minigameslib.de

McLib

Minigames Core Library – Eclipse plugins and Junit-Support

mepeisen

30.01.2017

1 Table of contents

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 scenario 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/

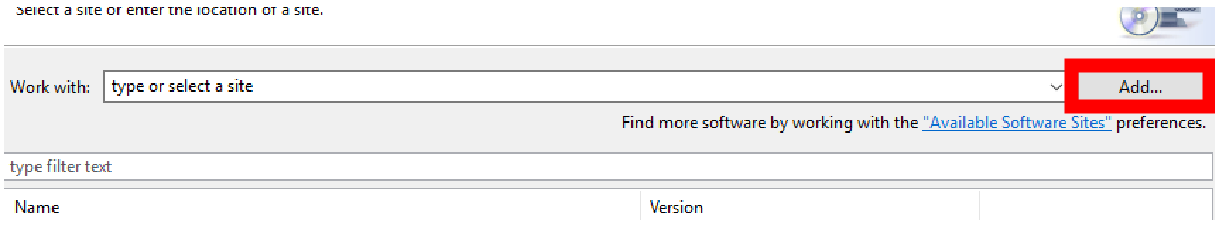
We use JDK 8 for development.

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.


Select a site or enter the location of a site.

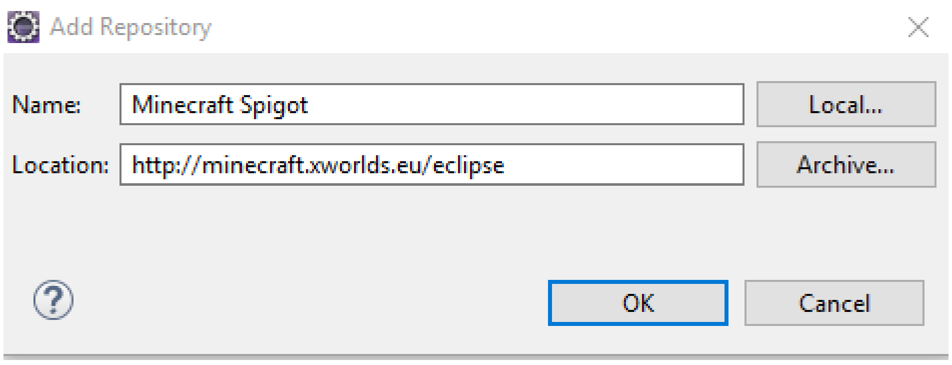
Work with: type or select a site

Find more software by working with the "Available Software Sites" preferences.

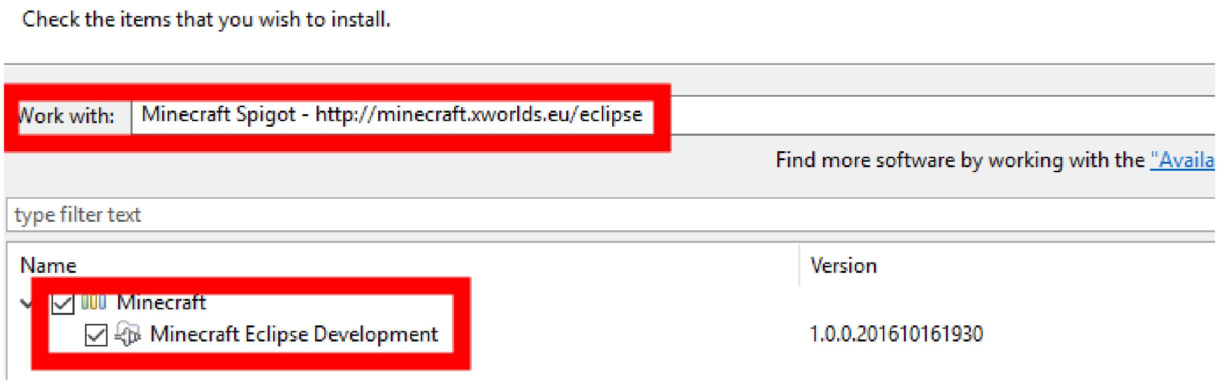
type filter text

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.
Work with: Minecraft Spigot - http://minecraft.xworlds.eu/eclipse
Find more software by working with the "Availa
type filter text
Name
Version
Minecraft
Minecraft Eclipse Development
1.0.0.201610161930

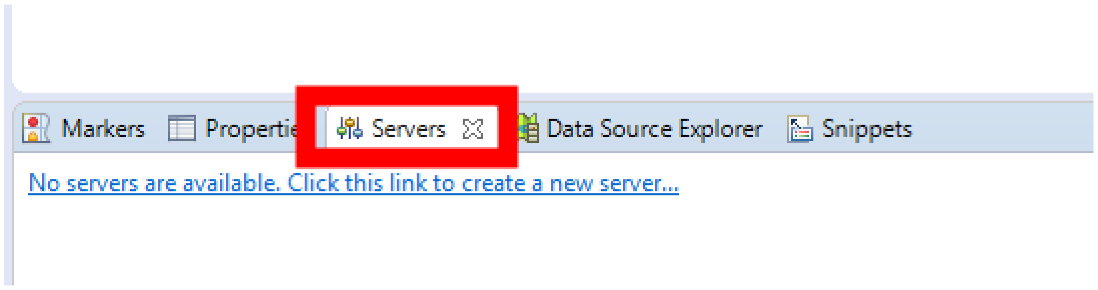
4) Follow the wizard steps and let eclipse install the plugins.

4.3 Setup spigot servers

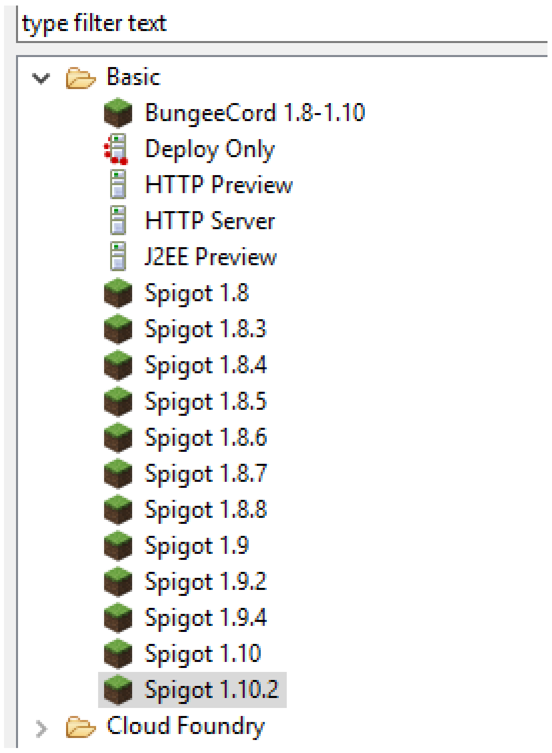
Some background information: Eclipse webtools divides servers and runtimes. A runtime is a location of your spigot server on your local hdd. A server holds all configurations, plugins and everything you need to start spigot.

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”:


Markers
Propertie
Servers
Data Source Explorer
Snippets
No servers are available. Click this link to create a new server...

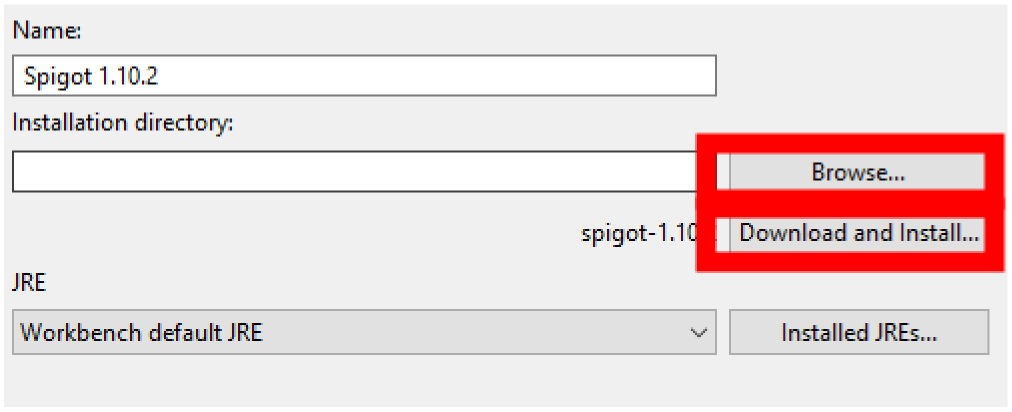
2) Click on the hyperlink. In the “new Server” wizard select the spigot server under category “Basic”.



type filter text

- Basic
  - BungeeCord 1.8-1.10
  - Deploy Only
  - HTTP Preview
  - HTTP Server
  - J2EE Preview
  - Spigot 1.8
  - Spigot 1.8.3
  - Spigot 1.8.4
  - Spigot 1.8.5
  - Spigot 1.8.6
  - Spigot 1.8.7
  - Spigot 1.8.8
  - Spigot 1.9
  - Spigot 1.9.2
  - Spigot 1.9.4
  - Spigot 1.10
  - Spigot 1.10.2
- Cloud Foundry

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.



Name: Spigot 1.10.2

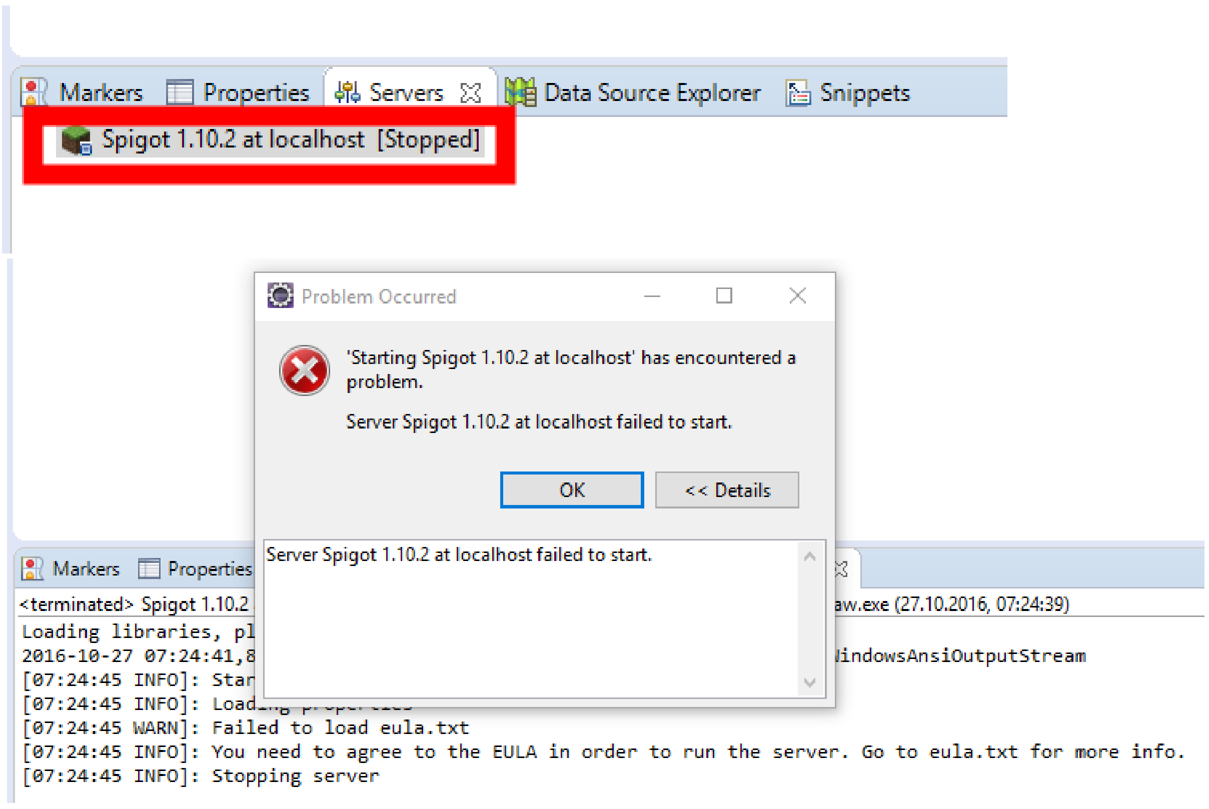
Installation directory:

spigot-1.10

JRE: Workbench default JRE

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.



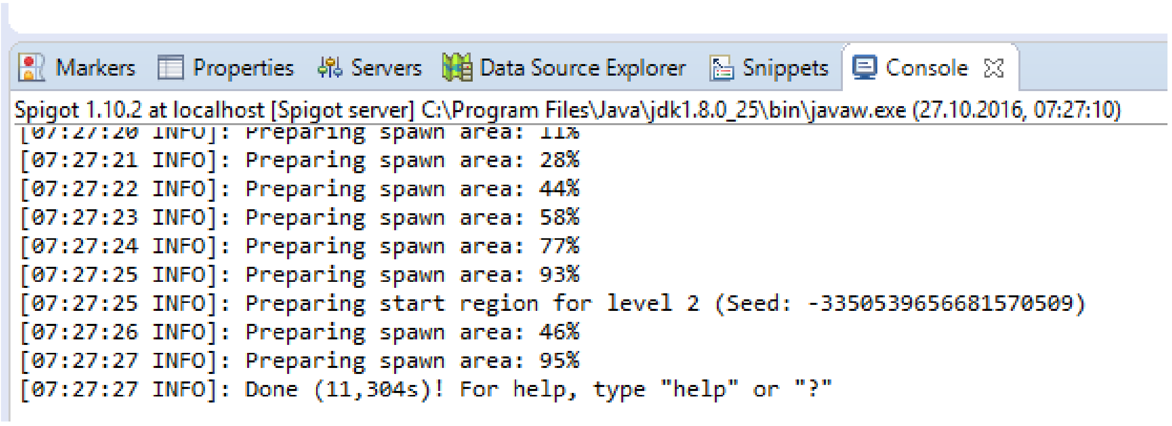
'Starting Spigot 1.10.2 at localhost' has encountered a problem.

Server Spigot 1.10.2 at localhost failed to start.

Server Spigot 1.10.2 at localhost failed to start.

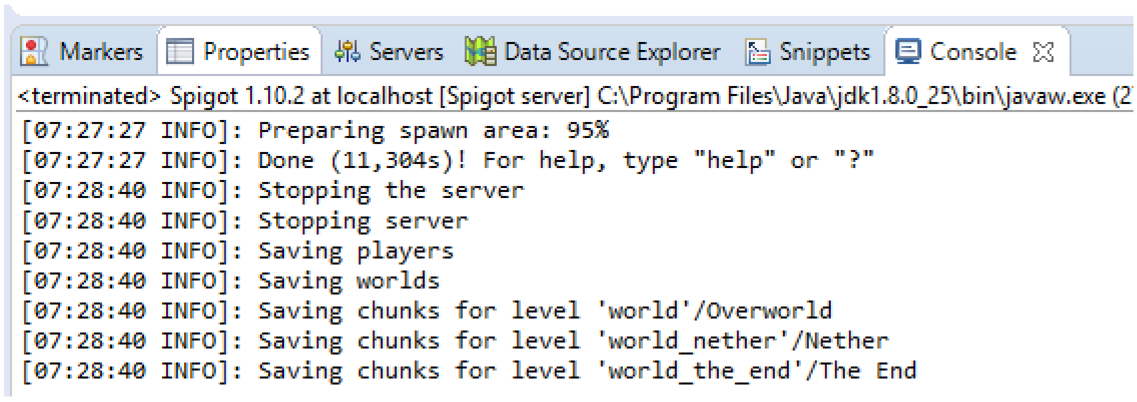
```
<terminated> Spigot 1.10.2                                              aw.exe (27.10.2016, 07:24:39)
Loading libraries, pl
2016-10-27 07:24:41,8                                                   WindowsAnsiOutputStream
[07:24:45 INFO]: Star
[07:24:45 INFO]: Load
[07:24:45 WARN]: Failed to load eula.txt
[07:24:45 INFO]: You need to agree to the EULA in order to run the server. Go to eula.txt for more info.
[07:24:45 INFO]: Stopping server
```

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.



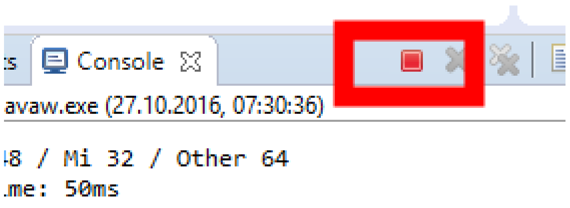
```
Spigot 1.10.2 at localhost [Spigot server] C:\Program Files\Java\jdk1.8.0_25\bin\javaw.exe (27.10.2016, 07:27:10)
[07:27:20 INFO]: Preparing spawn area: 11%
[07:27:21 INFO]: Preparing spawn area: 28%
[07:27:22 INFO]: Preparing spawn area: 44%
[07:27:23 INFO]: Preparing spawn area: 58%
[07:27:24 INFO]: Preparing spawn area: 77%
[07:27:25 INFO]: Preparing spawn area: 93%
[07:27:25 INFO]: Preparing start region for level 2 (Seed: -3350539656681570509)
[07:27:26 INFO]: Preparing spawn area: 46%
[07:27:27 INFO]: Preparing spawn area: 95%
[07:27:27 INFO]: Done (11,304s)! For help, type "help" or "?"
```

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.

