance to detect changed signatures or changed method names. Your code will do strange things and you even do not know what is broken till you see some exception at runtime.

Calling the vanilla code without reflection is not fail-safe but it gives you a higher chance to see changes in method names during compilation time. In my 15 years of java experience I always prefer if compiler can detect problems for me.

4.7.5 Multi-project setup for NMS

The solution around this problem is a multi-project setup.

- 1) Create an API project only using the newest spigot API. This API project will hold interfaces of all things you will call version dependent code. Even if you use classes like “CraftWorld” from Bukkit, do never call it directly within your plugin but call it via your NMS API.

- 2) Create a NMS project for each version you like to support. One Project per version. If you want to support 1.8 to 1.11 you will need:
 - a. 1.8R1
 - b. 1.8R2
 - c. 1.8R3
 - d. 1.9R1
 - e. 1.9R2
 - f. 1.10R1
 - g. 1.11R1
- 3) Now place your NMS code for each version in the corresponding project. Use package names to divide them, for example “.....v1_8R1....”
- 4) At last add the API and the NMS modules to your main plugin project to make them available. Detect the proper version and use the classes from the target NMS project. Version detection is relay simple in MCLIB. TODO
- 5) The maven shade plugin will do the rest. It will package all your nms projects within a simple jar file during build.

4.7.6 Plugin API vs. Implementation

Looking at MCLIB and MinigamesLib you will find out that we have a huge API. This API is divided from plugin implementation.

Main goal is to let developers develop against a very stable API. They should not even see the implementation details. Many big projects have divided API modules from implementation modules. Developers of libraries are more flexible. They are careful and do not change API often. But they are flexible to change implementation details since it is not expected that anyone depend on them.

I already explained some background in the dependencies chapter how to add plugins as scope “provided” or scope “runtime”.

4.8 Forge client

TODO

5 Developing against MCLIB (Spigot server)

5.1 Dependencies

Edit your pom.xml and add the following dependencies:

```
<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>
```

We already mentioned the dependencies in the previous chapters. It will add the MCLIB api for compiling (scope Provided) and the MCLIB plugin itself on your spigot server (scope Runtime).

You will not have to include MCLIB into your plugin. So do not add it to the shade plugin. It will remain an independent plugin the server administrator must install along with your plugin.

Edit your plugin.yml and add the following line:

```
depend: [mclib]
```

This will tell spigot that you depend on MCLIB bringing the plugins in the right order during startup.

5.2 onEnable

During startup, we need to tell MCLIB what we want to do. Although some of the methods are always working at a later point this behavior may change in future versions. Simply do all your setup code in the “onEnable” method of your plugin.

I will explain the needed setup in the following chapters about the features.

If you require to access the MCLIB Api during “onEnable” ensure that the setup of your plugin is finished.

5.3 onDisable

MCLIB requires to perform some termination during “onDisable”. You should follow the instructions in the following chapters.

If you do not follow them there may be unexpected behavior while disabling and reloading plugins.

5.4 Version detection

To check the Minecraft version, you can use the following code fragment:

```

switch (McLibInterface.instance().getMinecraftVersion)
{
default:
    throw new IllegalStateException("Unsupported minecraft version");
case V1_11:
case V1_11_R1:
    // do whatever you like for V1.11
    break;
case V1_10:
case V1_10_R1:
    // do whatever you like for V1.10
    break;
}

```

Java is nice. It will even compile and execute this switch if the installed MCLIB is older than the version you compile against. But be aware of other constructs. The following may compile but on runtime it can lead to a “NoSuchFieldException”.

```

if (McLibInterface.instance().getMinecraftVersion().isAfter(
    MinecraftVersionsType.V1_11)

```

To solve this problem, you should always check the API-Version of MCLIB before doing any other things. For example, the following code will always succeed without runtime errors.

```

if (McLibInterface.instance().getApiVersion() !=
    McLibInterface.APIVERSION_1_0_0)
{
    throw new IllegalStateException("Incompatible Version");
}

```

Checking the MCLIB API-Version and the Minecraft Server Version during “onEnable” is recommended. If there are unexpected versions do not enable your plugin or the features using MCLIB.

But when does the API version change? We are always changing the api version as a new Minecraft/spigot version arrives. Any other api version change will be communicated.

5.5 Enumerations and their benefits

MCLIB uses java enumerations and interface default methods in many situations, for example for the multi-language system, configuration options and many other things.

We will explain the details later on.