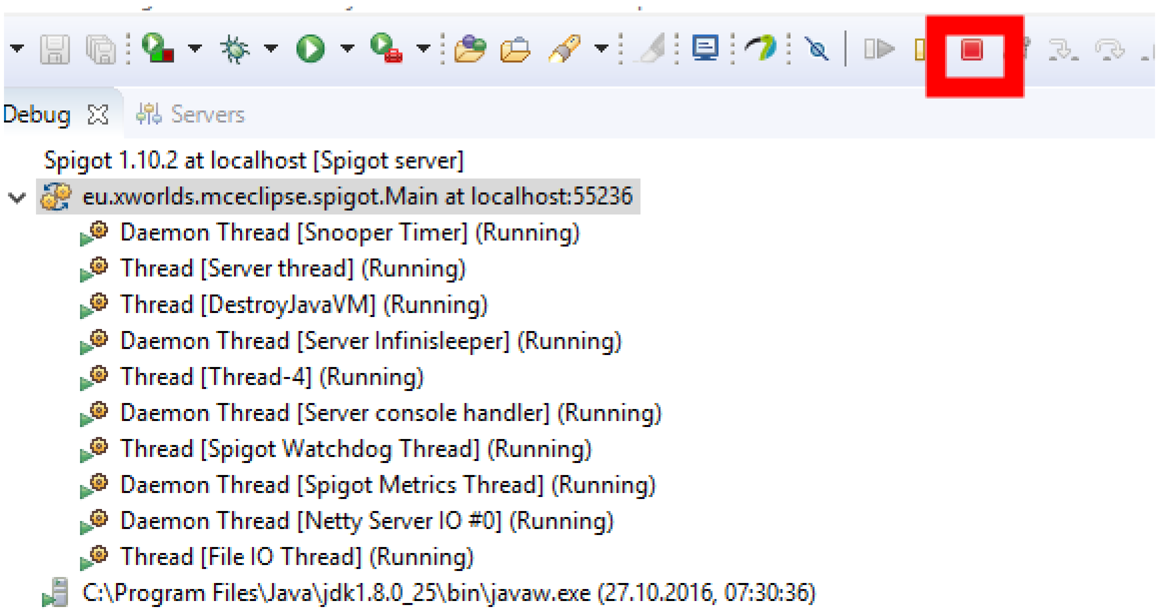


```
<terminated> Spigot 1.10.2 at localhost [Spigot server] C:\Program Files\Java\jdk1.8.0_25\bin\javaw.exe (2
[07:27:27 INFO]: Preparing spawn area: 95%
[07:27:27 INFO]: Done (11,304s)! For help, type "help" or "?"
[07:28:40 INFO]: Stopping the server
[07:28:40 INFO]: Stopping server
[07:28:40 INFO]: 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.



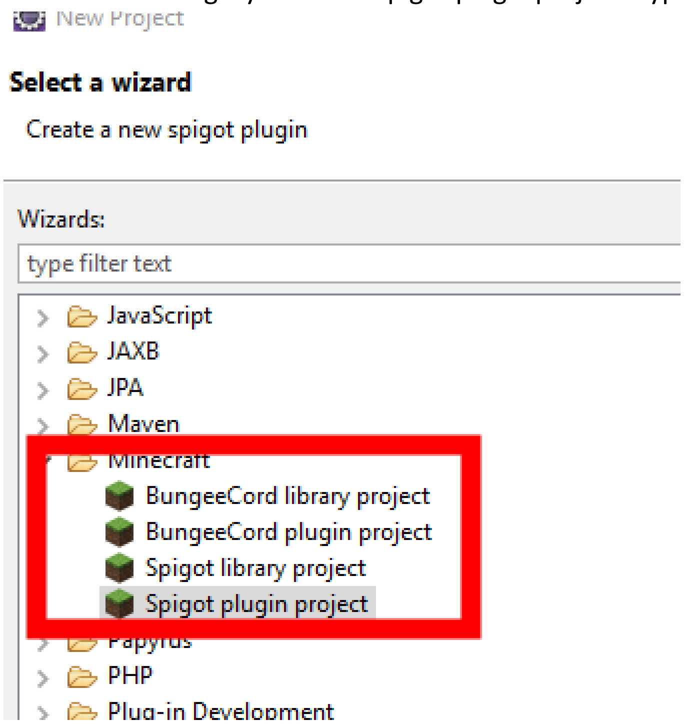
```
avaw.exe (27.10.2016, 07:30:36)
8 / Mi 32 / Other 64
me: 50ms
```



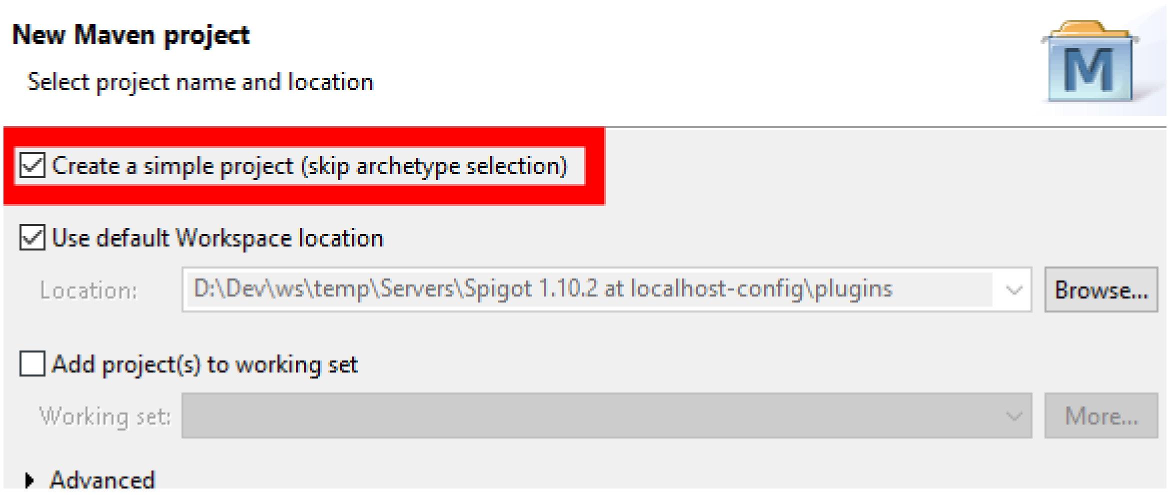
```
Spigot 1.10.2 at localhost [Spigot server]
  eu.xworlds.mceclipse.spigot.Main at localhost:55236
    Daemon Thread [Snooper Timer] (Running)
    Thread [Server thread] (Running)
    Thread [DestroyJavaVM] (Running)
    Daemon Thread [Server Infinisleeper] (Running)
    Thread [Thread-4] (Running)
    Daemon Thread [Server console handler] (Running)
    Thread [Spigot Watchdog Thread] (Running)
    Daemon Thread [Spigot Metrics Thread] (Running)
    Daemon Thread [Netty Server IO #0] (Running)
    Thread [File IO Thread] (Running)
  C:\Program Files\Java\jdk1.8.0_25\bin\javaw.exe (27.10.2016, 07:30:36)
```

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.



2) Select the "Create a simple project" checkbox.



☑ Create a simple project (skip archetype selection)

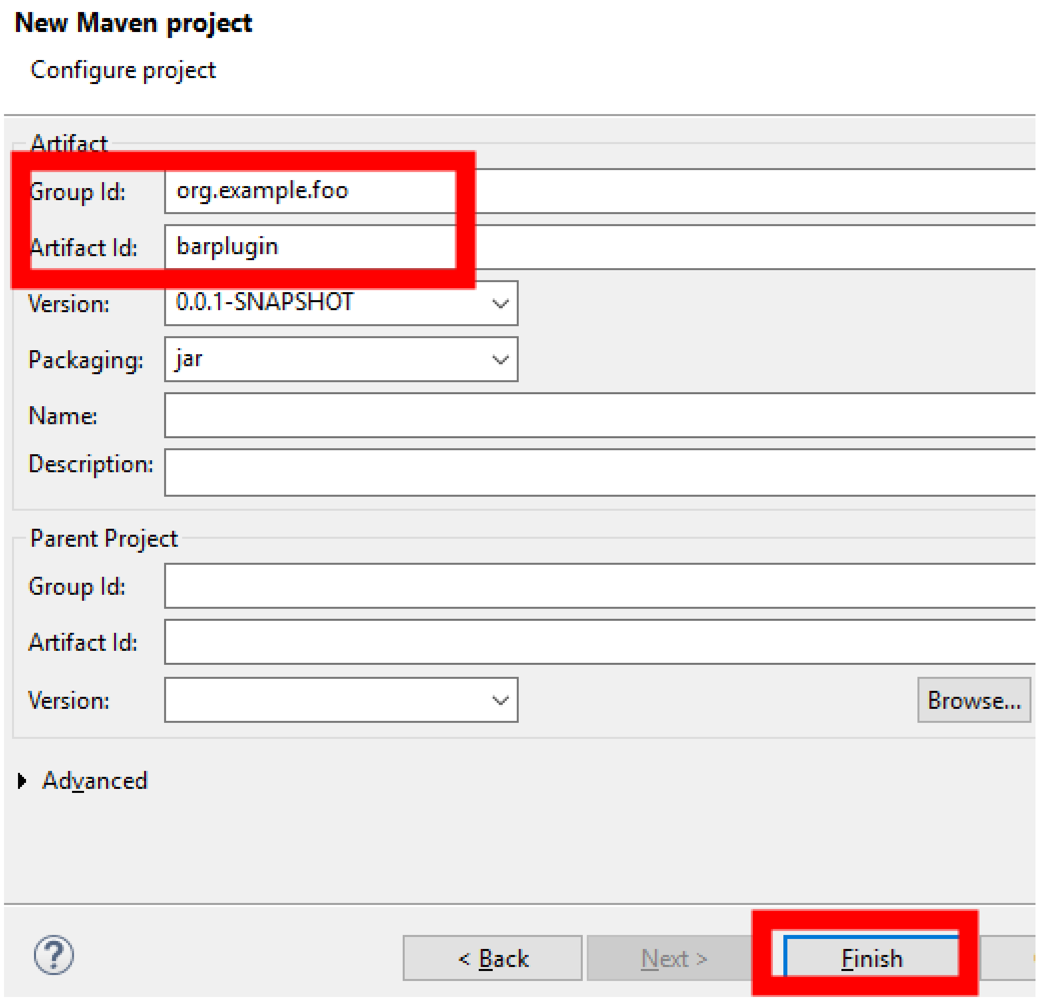
☑ Use default Workspace location

Location: D:\Dev\ws\temp\Servers\Spigot 1.10.2 at localhost-config\plugins

☐ Add project(s) to working set

Working set:

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

Artifact

Group Id: org.example.foo

Artifact Id: barplugin

Version: 0.0.1-SNAPSHOT

Packaging: jar

Name:

Description:

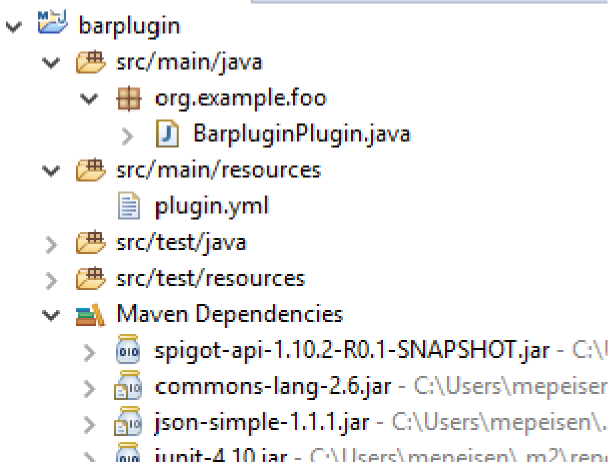
Parent Project

Group Id:

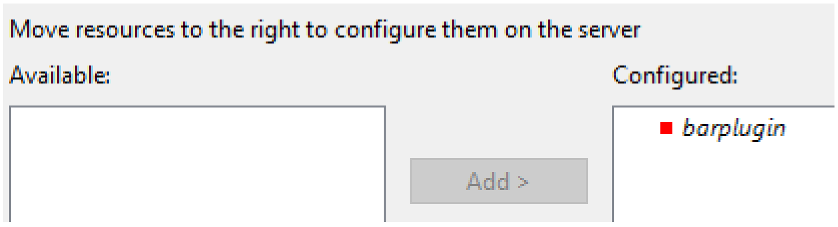
Artifact Id:

Version:

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\rep
```

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.

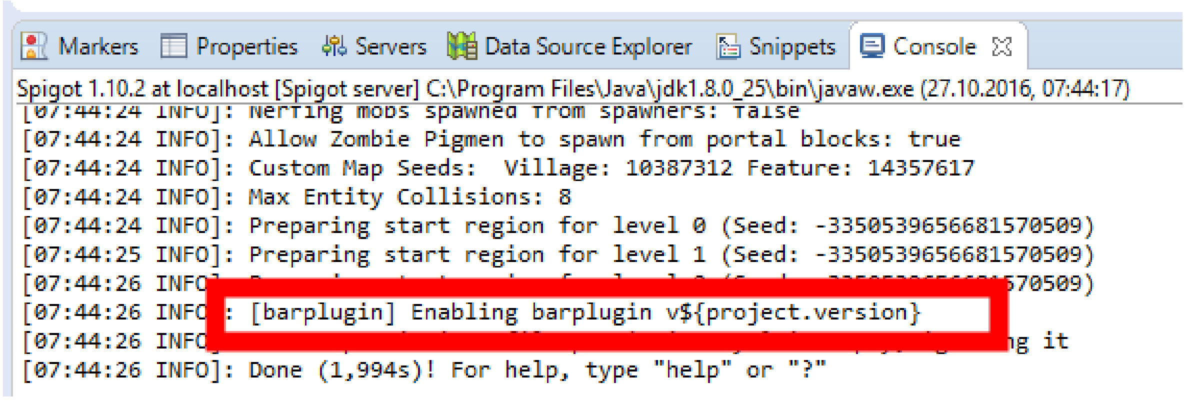


Move resources to the right to configure them on the server

Available: Configured:

barplugin

6) Restart the server and watch the console.

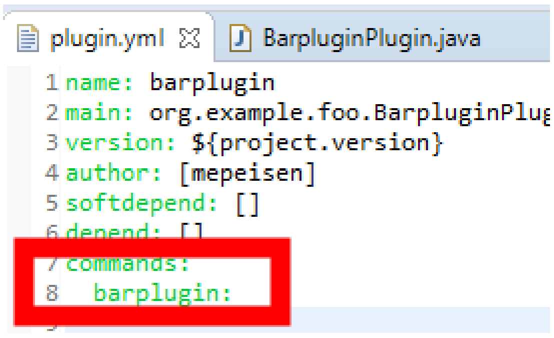


Spigot 1.10.2 at localhost [Spigot server] C:\Program Files\Java\jdk1.8.0_25\bin\javaw.exe (27.10.2016, 07:44:17)

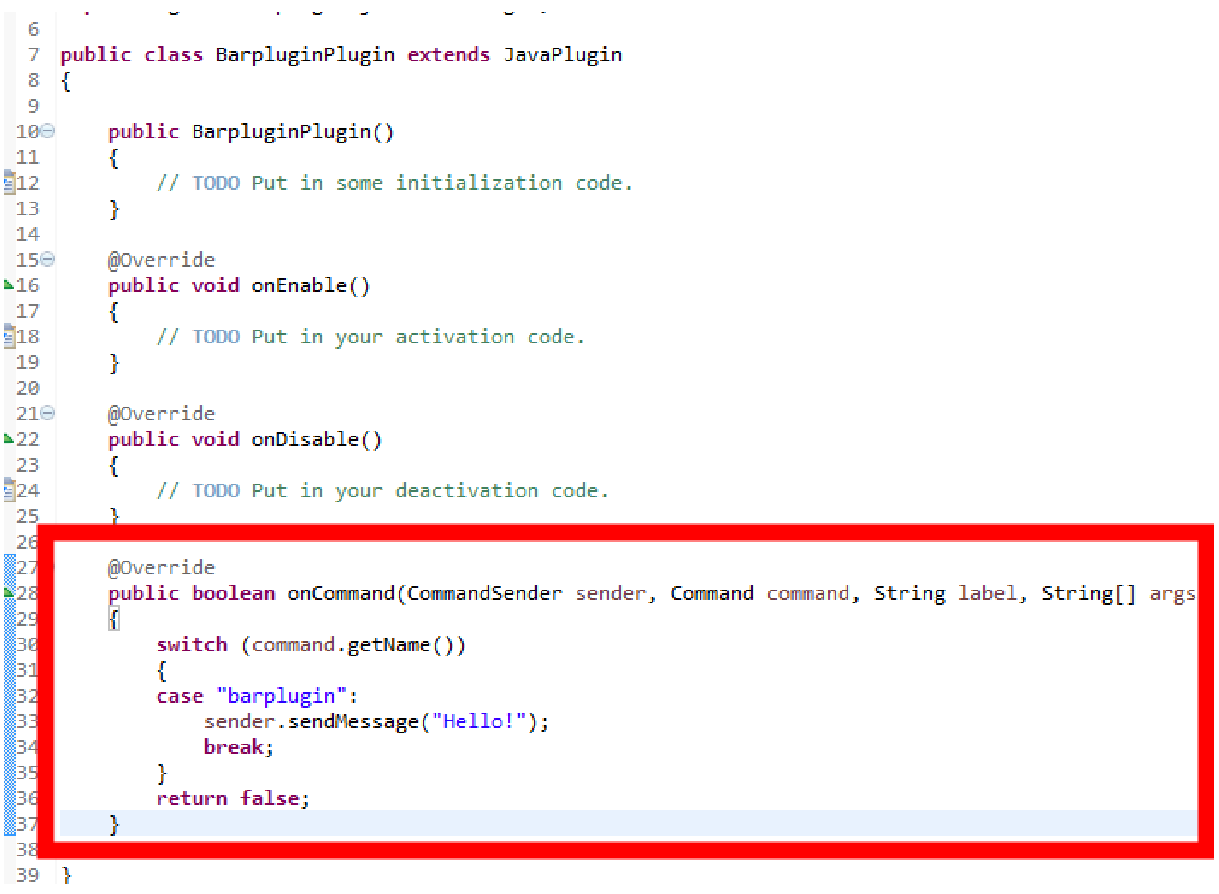
```
[07:44:24 INFO]: Nerfing mobs spawned from spawners: false
[07:44:24 INFO]: Allow Zombie Pigmen to spawn from portal blocks: true
[07:44:24 INFO]: Custom Map Seeds:  Village: 10387312 Feature: 14357617
[07:44:24 INFO]: Max Entity Collisions: 8
[07:44:24 INFO]: Preparing start region for level 0 (Seed: -3350539656681570509)
[07:44:25 INFO]: Preparing start region for level 1 (Seed: -3350539656681570509)
[07:44:26 INFO                                                        570509)
[07:44:26 INFO : [barplugin] Enabling barplugin v${project.version}
[07:44:26 INFO                                                         ng it
[07:44:26 INFO]: Done (1,994s)! For help, type "help" or "?"
```

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.