Using the system is fairly simple. You declare a java enumeration that implements the interface. Some of the features will require some additional Annotations.

The most important thing happens in your “onEnable” method. There you will have to register the enumerations. For example:

```
EnumServiceInterface.instance().registerEnumClass(this, MyMessages.class);
EnumServiceInterface.instance().registerEnumClass(this, MyConfig.class);
```

Within the “onDisable” method we remove the enumeration classes.

```
EnumServiceInterface.instance().unregisterAllEnumerations(this);
```

Using enumerations have a major benefit: They are unique, they can be used as map keys without caring about the details. And they can be used in switch statements or with “==” (equals) operator etc.

The most important benefit is relay simple: You do not depend on string constants. A string constant may be misspelled and you may be confused. If you use misspell the wrong enumeration value name, the compiler detects it. MCLIB supports this in a flexible way. Removing configuration options will remove them in the config.yml automatically. Nothing you need care about.

5.5.1 Identifying plugin from enumeration

The EnumServiceInterface provides a method “getPlugin”. It tries to detect the plugin that registered an enumeration value.

If this method returns NULL, the numeration value was not registered. This method can be useful in many situations. For example, MinigamesLib has a feature to disable a minigame and thus it requires to disable arena types (which are enumerations) by plugin. Even if the plugin itself is not stopped.

Thus, the second method “getEnumValues” has signatures where the plugin is given. Interested API users can query the enum service for all enumeration values a plugin registered.

For example, the following code returns the Multi-Language nodes:

```
final Set<LocalizedMessageInterface> messages =
    EnumerationServiceInterface.instance().getEnumValues(
        LocalizedMessageInterface.class);
```

The following snippet returns all messages of “myPlugin”.

```
final Set<LocalizedMessageInterface> messages =
    EnumerationServiceInterface.instance().getEnumValues(
        myPlugin, LocalizedMessageInterface.class);
```

5.5.2 Building your own enumerations

MCLIB is not limited to enumerations of their own API. You can at least use any enumeration with it.

- 1) You should first declare an interface to let the system work correctly.

```
Public interface MySpecialEnum
{
    default void sysout()
    {
        System.out.println("Called sysout from " + this.getClass().getName() +
            "." + this.name());
    }
    String name(); // enumeration value name
}
```

- 2) Now build some enumerations using this interface.

```
public enum MyEnum1 implements MySpecialEnum
{
    MyValue1A,
    MyValue1B
}
public enum MyEnum2 implements MySpecialEnum
{
    MyValue2A,
    MyValue2B
}
```

- 3) Register the enumerations with MCLIB during "onEnable" method.

```
EnumServiceInterface.instance().registerEnumClass(this, MyEnum1.class);
EnumServiceInterface.instance().registerEnumClass(this, MyEnum2.class);
```

- 4) Now let us invoke sys out on all special enums the MCLIB knows about.

```
EnumerationServiceInterface.instance().getEnumValues(
    MySpecialEnum.class).forEach(MySpecialEnum::sysout);
```

As you will see it will print out the sysout for all four enumeration values. Pretty cool, isn't it?

And you are always allowed to do something like this:

```
If (myValue == MyEnum1.MyValue1A) ...
```

Or something like that:

```
MyEnum1.MyValue1A.sysout();
```

5.5.3 Child enumerations

Structuring your code and having multiple enumerations can be confusing. You can split your enumerations into several pieces to structure the code. You already have seen that you can register multiple enumeration classes implementing the same interface. There is no limitation.

Hiding enumeration complexity from your plugin main class (onEnable) is really simple. You can use the “@ChildEnum” annotation.

```
@ChildEnum({MyEnum1.class, MyEnum2.class})
public enum MyCoreEnum implements MySpecialEnum
{
    CoreValue,
    CoreFoo,
    MyValue
}
public enum MyEnum1 implements MySpecialEnum
{
    MyValue1A,
    MyValue1B
}
public enum MyEnum2 implements MySpecialEnum
{
    MyValue2A,
    MyValue2B
}
```

Now you only need to register “MyCoreEnum”. The enumerations “MyEnum1” and “MyEnum2” are registered automatically.

When to use this feature? I think the most common use case is the Multi-Language system. There may be common messages used by your plugin in different situations. But most of the time you declare a message to be used once somewhere deep inside the code. Personally, I use to declare the Message enumeration within the class I use it. So, to say: Having the message texts exactly on the same place they are used.