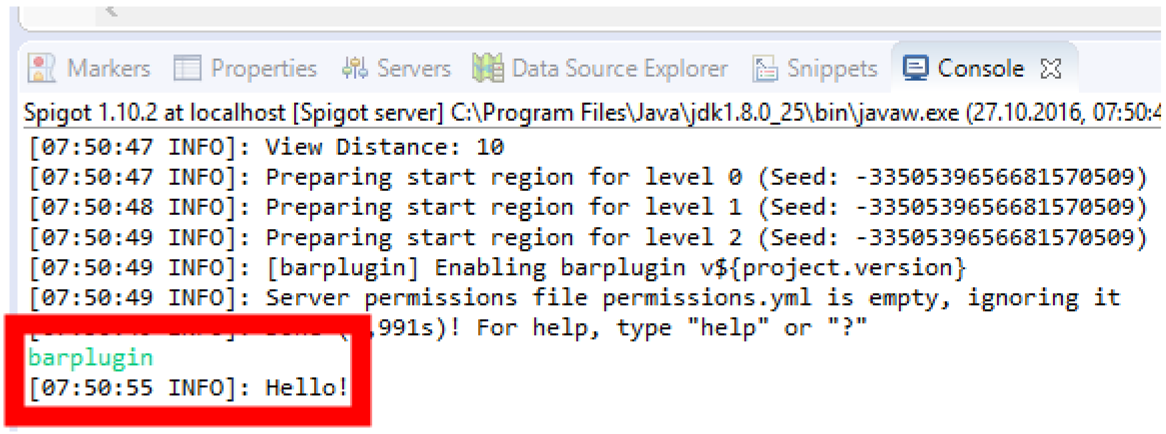


```
name: barplugin
main: org.example.foo.BarpluginPlug
version: ${project.version}
author: [mepeisen]
softdepend: []
depend: []
commands:
  barplugin:
```



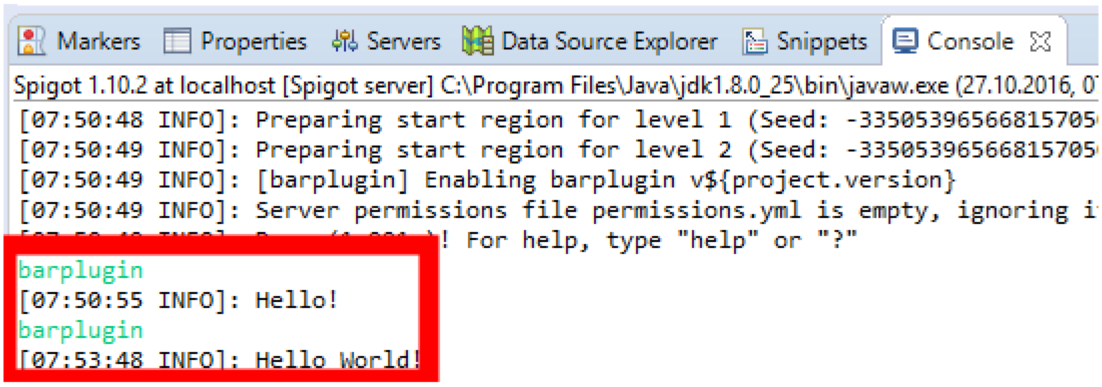
```java

public class BarpluginPlugin extends JavaPlugin
{

    public BarpluginPlugin()
    {
        // TODO Put in some initialization code.
    }

    @Override
    public void onEnable()
    {
        // TODO Put in your activation code.
    }

    @Override
    public void onDisable()
    {
        // TODO Put in your deactivation code.
    }

    @Override
    public boolean onCommand(CommandSender sender, Command command, String label, String[] args
    {
        switch (command.getName())
        {
        case "barplugin":
            sender.sendMessage("Hello!");
            break;
        }
        return false;
    }

}
```

Now start the server and check your code.



```
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
                                          ,991s)! For help, type "help" or "?"
barplugin
[07:50:55 INFO]: Hello!
```

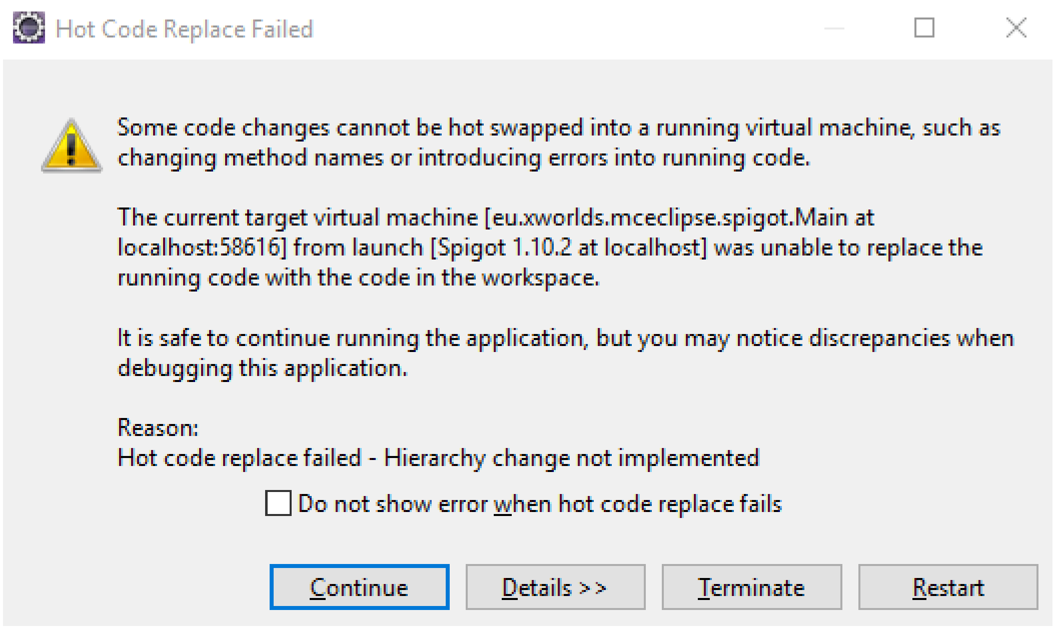
It is working. But let us now use the hot code replacement. WIHTOUT restarting the server change the message in your code to “Hello World!”. Save the java file and check the command again.



```
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: -335053965668157050
[07:50:49 INFO]: Preparing start region for level 2 (Seed: -335053965668157050
[07:50:49 INFO]: [barplugin] Enabling barplugin v${project.version}
[07:50:49 INFO]: Server permissions file permissions.yml is empty, ignoring i
[07:50:49 INFO]: [illegible]! For help, type "help" or "?"
barplugin
[07:50:55 INFO]: Hello!
barplugin
[07:53:48 INFO]: 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.



Hot Code Replace Failed

Some code changes cannot be hot swapped into a running virtual machine, such as changing method names or introducing errors into running code.

The current target virtual machine [eu.xworlds.mceclipse.spigot.Main at localhost:58616] from launch [Spigot 1.10.2 at localhost] was unable to replace the running code with the code in the workspace.

It is safe to continue running the application, but you may notice discrepancies when debugging this application.

Reason:
Hot code replace failed - Hierarchy change not implemented

☐ Do not show error when hot code replace fails

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

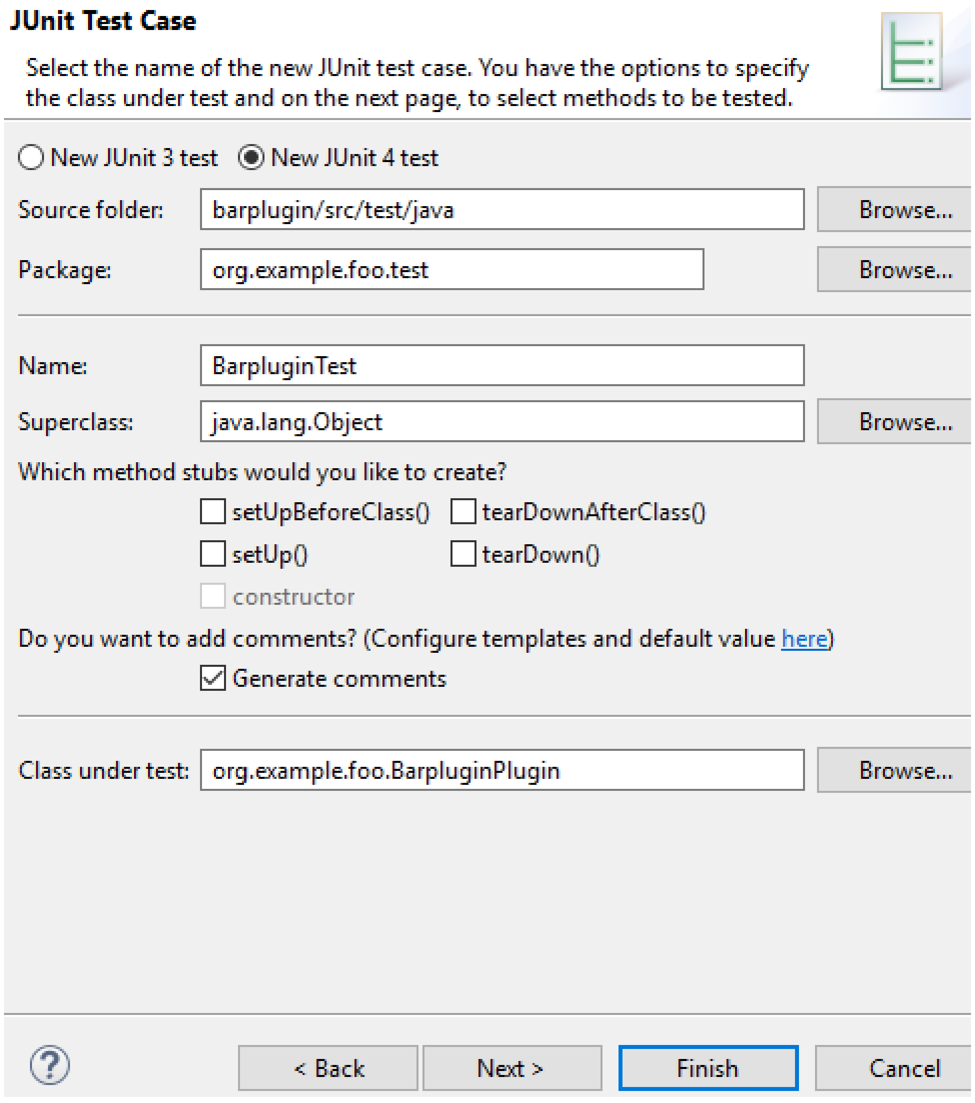
4.6 Advanced spigot topics

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: barplugin/src/test/java

Package: org.example.foo.test

Name: BarpluginTest

Superclass: java.lang.Object

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: org.example.foo.BarpluginPlugin

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

```
 */
@RunWith(SpigotJunit4Runner.class)
@SpigotTest(versions = {SpigotVersion.V1_10_2})
public class BarpluginTest {
```

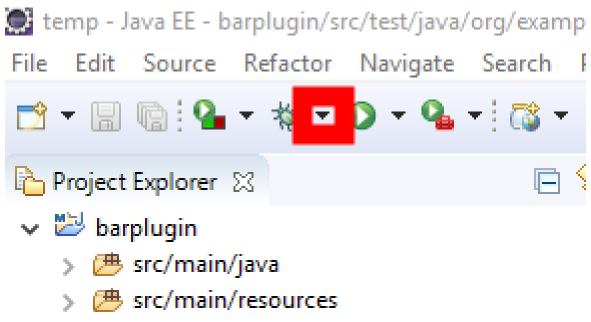
4) To access our spigot server during junit tests we need to add a variable annotated with “SpigotInject”.

```
@SpigotInject
private SpigotServer server;
```

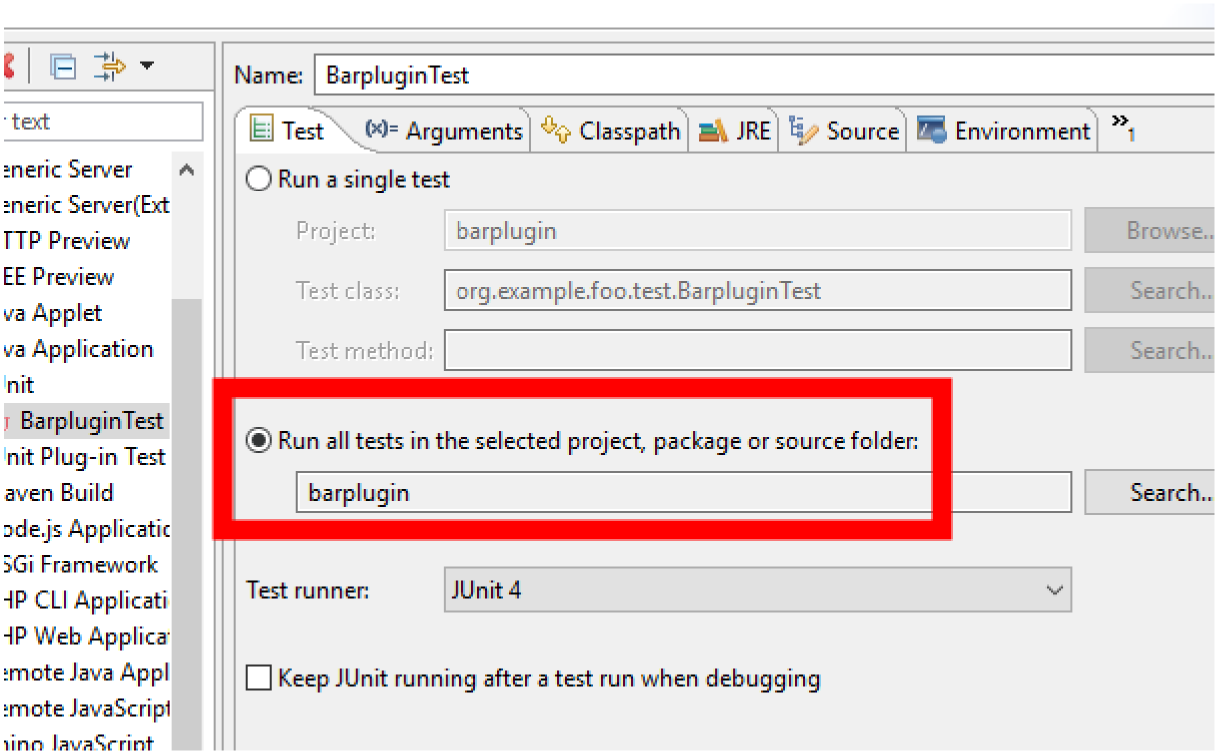
5) We are now ready to perform our test. As we already have a command we will simply test it:

```
@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…”.


temp - Java EE - barplugin/src/test/java/org/examp
File Edit Source Refactor Navigate Search
Project Explorer
barplugin
src/main/java
src/main/resources

7) Right click on "Junit" and select "New". Change the properties to run all tests from your project.



Name: BarpluginTest

Run a single test

Project: barplugin

Test class: org.example.foo.test.BarpluginTest

Test method:

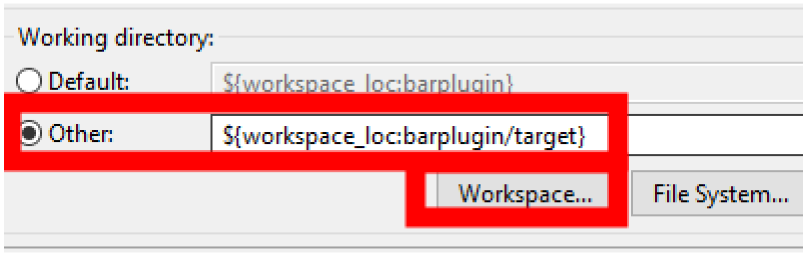
(●) Run all tests in the selected project, package or source folder:

barplugin

Test runner: JUnit 4

Keep JUnit running after a test run when debugging

8) Change the arguments to start in directory "target".

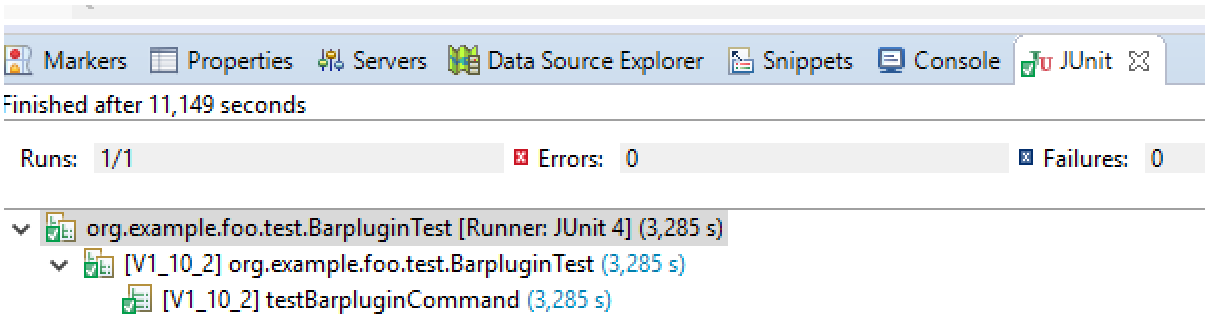


Working directory:

Default: ${workspace_loc:barplugin}

(●) Other: ${workspace_loc:barplugin/target}

9) Start the test by selecting the debug button.



```
Finished after 11,149 seconds

Runs: 1/1        Errors: 0        Failures: 0

org.example.foo.test.BarpluginTest [Runner: JUnit 4] (3,285 s)
  [V1_10_2] org.example.foo.test.BarpluginTest (3,285 s)
    [V1_10_2] testBarpluginCommand (3,285 s)
```

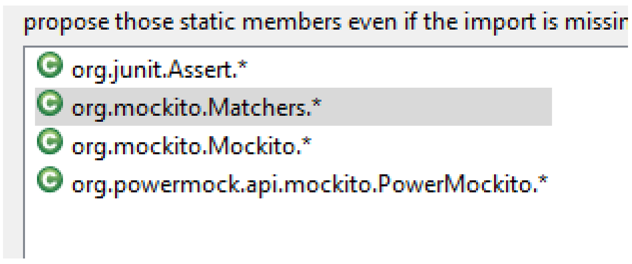
Hint: We do not really need our own debug configuration for the whole project and with folder "target".

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:

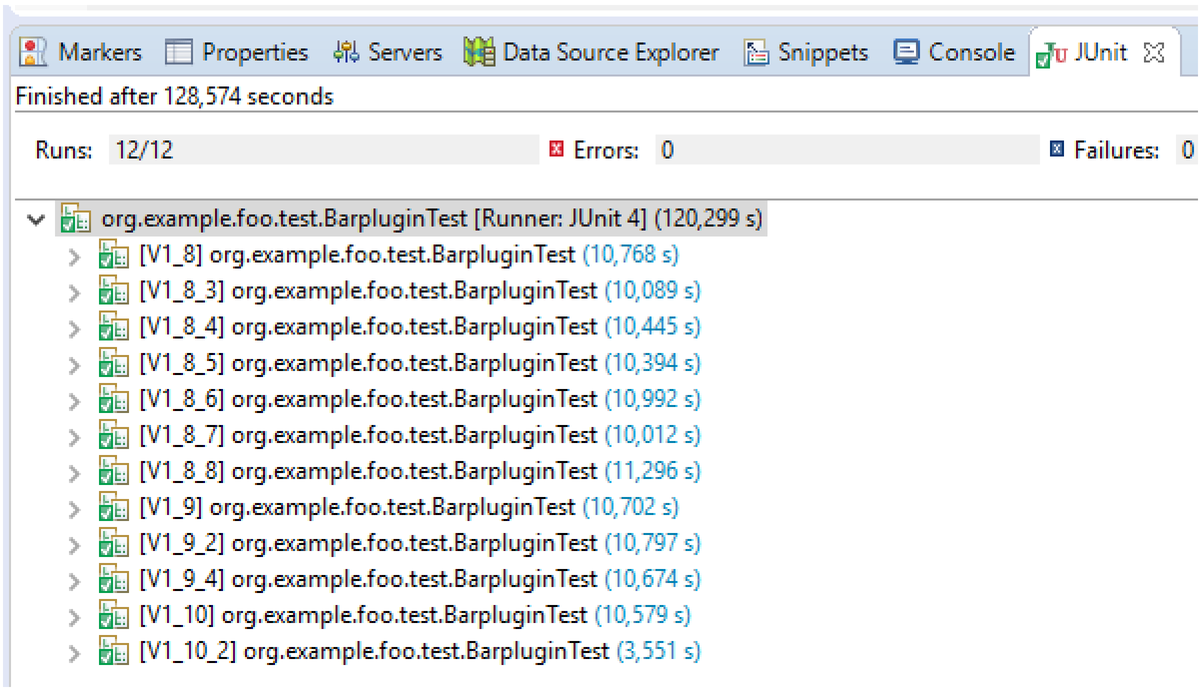


propose those static members even if the import is missir

- org.junit.Assert.*
- org.mockito.Matchers.*
- org.mockito.Mockito.*
- org.powermock.api.mockito.PowerMockito.*

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:



```
Finished after 128,574 seconds

Runs: 12/12        Errors: 0        Failures: 0

org.example.foo.test.BarpluginTest [Runner: JUnit 4] (120,299 s)
  [V1_8] org.example.foo.test.BarpluginTest (10,768 s)
  [V1_8_3] org.example.foo.test.BarpluginTest (10,089 s)
  [V1_8_4] org.example.foo.test.BarpluginTest (10,445 s)
  [V1_8_5] org.example.foo.test.BarpluginTest (10,394 s)
  [V1_8_6] org.example.foo.test.BarpluginTest (10,992 s)
  [V1_8_7] org.example.foo.test.BarpluginTest (10,012 s)
  [V1_8_8] org.example.foo.test.BarpluginTest (11,296 s)
  [V1_9] org.example.foo.test.BarpluginTest (10,702 s)
  [V1_9_2] org.example.foo.test.BarpluginTest (10,797 s)
  [V1_9_4] org.example.foo.test.BarpluginTest (10,674 s)
  [V1_10] org.example.foo.test.BarpluginTest (10,579 s)
  [V1_10_2] org.example.foo.test.BarpluginTest (3,551 s)
```

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. 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.

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>
```

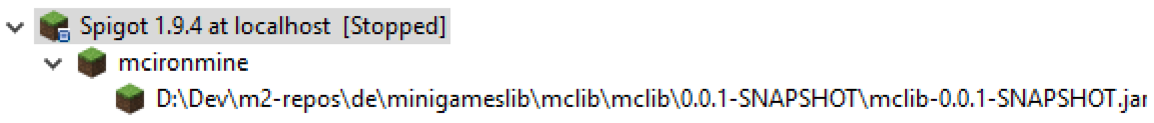
What do we do here?

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:



```
Spigot 1.9.4 at localhost  [Stopped]
  mcironmine
    D:\Dev\m2-repos\de\minigameslib\mclib\mclib\0.0.1-SNAPSHOT\mclib-0.0.1-SNAPSHOT.jar
```

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 really simple in McLib. See Chapter 5.4 for details.

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 forge client scenario

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.

```
EnumServiceInstance.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 extends EnumerationValue
{
    default void sysout()
    {
        System.out.println("Called sysout from " + this.getClass().getName() +
            "." + this.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();
```

5.5.2.1 UniqueEnumerationValue

You can inherit UniqueEnumerationValue to ensure unique names of your enumeration. For each plugin, the McLib will throw exceptions if it registers an enumeration with already existing names.

If you use this interface it will be safe to identify enumerations values by two strings: plugin name and enumeration name. You do not need to store class names.

5.5.2.2 McEnumInterface / McUniqueEnumInterface

It is preferred to use this interfaces within server code. It contains default methods to get the java plugins and plugin names for enumeration values.

Why do we have four interfaces? The reason is simple. EnumerationValue and UniqueEnumerationValue are part of the shared api and they are available on both, client and server. While the other two interfaces are only available within server code.

Choosing the correct interface depends on whether you need the enumeration on client or on servers.

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

Each message can contain arguments. Argument substitution is done through Javas String format. See http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax for syntax details.

Here is a small example using two arguments.

```
    @LocalizedMessage(
        defaultMessage = “Hello %1$s!”,
        defaultAdminMessage = “Hello %1$s (%2$s)!”,
        severity = MessageSeverityType.Success
    )
    HelloWorld
```

This example uses two arguments. At first the players name and at second the players uuid. Only administrators will see their uuid. Now let us send this message along.

```
final McPlayerInterface player = …;
player.send(MyMessages.HelloWorld,
    player.getBukkitPlayer().getDisplayName(),
    player.getPlayerUUID().toString());
```

As you can see you do not need to wonder about the arguments and even if the server administrators like them. Simply pass them to the send method and let McLib do the rest.

The arguments are limited to Serializables.

5.6.6 Nesting messages

Sometimes you want to nest messages. That means pass a message as an argument to other messages. Here is a real-world example passing a proper welcome message depending on the current time.

First, we declare the messages.

```
        @LocalizedMessage(
            defaultMessage = “%1$s %2$s!”,
            defaultAdminMessage = “%1$s %2$s (%3$s)!”,
            severity = MessageSeverityType.Success
        )
        HelloWorld,

        @LocalizedMessage(
            defaultMessage = “Hello”
            severity = MessageSeverityType.Neutral
        )
        HelloDay,

        @LocalizedMessage(
            defaultMessage = “Good morning”
            severity = MessageSeverityType.Neutral
        )
        HelloMorning
```

Now let us send the messages.

```
    final McPlayerInterface player = …;
    final LocalTime time = LocalTime.now();
    LocalizedMessageInterface greeting = MyMessages.HelloDay;
    if (time.getHour() > 6 && time.getHour() < 11)
        greeting = MyMessages.HelloMorning;
    player.send(MyMessages.HelloWorld,
        greeting.toArg(),
        player.getBukkitPlayer().getDisplayName(),
        player.getPlayerUUID().toString());
```

The most important thing is to convert the nested message to a serializable by using method “toArg”.

Hint: The nested message has its own argument list. Pass arguments to method “toArg”. The nested message does NOT use the arguments of the container message.

5.6.7 Multi line messages

McLib supports messages over multiple lines. You do not have to take care about number of lines. If an administrator likes he can add as many lines as he wants to.

You can use every parameter in each line.

```
        @LocalizedMessageList(
            value = {“Hello, %1$s.”,
                “Your name is %1$s.”,
                “You are welcome to this Minecraft world.”,
                “Your UUID is %2$s.”}
            severity = MessageSeverityType.Success
        )
        HelloMultiLine
```

You can use this multi-line message like any other single line message.

```
    final McPlayerInterface player = …;
    player.send(MyMessages.HelloMultiLine,
        player.getBukkitPlayer().getDisplayName(),
        player.getPlayerUUID().toString());
```

5.6.8 Language packs

TODO language packs

5.6.9 Exceptions with messages

In many situations, you are able to throw a McException. For example, to show up that a chat command Is invalid.

If you have a look at the constructors, you will see all constructors taking localized messages as arguments. The reason is simple. Exceptions may not only be caught and logged into the server logs. They may be used to report the errors to the users too.

Keep in mind: Every time you throw a McException a user may read the message of the exception.

There is a good default message (see following chapters) you may use as a wrapper. It simply says “internal error” for normal users and print some useful information for administrators. You can use them for wrapping other exceptions, for example a SQLException from JDBC.

5.6.10 Message comments

You can set some administration hints for your messages. Those hints are placed as comments into the messags.yml file. Simply add the @MessageComment annotation. It takes a string list with each string representing a single line.

Here is an example how to use it:

```
    @LocalizedMessages(path = “core”)
    public enum MyMessages implements LocalizedMessageInterface
    {
        @LocalizedMessage(defaultMessage = “some text”)
        @MessageComment({
            “this is the first line”,
            “some other comment line”
        })
        SomeText
    }
```

The comment will be placed into the messages file if there is no existing comment. If your message supports arguments you can add them with @Argument annotation. For example:

```
    @LocalizedMessages(path = “core”)
    public enum MyMessages implements LocalizedMessageInterface
    {
        @LocalizedMessage(defaultMessage = “some text %1$s”)
        @MessageComment({
            “this is the first line”,
            “some other comment line”
        }, args = {
            @MessageCommand.Argument(“players display name”)
        })
        SomeText
    }
```

5.6.11 Predefined messages

All predefined messages are present in enumeration CommonMessages within the API.

5.6.11.1 Command handling messages

5.6.11.1.1 InvokeIngame

Command invocations may be written from players within chat or from console. Sometime you require that a player is executing a command.

Class AbstractCompositeCommand already has some example code to use it:

```
    command.when(c -> !c.isPlayer()).thenThrow(CommonMessages.InvokeIngame);
```

The API supports a more short variant of this code line:

```
    command.checkOnline();
```

5.6.11.1.2 InvokeOnConsole

Command invocations may be written from players within chat or from console. Sometime you require that only a server administrator on console is executing a command.

Class AbstractCompositeCommand already has some example code to use it:

```
    command.when(c -> c.isPlayer()).thenThrow(CommonMessages.InvokeOnConsole);
```

The API supports a more short variant of this code line:

```
    command.checkOffline();
```

5.6.11.1.3 NoPermissionForCommand
Use this message if a player has no permission to use a command. An example code fragment:

```
    if (!player.checkPermission(MyPermission.DoSomething))
        throw new McException(CommonMessages.NoPermissionForCommand,
            “/mycommand”);
```

The command interface has two shorter variants to check for permissions

```
    command.permThrowException(MyPermission.GreetMe, command.getCommandPath());
    command.permOpThrowException(MyPermission.GreetMe, command.getCommandPath());
```

The first line checks only for the permission and throws the exception on missing permission. The second line will check for operator flag too.

5.6.11.1.4 CommandDisabled
This message indicates that something is disabled.

```
    command.when(c -> myConfig.enabled.getBoolean()).
        thenThrow(CommonMessages.CommandDisabled, command.getCommandPath());
```

5.6.11.1.5 TooManyArguments
Usually commands take arguments. If you get too many arguments from user, you can use this message.

```
    if (command.getArgs().length > 2)
        command.send(CommonMessages.TooManyArguments);
```

or for a shorter variant use the following code:

```
    command.checkMaxArgCount(2);
```

5.6.11.2 Other messages
5.6.11.2.1 InternalError
If something goes terribly wrong, for example you caught an exception you normally do not get use this message. Sample code from exception handling:

```
    catch (Exception ex)
    {
        // do not forget to log this exception for stack trace etc.
        throw new McException(CommonMessages.InternalError, ex, ex.getMessage());
    }
```

5.6.12 Main locales

The MclibInterface supports registering of “main locales”. This is a list of locales your server likes to support. It is meant for information. The users can choose from their preferred language from the list of main languages and McLib helps you for translating messages (see McLib command description).

The main locales do not influence the API itself. Meaning that you can always use other Locales for messages, as default locale or for user locales programmatically.

5.6.13 Placeholders

Placeholders are part of messages that are resolved at runtime. You can think of them as kind of variables. Once a string is displayed (=encoded) the placeholders are replaced with concrete values. For example, you may use the following string in locale messages: “Hello {myplugin_level}!”.

And your module resolves the placeholder “myplugin_level” to the strings “rookie”, “master”, “chief” depending on the score of the current player.

5.6.13.1 Using placeholders

You can use placeholders in any localized string. During string encoding the placeholders are translated to their values.

Placeholders are using the following regex pattern:

```
[{]([\\p{L}\\p{Alnum}_-]+)[}]
```

You can manually resolve placeholders by invoking the MessageServiceInterface:

```
final String result = MessageServiceInterface.instance().replacePlaceholders(
    Locale.ENGLISH,
    “First placeholder is {myplugin_level} and second is {myplugin_name}!”);
```

5.6.13.2 Registering new placeholders

To register a custom placeholder simply invoke the registerPlaceholders method in your onEnable.

```
    MessageServiceInterface.instance().registerPlaceholders(
        this, “myplugin”, (locale, placeholder) -> {
            if (placeholder.length == 0) return null;
            switch (placerholder[0])
            {
                case “level”:
                    // do something
                    return “ROOKIE”;
                case “name”:
                    return MclibInterface.instance().getCurrentPlayer().getName();
            }
});
```

5.7 Command handling

McLib contains a small but powerful API for command handling. It helps you to pretty print your commands including help pages.

5.7.1 onCommand method

As usual override the “onCommand” method of your plugin. Use the following code snippet to handle the commands.

```
public boolean onCommand(CommandSender sender, Command command, String label,
String[] args)
{
    // maybe divide multiple commands if your plugin as more than one command.
    final CommandImpl cmd = new CommandImpl(
        sender, command, label, args,
        “/mycommand”);
    try
    {
        McLibInterface.instance().runInNewContext(() -> {
            McLibInterface.instance().setContext(McPlayerInterface.class,
                cmd.getPlayer());
            this.myCommand.handle(cmd);
        });
    }
    catch (McException ex)
    {
        cmd.send(e.getErrorMessage(), e.getArgs());
    }
    return true;
}
```

There is happening some magic. At first a CommandImpl is created. This is a helper class for the command handling. It contains some useful things like sending localized messages to player/console.

At second step, we invoke “runInNewContext”. Do not wonder about this step if you do not understand it yet. The context handling is explained later in its own chapter.

The important line is “this.myCommand.handle(cmd)”. Here we delegate the work to a specific command handler. The command handler can be a simple variable created in constructor or “onEnable”. I will now explain what type of command handlers can be used.

You may implement your own command handler of course.

To write less code you can simply use the static onCommand method:

```
public boolean onCommand(CommandSender sender, Command command, String label,
String[] args)
{
    CommandImpl.onCommand(sender, command, label, args, this.myCommand);
}
```

5.7.2 onTabComplete command

McLib requires tab completion and it is always a good help for players to support this in your command handlers.

Use the following snippet:

```
public List<String> onTabComplete(CommandSender sender, Command command, String
alias, String[] args)
{
    // maybe divide multiple commands if your plugin as more than one command.
    String lastArg = null;
    String[] newArgs = null;
    if (args.length > 0)
    {
        lastArg = args[args.length – 1].toLowerCase();
        newArgs = Arrays.copyOf(args, args.length – 1);
    }

    final CommandImpl cmd = new CommandImpl(
        sender, command, null, args,
        “/mycommand”);
    final String last = lastArg;
    try
    {
        McLibInterface.instance().runInNewContext(() -> {
            McLibInterface.instance().setContext(McPlayerInterface.class,
                cmd.getPlayer());
            return this.myCommand.onTabComplete(cmd, last);
        });
    }
    catch (McException ex)
    {
        cmd.send(e.getErrorMessage(), e.getArgs());
    }
    return Collections.emptyList();
}
```

To write less code you can simply use the static onCTabComplete method:

```
public List<String> onTabComplete(CommandSender sender, Command command, String
alias, String[] args)
{
    return CommandImpl.onTabComplete(sender, command, alias, args,
this.myCommand);
}
```

5.7.3 AbstractCompositeCommandHandler

Whenever you like to have sub commands inherit this class.

In Constructor, you can add your sub commands and in Usage method you can send a usage information to users that do not know how to use this command. The usage is displayed if either

- No command arguments are given
- An invalid sub command is given

Example constructor:

```
this.subCommands.put(“help”, new HelpCommandHandler(this));
this.subCommands.put(“foo”, new FooCommandHandler());
this.subCommands.put(“bar”, new BarCommandHandler());
```

Here we have three sub commands. The help sub command uses another default class from API: The HelpCommandHandler. The other two sub commands are your own classes. They implement SubCommandHandlerInterface.

5.7.4 SubCommandHandlerInterface

Building commands from sub commands requires you to create classes that implement this interface.

By default, all commands are always visible. You may override method “visible” to change this behavior. For example, let a command only be visible if a plyer has a permission. But remember to always check for the permission in method “handle” too. It increases your security doing this.

In method “handle” you place your regular code to handle the command, pass the arguments and do anything you like to do. Remember to notify the user about succeeding commands so he knows everything was ok. If you simple do something but not send any information to users they may be confused. You can always throw a McException to interrupt your work. The message text will be caught within your “onCommand” method of the Plugin class.

The method “onTabComplete” is required. If you do not have any completions for your command simple return “Collections.emptyList()”. Many commands take arguments, for example some text that can be auto completed. Pass the list of strings you allow at next. Notice: You should filter the result for all strings beginning with text “lastArg”.

The method “getShortDescription” is used by the help command. It displays a one line help text what this command does.

The method “getDescription” is meant for multi-line help texts. It is used by the help command too.

5.7.5 AbstractPageableCommandHandler

Many commands may display a variable list of entries. For example, a command to display the friends of a user.

Inherit this class to help you with paging.

There are three methods you need to implement. First of all, “getHeader”. This is used to buld a nice header. Return a message identifying the list or your command.

Within “getListCount” you will have to do a count on your lines. Returns the number of lines the command will return.

The method “getLines” will return the text of the actual lines, starting at index “start”.

That’s it. As you see you only need to concentrate how many lines you have and what lines to return. The rest is done by McLib.

5.7.6 LocalizedPagableCommand

This class is a special variant of AbstractPagableCommand. It displays multi line texts that may be spread over multiple pages.

You can use it for displaying manuals.