But I cannot say you MUST split of your enumerations or even you MUST use the ChildEnum feature. Use it whenever you like or do not. It is up to you.

5.6 Multi-language and customizable messages

InstanceLabs already followed the principle to customize every message the user is seeing. Because a developer should not decide and hard code user experience. If an administrator likes to customize messages he should do it. We followed this principle.

On top of this principle we added a multi-language system. The administrator decides which languages he likes and the users decide which they are using during runtime. This requires us to build some localization into MCLIB.

Looking at some solutions did not satisfy all of our needs. We require:

- Dividing multiple languages
- Let the administrator change values in messages.yml files
- Divide user messages from admin/operator messages
- Let the administrator choose a default language
- Let the user choose their favorite language
- Support Minecraft color encodings
- Support raw messages
- Let the administrator install language packs.

Before losing our brain, we decided to create our own message system. I hope it is both, simple and powerful.

5.6.1 Servers default locale

First of all, we support a default locale. Whenever a user did not make a choice for his favorite language this default language is chosen.

You can receive or change the value through McLibInterface:

```
McLibInterface.instance().getDefaultLocale();
McLibInterface.instance().setDefaultLocale(...);
```

Setting the default locale programmatically is possible but not recommended. Because a plugin should not change the value. Instead it is up to the administrator, to either change config.yml or to use a chat command to change the value.

5.6.2 Users preferred locale

The users preferred locale can be received and set through “getPreferredLocale” or “setPreferredLocale”.

```
final McPlayerInterface player = ...;
player.getPreferredLocale();
player.setPreferredLocale(...);
```

Once again setting the preferred locale programmatically is not recommended. Instead the players should choose the preferred locale through chat api.

The users preferred locale may be NULL. This means the user is using the servers default.

5.6.3 Declaring messages

For an example have a look at the “CommonMessages” enumeration. It is part of the API. You can declare your own message enumeration. Remember to register it in your “onEnable” method.

```
@LocalizedMessages("some.path")
public enum MyMessages implements LocalizedMessageInterface
{
    @LocalizedMessage(
        defaultMessage = "Hello World!",
        defaultAdminMessage = "Hello Operator!",
        severity = MessageSeverityType.Success
    )
    HelloWorld
}
```

What have we done?

- 1) The class level annotation “@LocalizedMessages” is required. It holds some useful information. First of all, the path within messages.yml. If you are using multiple message enumerations each one should have different paths to not have conflicts.

In this sample, we do not override the default language. If you add the defaultLocale attribute it will override the language of your default values. It may be not clever to override the default locale except for some private plugins you only use on your server. Keep in mind that on public servers the administrators assume that at least one language is supported: English or “en”. If you try to share your plugins with public keep this in mind.

- 2) The enumeration implements “LocalizedMessageInterface” for using the message in the API and for having some additional methods on it.
- 3) The enumeration value uses one of two required annotations: “@LocalizedMessage” for a single line message.

It contains a user message and an (optional) administrator message. It declares a message severity causing some nice coloring of the message.

5.6.4 Using the message

The simplest use case is to send the message to players. Looking at the McPlayerInterface” you will find the method “sendMessage”.

```
final McPlayerInterface player = ...;
player.send(MyMessages.HelloWorld);
```

That’s all the magic. You do not need to ask for locales nor to check if the player is an operator or a normal user. No coloring questions. Simply send the message and let MCLIB do the rest.

A second option is to encode the message into a java string array (one entry per line). This is done at the McPlayerInterface too.

```
final McPlayerInterface player = ...;  
final String[] msgArr = player.encodeMessage(MyMessages.HelloWorld);
```

5.6.5 Message arguments

TODO

5.6.6 Multi line messages

TODO

5.6.7 Language packs

TODO

5.6.8 Exceptions with messages

TODO

5.6.9 messages.yml and Message comments

TODO

5.6.10 Predefined messages

TODO

5.7 Command handling

TODO

5.8 Context sensitive execution

TODO

5.9 Configuration Framework and DataFragments

TODO

5.10 Storages

TODO

5.11 Extension handling

TODO

5.12 Simple GUI

TODO

5.13 Permissions

TODO

5.14 Objects framework

TODO

5.15 Event framework

TODO

6 Developing against MCLIB (Forge clients)

TODO

7 BungeeCord-Clusters and Client/Server

7.1 Communication channels

TODO

8 Smart GUI

TODO

9 Advanced testing

9.1 spigot-testsupport TODO

9.2 fakeplayer TODO