```
new LocalizedPageableCommand(Messages.MyManual, Messages.MyManualHeader);
```

5.7.7 Command argument handling

The CommandInterface contains helper methods for simple command argument parsing. The fetch methods can be invoked in order of the arguments you want to receive. Each invocation of a fetch method will remove the next argument from array.

5.7.7.1 fetchString

This method fetches a string at current position. If there is no argument left the method will throw McException with given arguments.

```
final String myName = command.fetchString(Messages.NameMissing);
```

5.7.7.2 fetch

The fetch method will return an object of any kind by using a helper method. There are two variants. The first one will directly return the object throwing an exception if there is no more argument.

```
class MyArgObject
{
    MyArgObject(CommandInterface cmd, String arg)
    {
        // ...
    }
}

final MyArgObject myArgObject = command.fetch(
    MyArgObject::new,
    Messages.MyArgMissing);
```

The second variant is used for optional arguments. It does not throw any exception if the argument is missing. Instead it returns a java “Optional”.

```
final Optional<MyArgObject> myArgObject = command.fetch(
    MyArgObject::new);
if (myArgObject.isPresent())
{
    // ...
}
```

5.7.7.3 *checkMinArgCount*

This method is used to check for minimum count of arguments. If there are not enough arguments, it throws exception with given message.

```
command.checkMinArgCount(2, MyMessages.NotEnoughArguments);
```

5.7.7.4 *checkMaxArgCount*

This method is used to check for maximum count of arguments. If there are too many arguments, it throws exception with given message.

```
command.checkMaxArgCount(2, MyMessages.TooManyArguments);
```

5.7.7.5 *Complete example*

Let us assume that we want to create a minigame arena. Actually, this is a live example from “/mg2 admin create” command. We take at least three arguments:

- Arena name
- Minigame name
- Arena type name

The code to handle the arguments can be something like that:

```
command.checkMaxArgCount(3, CommonMessages.TooManyArguments);
final String arenaName = command.fetchString(
    Messages.ArenaNameMissing,
    Messages.Usage);
final MinigameInterface minigame = command.fetch(
    this::getMinigame,
    Messages.MinigameNameMissing,
    Messages.Usage);
final ArenaTypeInterface type = command.fetch(
    (c, n) -> this.getType(c, n, minigame),
    Messages.ArenaTypeMissing,
    Messgaes.Usage);
```

The code first checks that we have not more than 3 arguments. Instead it throws an exception that will abort our command handler. Then it consumes the first argument (the arena name). Second it consumes the minigames name and then the arena types name.

Here are the helper methods:

```
private MinigameInterface getMinigame(CommandInterface cmd, String name)
    throws McException
{
    final MinigameInterface result = searchByName(name);
    if (result == null)
```

```
            throw new McException(Messages.MinigameNotFound, name);
        return result;
    }
    private ArenaTypeInterface getType(CommandInterface cmd, String name,
        MinigameInterface minigame) throws McException
    {
        final ArenaTypeInterface result = minigame.searchByName(name);
        if (result == null)
            throw new McException(Messages.ArenaTypeNotFound, name);
        return result;
    }
```

5.8 Context sensitive execution

In previous chapters, you already found out that there is a feature to run code within a context. This feature is a light weight option to store objects within the current execution context and get them back whenever you like. No need to pass the objects as method arguments.

The most common use case is to set the current user within this context. In other words: You say to the context: “Here is the user that is invoking everything.”

The McLibInterface (and some others) own the methods I am talking about. They are called “getContext” or “setContext”. The first one receives a context variable and the second one sets it.

5.8.1 Run bukkit tasks with contexts

5.8.1.1 Run tasks on new context

You can always use the existing way via BukkitScheduler or BukkitRunnable. Every time a task is run it has its own clean context.

There is no secret doing it this way.

5.8.1.2 Run tasks on copied context

Use the methods from McContext/ McLibInterface to copy the current context and run a task on the copy.

```
    // somewhere before
    final MyVar myvar = new MyVar();
    MclibInterface.instance().setContext(MyVar.class, myvar);

    // later on run a task at next server tick
    MclibInterface.instance().runTask(myPlugin, this::run);

    // the handler
    private void run(BukkitTask task)
    {
        final MyVar myvar = McLibInterface.instance().getContext(MyVar.class);
        // do anything you like
    }
```

You will find multiple methods to run tasks. They are similar to bukkits scheduler methods.

5.8.1.3 *Run tasks for players*

As second option, you will find methods within McPlayerInterface. They invoke a task on a clean context with only the player set as current player.

5.8.2 The default get implementations

5.8.2.1 *getCurrentCommand*

This method returns the CommandInterface for code running within commands. The current command is automatically set by “CommandImpl.onCommand” and “CommandImpl.onTabComplete”.

5.8.2.2 *getCurrentComponent / getCurrentObject / getCurrentZone / getCurrentEntity*

This method returns the current object for code running with McLib objects (components, zones, entities etc.). Mainly these are event handlers and the methods within the handler interfaces.

5.8.2.3 *getCurrentEvent*

This method returns the current event for code running inside events. This only work for McLib event handlers not for classic bukkit event handlers.

5.8.2.4 *getCurrentPlayer*

The current player is extracted from events as well as command handlers. It returns the player invoking a command or causing the event.

5.8.3 Pushing and resetting the context

Sometimes you want to manipulate the context and restore it once your code is finished. McLib supports four methods for this task (all in McLibInterface):

- calculateInNewContext / runInNewContext
- calculateInCopiedContext / runInCopiedContext

The methods starting with calculate are able to return an object (non-void). The methods starting with run are calling classic void methods.

They will either copy the current context and restore it on finish or start with a cleared context. Similar to the task run methods but the code will be invoked immediately and not within later tasks.

5.8.4 Context Handlers

A context handler supports calculating context variables from events or commands. They will be invoked as soon as an object is requested via “getContext” and the object is not yet known. You can use it to calculate your custom object without setting it directly.

Invoke “registerContextHandler” in your onEnable method to install your custom context handler.

5.9 Configuration Framework and DataFragments

5.9.1 Using configuration files

You need the following steps to use the configuration API:

1) Create an enumeration holding your configuration variables. Let the enumeration implement ConfigurationValueInterface.

2) Add the @ConfigurationValues annotation at enum type level.

3) Add enumeration values. Each enumeration value represents one entry in your configuration file.

4) Add the enumeration in your plugins “onEnable” method (see the chapter about enumerations).

Here is a simple example with a simple Boolean configuration option.

```
@ConfigurationValues(path = “core”)
public enum MyConfig implements ConfigurationValueInterface
{
    @ConfigurationBool(defaultValue = true)
    MyCommandEnabled
}
```

To access the configuration value simply invoke the corresponding getter:

```
MyConfig.MyCommandEnabled.getBoolean();
```

That’s it.

5.9.2 Changing configuration options

Each configuration value has corresponding setters. Simply invoke them and call save.

```
MyConfig.MyCommandEnabled.setBoolean(false);
MyConfig.MyCommandEnabled.saveConfig();
```

Hint: You can change multiple values and only call “saveConfig” on any of the value. It may be confusing that we put this method in the interface and that you invoke it on a single configuration value. But that was the only way to hide the complex implementation from your code. Simply keep in mind: Whenever calling “saveConfig” the whole config file will be saved.

5.9.3 Using multiple files

Per default everything goes to “config.yml” of your plugin. If you tend to split your config in multiple files you can have a look at the @ConfigurationValues annotation.

Override the file variable to declare a second configuration with some other name.

5.9.4 Valid data types

You will find tons of getters and setters. Here is a table of all variable types we support out-of-the-box.

Java type	Getter	Setter	Annotation class
boolean	getBoolean	setBoolean	ConfigurationBool
boolean[]	getBooleanList	setBooleanList	ConfigurationBoolList
byte	getByte	setByte	ConfigurationByte
byte[]	getByteList	setByteList	ConfigurationByteList
char	getCharacter	setCharacter	ConfigurationCharacter
char[]	getCharacterList	setCharacterList	ConfigurationCharacterList
ConfigColorData	getColor	setColor	ConfigurationColor
ConfigColorData[]	getColorList	setColorList	ConfigurationColorList
double	getDouble	setDouble	ConfigurationDouble
double[]	getDoubleList	setDoubleList	ConfigurationDoubleList
float	getFloat	setFloat	ConfigurationFloat
float[]	getFloatList	setFloatList	ConfigurationFloatList
int	getInt	setInt	ConfigurationInt
int[]	getIntList	setIntList	ConfigurationIntList
ConfigItemStackData	getItemStack	setItemStack	ConfigurationItemStack
ConfigItemStackData[]	getItemStackList	setItemStackList	ConfigurationItemStackList
long	getLong	setLong	ConfigurationLong
long[]	getLongList	setLongList	ConfigurationLongList
DataFragment	getObjectt	setObject	ConfigurationObject
DataFragment[]	getObjectList	setObjectList	ConfigurationObjectList
McPlayerInterface	getPlayer	setPlayerList	ConfigurationPlayer
McPlayerInterface[]	getPlayerList	setPlayerList	ConfigurationPlayerList
short	getShort	setShort	ConfigurationShort
short[]	getShortList	setShortList	ConfigurationShortList
String	getString	setString	ConfigurationString
String[]	getStringList	setStringList	ConfigurationStringList
ConfigVectorData	getVector	setVector	ConfigurationVector
ConfigVectorData[]	getVectorList	setVectorList	ConfigurationVectorList
EnumerationValue	getEnum	setEnum	ConfigurationEnum
EnumerationValue[]	getEnumList	setEnumList	ConfigurationEnumList
Enum	getJavaEnum	setJavaEnum	ConfigurationJavaEnum
Enum[]	getJavaEnumList	setJavaEnumList	ConfigurationJavaEnumList

5.9.5 DataFragment

The configuration framework is able to wrap complex objects to configuration files. If you want to create your own complex object you will have to implement the DataFragment interface.

5.9.5.1 *Implementing DataFragment*

A DataFragment is made of three methods:

- read
- write
- test

The read method is responsible to read the data fragment from given DataSection. The write method stores the data into the given DataSection.

The test method will be called to check if the DataSection is valid and holds the attributes to read.

You can assume that the DataSection is only used by this single DataFragment. There are no naming conflicts. Simply read and write to the section by using the keys you like.

5.9.5.2 *Using Plan objects with AnnotatedDataFragment*

There is a default implementation you can use. It is very easy to handle. Simply inherit AnnotatedDataFragment and tag your fields with annotation @PersistentField. The class does all the magic. See the following example as a starting point:

```
public class MyConfigObject extends AnnotatedDataFragment
{
    @PersistentField
    protected final int i;

    public MyConfigObject()
    {
        this.i = -1;
    }

    public MyConfigObject(int i)
    {
        this.i = i;
    }

    public int getI()
    {
        return this.i;
    }

}
```

Your data fields should be protected. If you decide to use inheritance the protected keyword prevents naming conflicts.

You always need a constructor without arguments is needed by the framework. Reading objects from config will always use this constructor.

The following data types are allowed within AnnotatedDataFragment (primitives are allowed as java primitives or as Object; “lower Cased” boolean as well as “upper Cased” Boolean).

String	Boolean	Byte	Short

Integer	Long	Float	Double
LocalDateTime	LocalDate	LocalTime	Character
VectorDataFragment	PlayerDataFragment	ItemStackDataFragment	ColorDataFragment
DataSection *)	List / Set **)	Map **)	DataFragment
EnumerationValue			

*) DataSection is a wrapper around any kind of unstructured data.

**) Lists of DataFragment or any of the “java primitives” (including String, LocalDate/Time). Maps can hold string keys and as values any DataFragment or “java primitive”.

Hint: The java serialization ensures that the same instance is not serialized two times. But McLib does NOT work in this way. If the same object instance is used in several attributes and you store the object to config it will break up the object instances into multiple ones. This means: Re-reading the object you will have multiple instances of this objects.

5.9.5.3 Inheritance and versions

McLib does not directly support inheritance or version systems, nor does it hold hints which object type you are storing to database. If you have two classes, both having an int field called “i” it is possible for the framework to read both of them. The caller of getDataFragment decides which class to use.

If you require hints which object type is stored in config you will have to store this hint in a separate field.

The same solution should be used for versions. Maybe one day you are extending your plugin. You decide to introduce new fields into an existing DataFragment. Doing this you should always store a version number in the config. That you are able to detect which version of your DataFragment was stored into config.

5.9.6 ConfigurationSection and sub path handling

Configuration values can be treated as sections to hold multiple values. To declare such sections, use the @ConfigurationSection annotation.

```
@ConfigurationValues(path = “core”)
public enum MyConfig implements ConfigurationValueInterface
{
    @ConfigurationSection
    FullName
}
```

You can know use the methods having a “subpath” parameter to receive and store multiple values.

```
final String forname = MyConfig.FullName.getString(“forname”, “Some default”);
final String surname = MyConfig.FullName.getString(“surname”, “default2”);
```

5.9.7 File comments

Using file comments on configuration enums is easy. Simply add the @ConfigComment annotation. It takes a string list with each string representing a single line.

Here is an example how to use it:

```
@ConfigurationValues(path = “core”)
public enum MyConfig implements ConfigurationValueInterface
{
    @ConfigurationBool(defaultValue = true)
    @ConfigComment({
        “this is the first line”,
        “some other comment line”
    })
    MyCommandEnabled
}
```

The comment will be placed into the configuration file if there is no existing comment. You can add the comments to describe valid values and the meaning of your configuration value. This helps administrators editing your files.

A second option is to check the DataSection (f.e. in your own custom DataFragment implementation). If the DataSection is implementing CommentableDataSection you can cast it and use the methods “setSectionComments” or “setValueComments” to set a comment within your configuration file.

5.9.8 Value validation

Configuration enumerations support validation through several annotations.

Simply add one or more annotations to your configuration enum and once you edit a configuration value you can invoke Method validate to check for invalid values.

```
MyConfig.MyValue.setInt(55);
try
{
    MyConfig.MyValue.validate();
    MyConfig.MyValue.saveConfig();
}
catch (McException ex)
{
    // will revert the changes made to MyValue
    MyConfig.MyValue.rollbackConfig();
}
```

After doing multiple changes you can invoke verifyConfig for checking all enumeration values in the same config file.

5.9.8.1 ValidateStrMin / ValidateStrMax

Topic: Verifies for minimum or maximum string length.

Types: String, StringList

5.9.8.2 ValidateLMin / ValidateLMax

Topic: Verifies for minimum or maximum values of numeric types.

Types: Byte, ByteList, Short, ShortList, Int, IntList, Long, LongList

5.9.8.3 ValidateFMin / ValidateFMax

Topic: Verifies for minimum or maximum values of floating point types.

Types: Float, FloatList, Double, DoubleList

5.9.8.4 ValidateStrMin / ValidateStrMax

Topic: Verifies for minimum or maximum string length.

Types: String, StringList

5.9.8.5 ValidateListMin / ValidateListMax

Topic: Verifies for number of list elements.

Types: Any list type

5.9.8.6 ValidateIsset

Topic: Verifies for required values

Types: Any type

5.9.8.7 Validator

Topic: Custom verification class.

Types: Any type

5.9.9 Directly loading/writing yml files

You can directly load DataSections from yml files. Simply pass a java.io.File or java.io.InputStream to “readYmlFile” of McLibInterface.

The resulting data section will support comments. See previous chapter for details.

After manipulating the contents, you can save the config file via “saveYmlFile”.

5.10 Storages

A storage is a light weight option to save and receive values with associated objects. A simple example is found in the McPlayerInterface class. Look at the method getPersistentStorage. It returns an McStorage object.

Here you find a getter and a setter. The meaning of these methods should be clear.

The only limitation is that you can only store exactly one object per class within a single storage. For storing your information first create your DataFragment implementation, for example by using AnnotatedDataFragment (see chapter 5.9.5.2 Using Plan objects with AnnotatedDataFragment for details).

You can use the storage in your custom code:

```
final McPlayerInterface player = -.....;
final MyDataFragment data = .....;
player.getPersistentStorage().set(MyDataFragment.class, data);

// do whatever you like

final MyDataFragment data = player.getPersistentStorage().
    get(MyDataFragment.class);
if (data != null)
{
    // do whatever you like
}
```

Hint: The Storage is really meant to be light weight. Do not store tons of data in it. This may slow down the whole system.

Currently the following storages are known:

5.10.1 McPlayerInterface.getPersistentStorage

This storage is persistent. You can store data that needs to be available even during server restarts. The storage may be present in data files in the McLib configuration folder or maybe present in databases. This depends on configuration of McLib.

As a plugin writer, you currently do not care about the location or system the storage is stored to.

5.10.2 McPlayerInterface.getSessionStorage

The session storage holds data in memory. It is valid till the player goes offline. You can use it for data only needed during online sessions.

Hint: Teleporting players over bungee networks will cause them being offline of the source server. Remember that the data becomes invalid. It is not yet possible to catch the “goes offline from bungee server” event to invalidate your data. If you need to store data even if the player only switches to a new server within the same network, you will currently have to use the persistent storage.

5.10.3 McPlayerInterface.getContextStorage

The context storage holds data during the current execution context. The context feature is explained in chapter 5.8 Context sensitive execution.

The data is lost as soon as the context is left. For example, execution of a command is wrapped into his own execution context. Using the context storage allows you to set data only visible during this command execution.

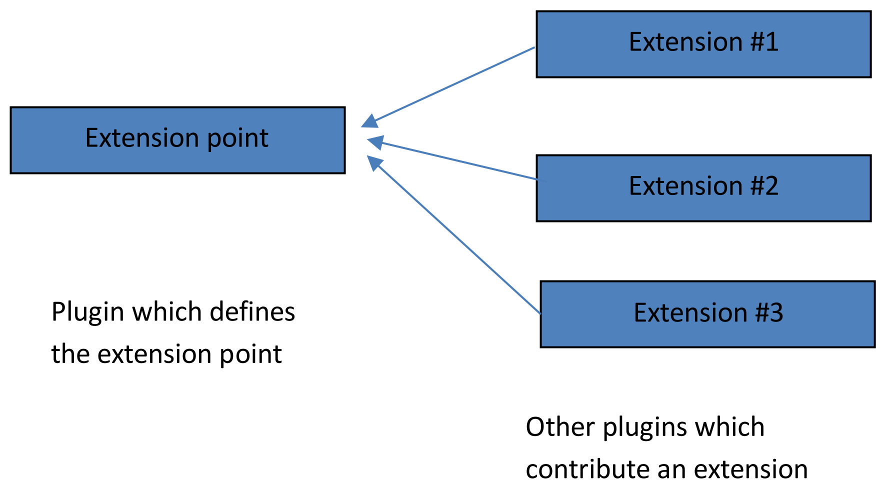
5.10.4 GuiSessionInterface.getGuiStorage

This storage is a special in memory storage. It is valid during gui sessions. In other words: As soon as the gui is closed or the user is going offline, this storage will be cleared up.

This storage can be useful if you want to create complex guis menus that need to hold additional information.

5.11 Extension handling

Extensions are special code fragments injected into existing plugin code.


Extension #1
Extension point
Extension #2
Extension #3
Plugin which defines the extension point
Other plugins which contribute an extension

5.11.1 Creating an extension point.

First create an interface providing your API. Let us assume you want to allow extensions to manipulate a special GUI by creating clickable icons.

```
public interface MyGuiExtension extends ExtensionInterface
{
    ClickGuiItem[] getGuiItems(MyObject object);
}
```

After creating the API interface, you need to have an extension point object. I suggest to put it into the same interface.

```
    public interface MyGuiExtension extends ExtensionInterface
    {

        ExtensionPointInterface<MyGuiExtension> POINT = new
ExtensionPointInterface<MyGuiExtension>(){
            public String name()
            {
                return “myplugin/mygui”;
            }
            public Class<MyGuiExtension> extensionClass()
            {
                return MyGuiExtension.class;
            }
        };

        ClickGuiItem[] getGuiItems(MyObject object);
    }
```

5.11.2 Adding extensions to extension points

Other plugins can contribute their extensions by using the extension service interface during onEnable.

```
    final MyGuiExtension extension  = new ...
    ExtensionServiceInterface.instance().register(
        this,
        myGuiExtension.POINT,
        extension);
```

5.11.3 Accessing extensions

Being in your code that creates the gui you can do the following to access the extensions:

```
    MyObject myobject = ...
    for (final MyGuiExtension ext : MyGuiExtension.POINT.getExtensions())
    {
        for (final ClickGuiItem item : ext.getGuiItems(myobject))
        {
            // do something with this item
        }
    }
```

5.11.4 Extensions vs. Bukkit events

Now we have two ways to communicate with other plugins. First the “classic” bukkit events and second the extension points.

In which situation do we prefer extensions? The answer is not easy. Both do their work and both can be used to communicate with other plugins.

However, the bukkit events should be used for small use-cases and specially if there is only small data to be transported between plugins. For example: Player Join Event. The event is small, it has only one use-case and the only data to be transported back is the join message.

Inventory click events are more difficult. They sum up many use cases and it is not very intuitive what attributes are filled. This is not a flame against the bukkit developers this event may be smarted if handled by extension concepts.

So, let us sum up when to prefer extensions:

- If you need to transport multiple data objects
- If you have multiple use cases (= multiple methods in your API) to be supported
- If you do not need priorities etc.

When to prefer bukkit events:

- If you talk to extensions not knowing anything about mclib
- If you have single use cases with clear API
- If you want to use the event stuff (priorities, monitoring, async invocations)

5.12 Simple GUI

The simple gui is a wrapper around nms classes and Minecraft builtin features to build simple in game guis. The GUI is limited and not that smart but it works for the most common solutions.

However, the most important thing is that the simple gui even works if the user does not have any other client mods. You should always choose between the several gui options carefully. Not all users like to install client mods to use your plugin.

5.12.1 Open a click gui (inventory)

First of all, you will have to create a unique id for your gui. If you have multiple guis you will need to separate them.

So, let us create an enumeration and register it within your plugins onEnable method.

```
public enum MyClickGuis implements ClickGuiId
{
    MySampleGui
}

// somewhere in onEnable
EnumServiceInterface.instance().registerEnumClass(this, MyClickGuis.class);
```

The next step is to implement ClickGuiInterface. You will have to return number of lines, gui id and the first initial page. The number of lines must be a value between 1 und 6. The simplest way is to use the default implementation SimpleClickGui and SimpleClickPage. Here is a code example:

```
final ClickGuiInterface gui = new SimpleClickGui(
    MyGlickGuis.MySampleGui,
    new SimpleClickPage(MyMessages.MyClickGuiName, 3)
        .setItem(0, 0, someItem)
        .setItem(1, 0, secondRowItem)
        .setItem(2, 0, thirdRowLeftItem)
        .setItem(2, 8, thirdRowRightItem),
    3);
player.openClickGui(gui);
```

The items are made of ClickGuiItem objects. They contain an item stack for representation, a localized message for display title and a handler to invoke if the player clicks the item.

A simple example to echo and close the gui:

```
    final ClickGuiItem thirdRowItem = new ClickGuiItem(
        new ItemStack(Material.REDSTONE_BLOCK),
        MyMessages.GreetAndCloseGui,
        (p, s, g) -> {
            .p.sendMessage(MyMessages.Greet);
             s.close();
        });
```

5.12.1.1 Pagable click guis

A pagable click gui supports displaying list of elements. It contains a header line with previous and next icons to display other pages.

It can be used as a base class for any click gui that wants to display a variable size list.

Here is a code example:

```
    public class MyPage extends PagableClickGuiPage<MyObject>
    {
        public Serializable getPageName()
        {
            return MyPageMessages.PageName;
        }

        protected int count()
        {
            // typically you query the database for total number of elements
            return numberOfElements;
        }

        protected List<MyObject> getElements(int start, int limit)
        {
            // query the database for your elements
            // starting with “start” index
            // and with maximum of “limit” elements
            return listOfElements;
        }

        protected ClickGuiItem map(int line, int col, int index, MyObject elm)
        {
            return new ClickGuiItem(...);
        }

        protected ClickGuiItem[] firstLine()
        {
            return new ClickGuiItem[]{
                this.itemPrevPage(),
                this.itemNextPage()
            };
        }
    }
```

5.12.1.2 Recommended layout of first line

It is recommended to have a common layout for the first line that users are not confused.

The first line will be made of the following items. It an item is useless you should null/remove it but without changing the position of the other items.

1. Home-Button, see GuiServiceInterface.itemHome
2. Refresh-Button, see GuiServiceInterface.itemRefresh
3. Prev Button (for paging elements), see GuiServiceInterface.itemPrevPage
4. Next Button (for paging elements), see GuiServiceInterface.itemNextPage
5. Back Button to display previous gui page, see GuiServiceInterface.itemBack
6. Create Button, see GuiServiceInterface.itemNew
7. Delete Button, see GuiServiceInterface.itemDelete
8. Search Button, see GuiServiceInterface.itemSearch
9. Close gui Button, see GuiServiceInterface.itemCloseGui

5.12.2 Open Anvil Gui (text input)

Similar to ClickGui we will first create an enumeration to identify our text input.

```
    public enum MyAnvilGuis implements AnvilGuiId
    {
        MySampleAnvil
    }

    // somewhere in onEnable
    EnumServiceInterface.instance().registerEnumClass(this, MyAnvilGuis.class);
```

Second step is to create the anvil gui itself. We need to implement AnvilGuiInterface. Similar to the click gui there is a simple class you can use: SimpleAnvilGui.

Here is an example to fetch an email address:

```
    final ItemStack stack = new ItemStack(Material.NAME_TAG);
    final ItemMeta meta = stack.getItemMeta();
```

```
meta.setDisplayName(player.encodeMessage(MyMessages.EnterEmailAdress)[0]);
stack.setItemMeta(meta);
    final SimpleAnvilGui gui = new SimpleAnvilGui(
    MyAnvilGuis.MySampleAnvil,
    stack,
    str -> checkAndUse(str),
    () -> doSomethingOnCancel()
    );
player.openAnvilGui(gui);
```

The method doSomethingOnCaancel is invoked if the player does not finish the input. For example, he pressed ESCAPE. It is optional.

The method checkAndUse gets the user input and checks it. For our email check, it could be the following code:

```
public void checkAndUse(String email) throws McException
{
    if (!EmailValidation.isValidEmail(email))
    {
        throw new McException(MyMessages.InvalidEmail);
    }
    // do something nice with the email here
}
```

Hint: The anvil gui will remain open if you throw an exception because exceptions indicate errors or invalid input. If you do not throw an exception the anvil gui will close.

5.12.3 Raw messages

Although not being a regular “gui” you can use the chat console of the player. The McPlayerInterface contains method “sendRaw”. It takes a RawMessage that can be used for many different things.

To construct a RawMessage simply invoke the constructor and use one or multiple of the add methods. Notice: The first parameter (message) must always be a single line message.

5.12.3.1 addMsg: add preformatted text

To add simple or preformatted text you can use the addMsg method. It will add the message text as it would appear directly sending it to the player.

5.12.3.2 addHover: add a text hover

You can add a text showing a tooltip as soon as the mouse cursor is over it.

5.12.3.3 addCommand: add a clickable text invoking a command

Clicking the given text will invoke a command. Either the command is written to the chat console that a player can influence it or it is invoked directly.

5.12.3.4 addHandler: add a clickable text invoking code

A special variant of clickable command is to directly invoke your command handler/ java code. This is useful if you do not have chat command handlers you want to invoke but if you directly invoke code. McLib will generate a unique handler id and tunnel the event through its own chat commands.

You should be aware that the player may click the text hours later (as long as he stays online). Therefor the method gets a timestamp when the handler is expired and will be cleared.

5.12.3.5 addUrl: add a clickable url

This method adds a clickable url to open in player's favorite browser.

5.12.3.6 addTargetSelection:influence player's target

Let's the player select a target once he clicks on the text.

The method can take players as well as entities.

5.12.3.7 addClickableHover

This method combines addHover and addCommand/addHandler. It will both, display a hover if mouse cursor enters the text and invoke a command/ handler if the plyer clicks it.

5.12.3.8 Add: Raw json text

You can always add raw json components. Hiwever this manual does not cover how the json texts are built.

5.12.4 Gui service

The Gui service interface supports several helper methods for building nice Gui.

First of all, there are several icons you can use in your gui. See chapter 5.12.1.2 Recommended layout of first line for details.

5.12.4.1 Text editor

You can open or nest a text editor to get text input. It simply opens an anvil gui and is a fast way with less code.

```
GuiServiceInterface.instance().openTextEditor(
    myPlayer,
    “input-string”,
    this::onCancel,
    this::onInput,
    false,
    MyMessages.DescriptionStringInput);
```

5.12.4.2 Localization editors

The ConfigServiceInterface supports a gui to edit localized messages. Either single line messages or multi line messages. The messages are of class LocalizedConfigLine or LocalizedConfigString. You can create a clickable icon for your gui to start the editor:

```
final ClickGuiItem item =
    ConfigServiceInterface.instance().createGuiEditorItem(
        MyMessages.EdittextTitle,
        myEditableText,
        this::doSave,
        this::onRefresh,
        this::onHomepage);
```

5.12.4.3 Configuration editor

The second way for providing fast value editors is to display an icon for any EditableValue. Each Configuration value is automatically implementing EditableValue interface. If you implement EditableDataFragment instead of AnnotatedDataFragment you can use its values for configuration editors too.

```
final ClickGuiItem item =
    ConfigServiceInterface.instance().createGuiEditorItem(
        myEditableValue,
        this::doSave,
        this::onRefresh,
        this::onHomepage);
```

5.13 Custom items and custom blocks

In many situations, you may need custom items. For example, a good in game gui should support nice icons for friend lists, profile settings etc.

McLib supports various icons for in-game GUI as well as custom items and blocks. The following chapter is a MUST READ!

5.13.1 IMPORTANT! (Modding prolog)

The GUI items are working with resource packs. There is no need to install client addons. Custom blocks are only working with Mclib client plugins (for example the forge plugin).

5.13.1.1 Always use a lobby without custom items/ blocks

You should always have some kind of lobby where players join without seeing any custom items or blocks because a client without Mclib client plugin may cash!

The best way is to create a lobby and teleport a player after clicking a sign or npc and after validating that they have installed the Mclib client.

Use the following code to check:

```
if (!player.hasSmartGui())
    throw new McExeption(MyMessages.PleaseInstallClient);
if (!ResourceServiceInterface.instance().hasResourcePack(player))
    throw new McException(MyMessages.DownloadResourcePack);
```

5.13.1.2 Using items in GUIs is safe

A typical click gui only uses items in anonymous inventories. They do not allow to put the items into player inventories. So they are not “persistent” in any way. Using custom items for GUIs is always safe.

Simply check if the user installed your custom resource pack and deny if the user declines server resource packs.

```
if (!ResourceServiceInterface.instance().hasResourcePack(player))
    throw new McException(MyMessages.DownloadResourcePack);
```

You may use any gui items even in your lobby world because checking for resource pack before opening the gui is safe. If the user denies downloading the resource pack he cannot use the gui.

5.13.1.3 Using persistent items and custom blocks

If you decide to use persistent items or even custom blocks you will are about to create a “modified” world. This means you create a world that clients and even other spigot plugins may not understand.

Notice: A modified world cannot be used without mclib. We introduce our own block and item ids that unmodified spigot servers or classic vanilla servers never understand. In every modified world there is an important section in config.yml of mclib plugin. It holds ids of custom blocks and items.

You MUST use only a single setup of config.yml in your whole bungee cord network so that all your blocks and items are using the same IDs. Introducing new plugins or new items/ blocks in existing plugins will require the following steps:

- Shut down all servers
- Install the new plugin or upgrade existing plugin jar
- Start on a single existing modified world and with existing config.yml
- Wait the server to start and to create the resource zip files
- Stop the server and upload the resource zip files to your webspace
- Copy the modified config.yml to your whole bungee cord network (to every server) and install the new plugins or upgrade the plugin jars
- Restart the servers (result: the all use the same item and block setup)

5.13.2 Creating items

Creating items is very easy. Use the ItemServiceInterface and request the item via “createItem” method.

```
final ItemStack euroSign = ItemServiceInterface.instance().createItem(
    CommonItems.App_Euro,
    “MONEY”);
```

Do whatever you like with this item. You will find many items in CommonItems enumeration.

The Icons are all taken from free icon sets of http://www.aha-soft.com/

Many thanks to them for providing parts of their work for free.

5.13.3 Resource pack

The custom icons are created with help of resource packs. So, every user needs to download a custom resource pack.

If you already have custom resource packs for your server, you need to do some work manually.

Right after server startup McLib will create a file called “mclib_core_resources_v1.zip” up to “mclib_core_resources_v3.zip” within the plugins folder. These are multiple versions of the resource packs needed by Minecraft clients. Copy it to your own web server. You will have to set the downloadUrl to your web server (ResourceServiceInterface.setDownloadUrl or via config).

That’s it. You can now choose your favorite download method. Either force every player to automatically download on login by setting ItemServiceInterface.setAutoResourceDownload to true. Or choose to download as soon as you need it, for example by opening a GUI.

Forcing downloads on demand can be done by code:

```
Status status = ResourceServiceInstance.instance().getState(player);
if (state == null)
{
    ResourceServiceInstance.instance().forceDownload(player, this::openGui);
}
else if (state == Status.SUCCESSFULLY_LOADED)
{
    this.openGui();
}
else
{
    throw new McException(Messages.DownloadResourcePack);
}
```

5.13.4 Creating your own icons

Creating your own icons is easy. Create and register your own enumeration that implements ItemId interface.

Every enumeration value needs ItemData annotation. The ItemData annotation contains classpath to your resource textures as well as a json string containing your item model.

Sample:

```
public enum MyItems implements ItemId
{
    @ItemData(
        textures = “my/package/textures/MyTexture.png”,
        modelJson = “{\”parent\”: \”item/handheld\”,\”textures\”:{\”layer0\”:
\”%1$s\”}}”)
    )
    MyItem
}
```

For details on model json files and textures see your favorite tutorial on writing resource packs.

5.13.4.1 Items with default names.

To setup default names for your items you can set the name attribute in ItemData annotation:

```
@ItemData(... name = MyItemName.class)

public class MyItemName implements NameProvider
{
    public LocalizedMessageInterface getName()
    {
        return MyMessages.MyItemName;
    }
}
```

This will setup a name and store it into the server resources so that clients download the right default name.

5.13.5 Tooling items

Tooling items are special helper for selecting blocks. The tooling item is put into players inventory and once a block is clicked a special handler is invoked. The item is removed if a player goes offline.

The following sample creates a pinion:

```
ItemServiceInterface.instance().prepareTool(
    CommonItems.App_Pinion, player, MyMessages.SelectSomeBlock)
    .onLeftClick(this::onLeftClick)
    .onRightClick(this::onRightClick)
    .singleUse()
    .build();
```

The single use method ensures that the tooling can only be used once. After using it the tooling is destroyed and removed from players’ inventory.

The click handler are simple methods receiving the player and the McPlayerInteractEvent.

```
private void onLeftClick(McPlayerInterface player, McPlayerInteractEvent evt)
{
    System.out.println(“leftclick:” + evt.getBukkitEvent.getClickedBlock());
}

private void onRightClick(McPlayerInterface player, McPlayerInteractEvent evt)
{
    System.out.println(“rightclick:” + evt.getBukkitEvent.getClickedBlock());
}
```

5.13.6 Custom blocks

For custom blocks, you will have to create an interface implementing BlockId. You can create single variant blocks and blocks with multiple variants.

5.13.6.1 Simple blocks

The following example adds a new type of ore:

```
    public enum MyBlocks implements BlockId
    {
        @BlockData
        @BlockVariantData(
            textures = “my/package/textures/MyOre.png”,
            modelJson = “{\”parent\”: \”block/cube_all\”,\”textures\”:{\”all\”:
\”%1$s\”}}”)
        FantasiumOre
    }
```

To create a block within the world you can use the BlockServiceInterface:

```
    BlockServiceInterface.instance().setBlockData(
        bukkitBlock,
        MyBlocks.FantasiumOre,
        BlockData.CustomVariantId.DEFAULT);
```

Or you can create items:

```
    final ItemStack stack = BlockServiceInterface.instance().createItem(
        MyBlocks.FantasiumOre,
        BlockData.CustomVariantId.DEFAULT,
        MyMessages.FantasiumOre);
```

5.13.6.2 Block with multiple variants

A block can be divided into up to 16 variants. Typical scenarios from classic Minecraft:

- Planks are divided by wood type: oak planks, spruce planks, birch planks etc.
- Carpets are divided by color: white carpet, orange carpet etc.

You can create block variants by additional enumeration. Here is a sample code to introduce multiple ore types:

```
    @ChildEnum({MyBlocks.OreVariants.class})
    public enum MyBlocks implements BlockId
    {
        @BlockData(variants = OreVariants.class
        CustomOre;

        public enum OreVariants implements BlockVariantId
        {
            @BlockVariantData(
                textures = “my/package/textures/MyFantasiumOre.png”,
                modelJson = “{\”parent\”:
\”block/cube_all\”,\”textures\”:{\”all\”: \”%1$s\”}}”)
            FantasiumOre,

            @BlockVariantData(
                textures = “my/package/textures/MyGeneriumOre.png”,
                modelJson = “{\”parent\”:
\”block/cube_all\”,\”textures\”:{\”all\”: \”%1$s\”}}”)
```

```
            GeneriumOre,
        }
    }
```

Now you have a new block (CustomOre) having two variants. Each variant has its own texture.

5.14 Permissions

You can create permissions by using McLib enumerations. McLib currently only allows simple yes/no permissions.

5.14.1 Creating and using permissions enumeration

First of all, we need to create an enumeration implementing PermissionsInterface and annotated with @Permissions.

```
    @Permissions(“myplugin”)
    public enum MyPermissions implements PermissionsInterface
    {
        @Permission
        Admin
    }

    // somewhere in onEnable
    EnumServiceInterface.instance().registerEnumClass(this, MyPermissions.class);
```

The McPlayerInterface has a simple Method called checkPermission. It returns true if the player has the permission.

5.15 Objects framework

McLib handles many different instrumentations of Minecraft “things”. We call them objects. Objects can listen for events they are involved in. For example, a block can listen for a break event.

5.15.1 Components

Components are used for blocks or more generally for locations. You can think of instrumented blocks.

In the following example, we will create a simple warp utility.

5.15.1.1 Create a component type.

First, we need a component type. As you might suppose we will create an enumeration and register it during onEnable.

```
    public enum MyComponents implements ComponentTypeId
    {
        Warp
    }
```

```
// somewhere in onEnable
EnumServiceInterface.instance().registerEnumClass(this, MyComponents.class);
```

The component type is only a marker for McLib to identify the component as a warp point.

5.15.1.2 Register a component handler during startup

Second step is to register a handler.

```
// somewhere in onEnable
EnumServiceInterface.instance().registerEnumClass(this, MyComponents.class);
ObjectServiceInterface.instance().register(MyComponents.Warp,
   WarpPoint.class);
```

The WarpPoint class implements ComponentHandlerInterface. This class is instantiated for each warp point or in other words for each single component of type “Warp”.

The handler can hold custom data if we need it. For us this is a string with the warp points name. We extend AnnotatedDataFragment and add a name field:

```
public class WarpPoint extends AnnotatedDataFragment
    implements ComponentHandlerInterface
{
    @PersistentField
    private String name;

    public WarpPoint()
    {
    }

    public WarpPoint(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return this.name;
    }
}
```

Now we have a warp point that is able to persist its name. Let’s go on and implement the needed methods. At first, we implement onCreate. This method is called as soon as the warp point is created by our chat command handler later on. We will save the component id for later use.

```
    private ComponentIdInterface id;

    @Override
    public void onCreate(ComponentInterface component)
    {
        this.id = component;
    }
```

Same code will be placed in onResume. This method will be called if the server starts and the already existing warp points are read from McLibs storage.

```
    @Override
    public void onResume(ComponentInterface component)
    {
        this.id = component;
    }
```

Now let us focus on object deletion. For warp points, there is no logic we need. If they are deleted it is fine for us. But there may be other components that we do not want to delete without checking some constraints. For example, MinigamesLib only allows deletion of objects if the owning arena is in maintenance mode. The check for arena state is placed in “canDelete” method.

```
    @Override
    public void canDelete() throws McException
    {
        // throw McException if constraints are violated
    }

    @Override
    public void onDelete()
    {
        // perform additional cleanup if needed
    }
```

Same logic goes with location changes. There is a method to check for constraint violations (canChangeLocation) and a method to perform additional cleanup after the location was changed.

```
    @Override
    public void canChangeLocation(Location newValue) throws McException
    {
        // throw McException if constraints are violated
    }

    @Override
    public void onLocationChange(Location newValue)
    {
        // perform additional cleanup if needed
    }
```

The last method is onPause. It will be invoked if your plugin gets disabled. You may do some cleanup, for example cancel tasks you started with onCreate/onResume.

```
@Override
public void onPause()
{
    // perform additional cleanup if needed
}
```

Now let us add some utility methods for a handy use.

```
public ComponentInterface getComponent()
{
    return ObjectServicesInterface.instance().findComponent(this.id);
}

public static WarpPoint byName(String name)
{
    final Collection<ComponentInterface> warps =
        ObjectServiceInterface.instance().findComponents(MyComponents.Warp));
    final Optional<WarpPoint> warp = warps.stream()
        .map(w -> (WarpPoint) w.getHandler())
        .filter(w -> name.equals(w.getName())
        .findFirst();
    return warp.isPresent() ? warp.get() : null;
}
```

Hint: Warp points are only markers in the meaning that it is always safe to create other components on the same location. But sometimes you do not like it. You can override method “checkCreation” and throw an exception if you do not like the creation of other components on the same location. Remember to check if there is already another component at same location in your command handler (see below) as well.

5.15.1.3 Resume objects during startup

We have to tell McLib when to finish object loading for our plugin. After we registered all enumerations and handlers we will invoke resumeObjects:

```
// somewhere in onEnable
EnumServiceInterface.instance().registerEnumClass(this, MyComponents.class);
ObjectServiceInterface.instance().register(MyComponents.Warp,
   WarpPoint.class);
ObjectServiceInterface.instance().resumeObjects(this);
```

The method resumeObjects will load McLibs storage and it will try to create the handlers. However, there may be things going terribly wrong. The method resumeObjects will NOT throw exceptions. Instead it will return a ResumeReport. Why? With resumeReport you may investigate what went wrong. You can decide to accept it or to drop the failing components/ stop your plugin.

5.15.1.4 Build a command handler

We now create a command handler for three commands:

- The use sub command will teleport the player to a warp by name.
- The add sub command will create a new warp point at the current player position.
- The remove sub command will delete an existing warp point by name.

We will use base class AbstractCompositeCommandHandler.

```
public class WarpCommand extends AbstractCompositeCommandHandler
{
    public WarpCommand()
    {
        this.subCommands.put(“help”, new HelpCommandHandler(this));
        this.subCommands.put(“add”, new WarpAddCommand());
        this.subCommands.put(“remove”, new WarpRemoveCommand());
        this.subCommands.put(“use”, new WarpUseCommand());
    }

    @Override
    protected void sendUsage(CommandInterface cmd)
    {
        cmd.send(MyMessages.WarpUsage);
    }
}
```

In the following chapter, we will implement the three commands add, remove and use.

5.15.1.5 Create “addWarp” command

Adding a warp takes one argument: the name of the warp. First of all let us add method “visibility” for security reasons.

```
public class WarpAddCommand implements SubCommandHandlerInterface
{
    @Override
    public boolean visible(CommandInterface command)
    {
        return command.isPlayer() ? command.isOp() ||
            command.getPlayer.checkPermissions(MyPermissions.AddWarp) : false;
    }
}
```

This method makes the command invisible on program console because we cannot add warps without having an online player’s location. The second check will go for a permission. Only operators or support people should be able to add new warps.

Now let us create the handle method. First it checks for security and then for correct arguments. It the arguments are passed it will check for duplicate names.

```
@Override
public void handle(CommandInterface command) throws McException
{
    // not valid on console
    command.when(CommandInterface::isPlayer)._elseThrow(
        MyMessages.NoConsole);
    // check for permission or operator flag
    command.permOpThrowException(MyPermissions.AddWarp, “add-warp”);

    if (command.getArgs().length != 1)
        throw new McException(MyMessages.MissingWarpName);
    final String name = command.getArgs()[0].trim();
    if (name.length() < 3)
        throw new McException(MyMessages.WarpNameTooShort);

    // check for duplicate name
    final WarpPoint existing = WarpPoint.forName(name);
    if (existing != null)
        throw new McException(MyMessages.WarpExists, name);

    // create it
    final Location l = command.getPlayer().getBukkitPlayer().getLocation();
    final WarpPoint warp = new WarpPoint(name);
    ObjectServicesInterface.instance().createComponent(
        MyComponents.Warp,
        l,
        warp,
        true);

    // success
    command.send(WarpCreated, name);
}
```

As you can see creating a component is easy. You need a location (a single block) and the handler class we created before. Notice the true flag at the end of createComponent invocation. This flag makes the warp point persistent, meaning to be stored in McLibs storage. If you have your own storage set this to false but remember to reload and recreate your warp points during onEnable manually.

You will have to implement more methods.

```
@Override
public List<String> onTabComplete(CommandInterface command, String lastArg)
{
    // add Warp has no tab completion
    return Collections.emptyList();
}
```

```
@Override
public LocalizedMessageInterface getShortDescription(CommandInterface command)
{
    return MyMessages.AddWarpShortDescription;
}

@Override
public LocalizedMessageInterface getDescription(CommandInterface command)
{
    return MyMessages.AddWarpDescription;
}
```

Method onTabComplete should be self-explained. The other two methods are used to build the help. As you can see we force each plugin writer to return some help text. We think that providing good help texts for chat command is always the best choice.

5.15.1.6 Create "removeWarp" command

Remove help command takes one argument: The name of the warp to be removed. We focus on two methods: handle and onTabComplete.

First the handle method that will fetch the warp point and delete it. It is very similar to addWarp except it deleted and not creates things.

```
@Override
public void handle(CommandInterface command) throws McException
{
    // removing is valid on console! No Exception on console!
    // check for permission or operator flag
    command.permOpThrowException(MyPermissions.RemoveWarp, “remove-warp”);

    if (command.getArgs().length != 1)
        throw new McException(MyMessages.MissingWarpName);
    final String name = command.getArgs()[0].trim();
    if (name.length() < 3)
        throw new McException(MyMessages.WarpNameTooShort);

    // check for existence
    final WarpPoint existing = WarpPoint.forName(name);
    if (existing == null)
        throw new McException(MyMessages.WarpNotFound, name);

    // delete it
    existing.getComponent().delete();

    // success
    command.send(WarpDeleted, name);
}
```

As you can see we directly call the delete method on the component. The framework is doing the rest.

Now let us create a onTabComplete method returning the warp point names.

```
@Override
public List<String> onTabComplete(CommandInterface command, String lastArg)
{
    final Collection<ComponentInterface> warps =
        ObjectServiceInterface.instance().findComponents(MyComponents.Warp));
    return warps.stream()
        .map(w -> (WarpPoint) w.getHandler())
        .map(WarpPoint::getName)
        .filter(w -> w.startsWith(lastArg))
        .sorted()
        .collect(Collectors.toList);
}
```

This method filters the warp points by given “lastArg” and returns a sorted list. This is also a nice sample how to use the java8 features.

5.15.1.7 Create “useWarp” command

Using a warp point causes the player to teleport to this warp point. The onTabComplete method is the same than the one taken from removeWarp command. We will concentrate on the handle method.

We have to check for the warp point and let the player teleport to this location.

```
@Override
public void handle(CommandInterface command) throws McException
{
    // not valid on console
    command.when(CommandInterface::isPlayer)._elseThrow(
        MyMessages.NoConsole);
    // check for permission or operator flag
    command.permOpThrowException(MyPermissions.UseWarp, “use-warp”);

    if (command.getArgs().length != 1)
        throw new McException(MyMessages.MissingWarpName);
    final String name = command.getArgs()[0].trim();
    if (name.length() < 3)
        throw new McException(MyMessages.WarpNameTooShort);

    // check for existence
    final WarpPoint existing = WarpPoint.forName(name);
    if (existing == null)
        throw new McException(MyMessages.WarpNotFound, name);

    // teleport
    command.getPlayer().getBukkitPlayer().teleport(
        existing.getComponent().getLocation());

    // success
    command.send(WarpUsed, name);
}
```

5.15.2 Signs

Signs are special component variants used by Minecraft signs. They are only valid on Minecraft signs. So before creating them in code you need to create a bukkit sign and after it create a sign component on it.

The second difference is that signs listen for break events. Breaking signs will cause to check for deletion. If the deletion is allowed the sign break event is OK. If the deletion is disallowed the break event is cancelled.

However, the signs may get broken because the block they are connected to is broken. In this case the sign will be deleted even if your code does not like it. Be aware of this situation.

Relocating sign objects is only valid if the new location represents a sign.

5.15.3 Entities

We divide two variants of entities. First of all, you are able to attach an entity handler to any existing Minecraft entity. Thus, the control of such entities is very limited. The second variant lets you spawn entities and control everything by yourself.

Entities can be created by ObjectServiceInterface. There is no secret. Compared to Components they only get a spigot entity during creation.

IMPORTANT: Attaching a handler to entities does not prevent them from being unloaded if no player is nearby. This will destroy the entities and the entity handlers. Creating persistent handlers on Minecraft entities maybe senseless.

In the following chapter I will give you an overview for the entities you can build from scratch.

5.15.3.1 Creating villagers

You can create custom villager NPCs. Currently they simply will do nothing.

After fetching a Builder from NpcServiceInterface you can invoke various methods and then create the NPC.

```
    NpcServiceInterface.instance().villager().
        location(loc).
        handler(MyEntities.MyDummyVillager, null).
        persistent().
        profession(Profession.PRIEST).
        create();
```

This code will create a persistent villager being recreated on server restart.

In future versions, we may support more methods within the builder.

5.15.3.2 Creating Humans with skins

We are able to spawn “fake players”. They may look like real players with support for individual skins. Creating them is similar to creating villagers.

```
NpcServiceInterface.instance().human().
    location(loc).
    handler(MyEntities.MyDummyHuman, null).
    persistent().
    name(“JACK”).
    skin(myPlayer).
    create();
```

This code will create a persistent human with individual player skin being recreated on server restart.

In future versions, we may support more methods within the builder.

5.15.4 Zones

Where a component represents a single block a zone is made of several blocks. The zone contains a low location and a high location forming a cuboid. Everything within both locations (inclusive) is meant to be inside the zone.

Hint: The cuboid always normalizes the lower and higher location. Means that the lower location will always contain the lowest coordinates. Simple example:

```
final World world = ...;
final Location low = new Location(world, 1, 3, 5);
final Location high = new Location(world, 6, -4, 12);
final Cuboid cub = new Cuboid(low, high);
assertEquals(1, cob.getLowLoc().getX());
assertEquals(-4, cob.getLowLoc().getY());
assertEquals(5, cob.getLowLoc().getZ());
assertEquals(6, cob.getHighLoc().getX());
assertEquals(3, cob.getHighLoc().getY());
assertEquals(12, cob.getHighLoc().getZ());
```

The red lines show the normalization.

Hint: Cuboids with low location/ high location in different Minecraft worlds are NOT allowed. The locations must always be in the same world.

Hint: Working with zones should always use zone type ids. If multiple plugins use McLib and a zone of plugin1 shares a location with a zone from plugin2 you might get unexpected results while checking the zones.

5.15.4.1 Zones sharing same locations

Zones may overlap each other. McLib does not check if one zone overlaps another one. You are free to create multiple zones with same cuboid or with overlapping cuboid.

The handling from players point of view is described in the following chapter.

If you do not like overlapping zones you will have to check for overlapping zones before creating or before moving the cuboid.

5.15.4.1.1 “Matching” zones

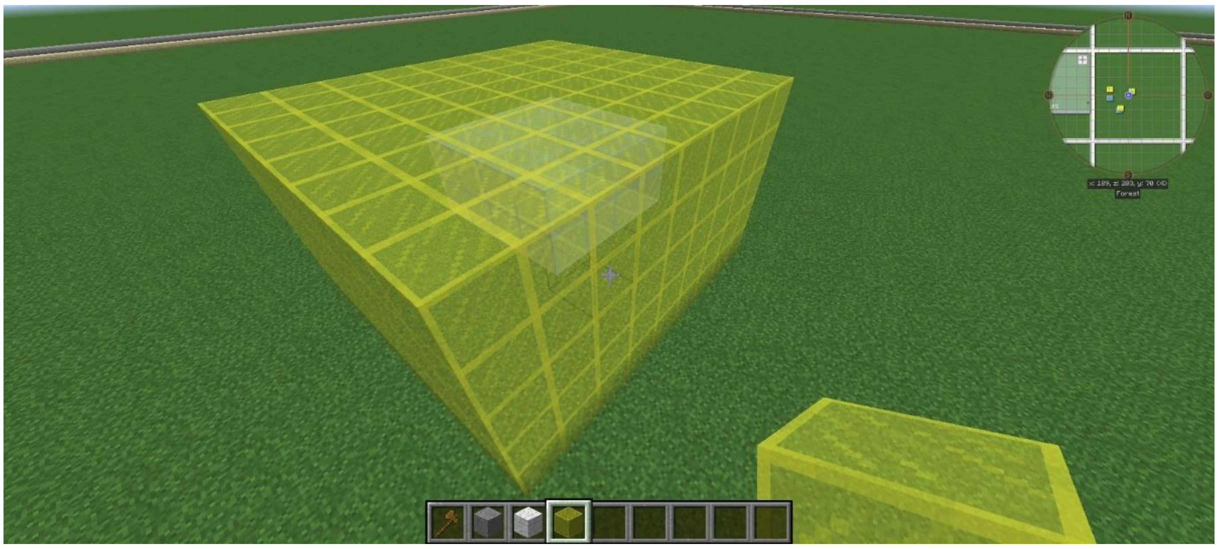
We call two zones that have identical cuboids “matching zones”. Example:

```
final Cuboid cub1 = zone1.getCuboid();
final Cuboid cub2 = zone2.getCuboid();
final isMatching = cub1.equals(cub2);
```

5.15.4.1.2 “Child” zones and “Parent” zones

If zone1 is completely surrounded by zone2 we call zone1 the child and zone2 the parent. In other word: All blocks within the child zone are contained in parent’s zone.

```
final Cuboid cub1 = zone1.getCuboid();
final Cuboid cub2 = zone2.getCuboid();
final isZone1Child = cub1.isChild(cub2);
final isZone1Parent = cub1.isParent(cub2);
```

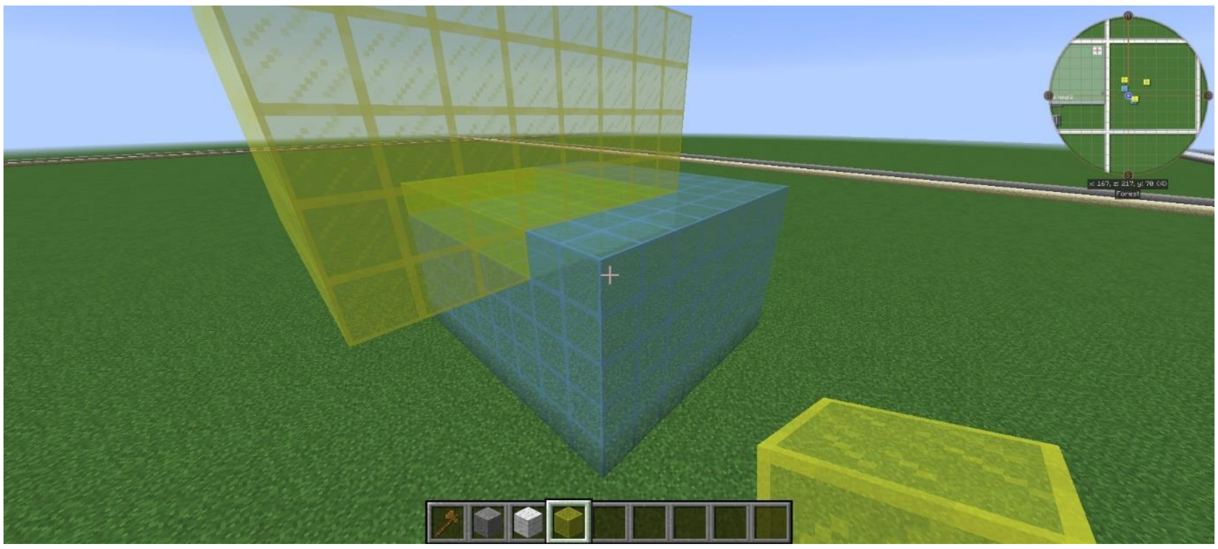


The blue glass (child) is completely within the yellow glass (parent).

5.15.4.1.3 “Overlapping” zones

If zone1 and zone2 share locations but are not child or parent for each other they are called “overlapping”.

```
    final Cuboid cub1 = zone1.getCuboid();
    final Cuboid cub2 = zone2.getCuboid();
    final isOverlapping = cub1.isOverlapping(cub2);
```



The blue glass and the yellow class share only some blocks (green glass). Both have own blocks outside the shared area.

5.15.4.2 Players current zone

The player always has maximum of one primary zone he Is located in. This zone is returned by McPlayerInterface.getZone.

If the player enters a new zone an event will be fired. If the player leaves the zone another event will be fired. If the player switches zones (leaves one zone and directly enters another zone) the two corresponding events will be fired. First the leave event and after it the enter event.

It will be more difficult if you think of overlapping or child/parent zones.

The player maybe within more than one zone. To find all the zones he entered you should invoke McPlayerInterface.getZones. This will return the primary zone as well as all other zones the player is in.

Understanding the zone concept is important if you watch for zone events.

5.15.4.2.1 Entering/leaving matching zones

Assume that zone1 and zone2 are matching zones.

If a player enters the zones, there will be an enter zone event for both zones. But the ordering of the events is random. Maybe one time the zone1 event is fired first, maybe in another situation the zone2 event is fired first.

It is not guaranteed which zone becomes the primary one for the player. It maybe zone1 or maybe zone2.

5.15.4.2.2 Entering/Leaving parent child zones

Assume that childZone is a child of parentZone.

McLib ensures that the player always enters the parent zone first. Even if the player is teleported inside the child zone. You can trust in the following event order:

```
ZoneEnterEvent(parent)
parent becomes primary zone
ZoneEnterEvent(child)
child becomes primary zone
ZoneLeaveEvent(child)
parent becomes primary zone
ZoneLeaveEvent(parent)
```

5.15.4.2.3 Entering/Leaving overlapping zones

This is more difficult. Let us assume the player walks from zone1 into the shared locations and leaves zone2 after leaving the shared locations. This will result in the following events:

```
ZoneEnterEvent(zone1)
zone1 becomes primary zone
ZoneEnterEvent(zone2)
zone2 becomes primary zone
ZoneLeaveEvent(zone1)
zone2 is still the primary zone
ZoneLeaveEvent(zone2)
```

If the player directly walks into the shared locations area the event ordering is random as of entering/leaving matching zones.

5.15.4.3 Finding zones by type

The ObjectServiceInterface contains various methods for finding zones by type. First of all, a simple findZones taking only a zone type id as argument. It will return all zones of a given type.

The other methods are variants of the methods without type argument. See the following chapters and keep in mind that there is always an option to find the zones either by type or without type filter.

5.15.4.4 Finding zones by location

Searching zones by location means: Search a zone that includes the location within its cuboid.

If you use method findZone, it will return he first zone it finds. It is random. The method findZones (plural) will return all matching zones.

5.15.4.5 Finding zones by location without Y coordinate

Sometimes you do not want to check for y coordinate. For example, if you want to check if a player is somewhere above or below a zone you can use findZoneWithoutY or findZonesWithoutY.

But be aware that there are no bukkit events if a player simply is above or below a zone but not inside the zone.

5.15.4.6 Finding zones by location without Y coordinate including 2 blocks

The third variant are methods findZoneWithoutYD and findZonesWithoutYD. They ignore Y coordinate and they add 2 to the zones boundary.

The effect is that they find zones the player is above, below or nearby.

5.15.4.7 Finding zones by cuboid

Finding zones by cuboid is similar than finding zones by location. You need to pass parameter cuboid as well as a cuboid search mode.

The search mode influences the type of cuboids returned. See the chapters about matching/child/parent/overlapping zones for details.

There are more handy methods within the ZoneInterface. For example, to find child zones you can invoke ZoneInterface.getChildZones.

5.15.5 Zone Scoreboards

A special feature of zones are the zone scoreboards. Once you enter a zone that has a scoreboard setup it is displayed. If you leave the zone the scoreboard will be removed.

The zone scoreboards are not persistent. Your zone handler needs to recreate them if a persistent zone is resumed.

To use this feature first create a scoreboard variant with a string identifier.

```
final ZoneInterface zone = ...
final ZoneScoreboardVariant sb = zone.createScoreboardVariant(“default”);
sb.setLines(“Line1”, “Line2”);
```

Zone scoreboards are always starting invisible. To enable them finally invoke show:

```
sb.show();
```

5.15.6 Abstract Objects

Abstract objects are any kind of objects without locations. They are kind of abstract because Minecraft itself does not know how to handle them (everything in Minecraft has at least a location).

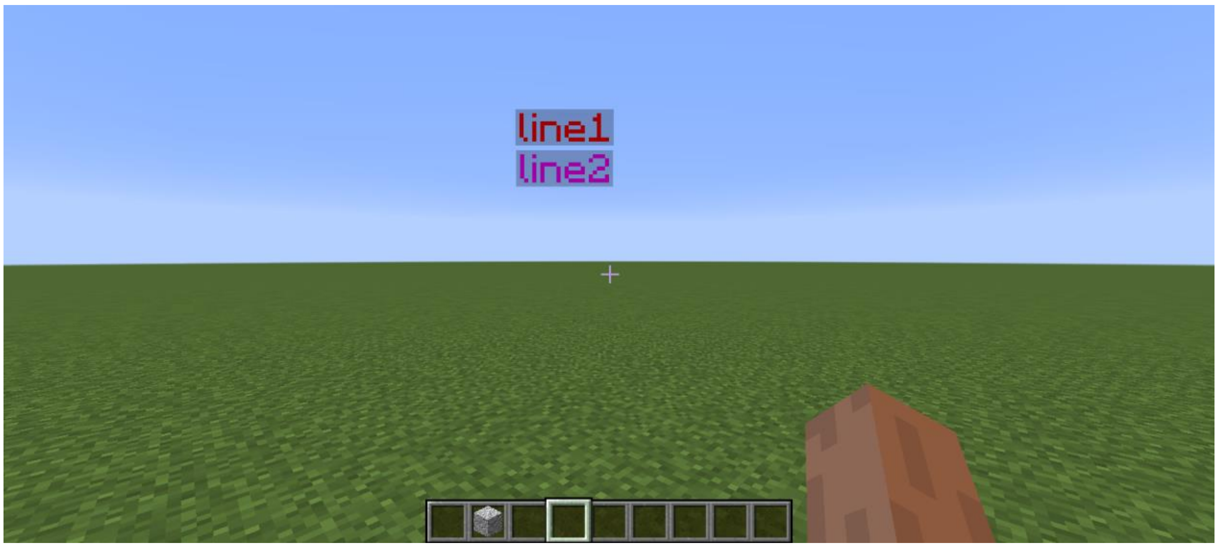
For example, the minigames arenas are abstract objects.

You will use objects the same way than components.

5.15.7 Holograms

Holograms are special component variants to display some text within Minecraft worlds.

They are bound to locations and can display any amount of text lines. The HologramHandlerInterface is responsible to set the hologram text during creation or resume of the hologram.



Code sample:

```
ObjectServiceInterface.instance().createHologram(
    MyHolograms.Dummy,
    myPlayer.getLocation().clone().add(0, 2, 0),
    new DummyHologram(),
    false);

// inside your hologram handler:
public void onCreate(HologramInterface hologram)
{
    hologram.setLines(new Serializable[]{
        “§4line1”,
        “§5line2”
    });
}
```

Notice: In this code sample, we use the player position and add 2 to y coordinate. This lets you choose a proper height for your hologram and does not create it inside the earth.

5.15.8 Localized object content

The objects from mclib object framework support player individualization under the hood. As you can see in the API the names or texts are always given as Serializables. What does it mean?

You can give static strings to them. Everything will work as expected. The entities get their static names etc.

But if you give localized messages to them the texts will be customized for each player. As soon as the player sees the object it's individual text is encoding depending on players locale and placeholders. So, each player may get individual content automatically.

We currently support the following localized object content:

Object type	Data type
Scoreboards	Line content
Signs	Line content
Holograms	Line content

5.16 Event framework

The McLib event framework extends the spigot classic event framework. Before you start with McLib event framework keep in mind: You can always use the classic spigot events and listeners. If you want to fetch all events for certain type you may not use the McLib framework at all.

The McLib event framework has two benefits:

- The framework sets context variables (see chapter 5.8 Context sensitive execution)
- Passing events to objects.

5.16.1 Fetching/sending events (Spigot)

As I already told there is no secret. Sending events is done through standard spigot services.

```
final MyEvent event = new MyEvent();
Bukkit.getPluginManager().callEvent(event);
```

Fetching events can be done through Spigot as well. See Spigot documentation for details.

5.16.2 Using the event bus

The event bus of McLib allows registering event listeners for McLib event handling.

You can either register handlers for a single event or a listener object for multiple events.

Pass your plugin, the McLib event class and a consumer method to register for a single event.

If your handler method throws McException it will only be logged.

```
McLibInterface.instance().registerHandler(
    plugin,
    McPlayerMoveEvent.class,
    this::onMove);
private void onMove(McPlayerMoveEvent evt) throws McException
{
        // ...
    }
```

To register multiple events, you can create your own Listener.

```
    McLibInterface.instance().registerHandlers(
        plugin,
        new MyListener());
    class MyListener implements McListener
    {
        @McEventHandler
        public void onMove(McPlayerMoveEvent evt) throws McException
        {
            // ...
        }
    }
```

As you can see it is similar to spigots event handling. Nothing new to learn. You only have some other methods to invoke and other Interfaces to use.

5.16.3 Player events

You can directly register events on McPlayerInterface. Whenever an event is invoked from this player it will be passed to your event handler. If other players invoke the event your event handler ignores it.

This allows you to work more object oriented. No more event catching for players that are not interesting.

5.16.4 Object events

Each object of McLib (Abstract objects, Zones, Components etc.) have their own event bus. Similar to chapter 5.16.3 Player events you will only catch events where these objects are involved. You will find the registerHandler and registerHandlers methods in the corresponding Interface.

5.16.5 Bukkit event wrapping

In chapter 5.16.2 Using the event bus you already saw an event class called “McPlayerMoveEvent”. McLib wraps nearly every spigot event. Simple add the “Mc” prefix. For details on the events see Javadoc.

5.16.6 Events introduced by McLib

McLib has two event types similar to spigot: Events that can be cancelled and events that only can be catched but not cancelled.

For cancellable events, we use the AbstractVetoEvent base class. It allows event listeners to veto the event and give a message for veto reason.

```
    event.setCancelled(Messages.MyVetoReason);
```

Most of the time the veto results in throwing a McException in the original method. For example, the creation of zones can get a veto and this causes method “createZone” to throw an exception.

You may assume that your veto reason will at least be displayed to the user invoking the original command causing the event.

Note: Throwing an exception in your listener will NOT cancel the event nor will the exception be thrown over event bus. It will only be logged. Throwing exceptions in event handlers means: “Something went terribly wrong for unknown reason”. Please be careful with throwing exceptions in listeners.

5.16.7 Creating own events

You need two things: First of all, create an event class extending spigots Event class or McLibs AbstractVetoEvent.

```
    class MyEvent extends Event implements MinecraftEvent<MyEvent, MyEvent>
    {
        private static final HandlerList handlers = new HandlerList();

        @Override
        public HandlerList getHandlers()
        {
            return handlers;
        }

        public static HandlerList getHandlerList()
        {
            return handlers;
        }

        public MyEvent getBukkitEvent()
        {
            return this;
        }

        public McOutgoingStubbing<MyEvent> when(McPredicate<MyEvent> test)
            throws McException
        {
            if (test.test(this)) return new TrueStub<>(this);
            return FalseStub<>(this);
        }
    }
```

In your plugins onEnable method we register the event with McLibs event bus:

```
    McLibInterface.instance().registerEvent(this, MyEvent.class);
```

If your event is related to players or McLibs objects you may override the corresponding getter methods too. See the MinecraftEvent interface for details. If you do not override the methods your event will be only a global event. If you override it the event will be passed to the objects you are returning in the corresponding getter.

5.17 Custom inventories

Mclib supports custom inventories of any sizes. To create your own inventory first create an enumeration and register it during onEnable.

```
public enum MyInventories implements InventoryTypeId
{
    MyGlobalInventory,
    MyChest
}
```

5.17.1 Accessing global inventories

A global inventory is unique within your server. It can be identified by your inventory type and a unique string. You can have multiple global inventories by using multiple string identifiers.

You can obtain an inventory id through InventoryServiceInterface.

```
final InventoryId iid = InventoryServiceInterface.instance().getInventory(
    MyInventories.MyGlobalInventory,
    "default");
```

You can create the inventory on demand if it does not exist:

```
final InventoryId iid =
    InventoryServiceInterface.instance().getOrCreateInventory(
        MyInventories.MyGlobalInventory,
        9*16, // 16 rows
        true, // fixed size
        true, // shared
        "default");
```

5.17.2 Accessing location bound inventories

A location bound inventory is bound to one or multiple locations. Instead of giving a string identifier you can give a bukkit location to the inventory service.

```
final InventoryId iid = InventoryServiceInterface.instance().getInventory(
    MyInventories.MyGlobalInventory,
    new Location(...));
```

```
final InventoryId iid =
    InventoryServiceInterface.instance().getOrCreateInventory(
        MyInventories.MyGlobalInventory,
        9*16, // 16 rows
        true, // fixed size
```

```
                true, // shared
                new Location(...));
```

5.17.3 Managing locations

You can create new locations for existing repositories by invoking method “bindInventory”. And you can remove inventories from existing locations by invoking method “unbindInventory”.

Keep in mind: The locations bound to an inventory is only some kind of information. It helps you locating an inventory in Minecraft worlds. This does not do any modification to the Minecraft world itself. You can bind an inventory to a stone and nothing changes. It is still a stone.

It is up to you to use the inventory. For example, you can create a component at the stones location and whenever a player right clicks it you will programmatically open the inventory bound to this stone.

5.17.4 Shared or non-shared inventories

Classic chests are always shared inventories. They have exactly one storage and each player uses the same storage.

Non-shared inventories are used for chests, where each player has its own storage. It is independent from other players. If player FOO opens the inventory he will only see his items and if player BAR opens it he will see other items. So, each player has its individual (non-shared) storage.

You will have to decide during creation of the inventory if it is shared or not.

5.17.5 Manipulating or opening inventories

First obtain the InventoryDescriptorInterface.

```
    final InventoryDescriptorInterface desc =
        InventoryServiceInterface.instance().getInventory(
            iid);
```

Now you can manipulate the inventory:

```
    desc.getInventory(myPlayer).clear();
```

Opening the inventory is really easy:

```
    desc.openInventory(myPlayer);
```

Mclib will open classic inventories if they can handle the inventory size. Instead it will open a simply GUI where you can browse between several pages of your inventory.

5.17.6 Binding inventories to custom block types

If you like to create modded world with custom chest types you can use the BlockInventoryMeta annotation. Please read the important technical background in the modded world chapter before using this feature.

Details how to create your own block enumerations are discussed in the modded world chapter.

```
@BlockVariantMeta(
    fixed = true,
    shared = true,
    size=3*9 * 2, // twice as large as a classic one field vanilla chest
    blockInventory = CustomInventory.class)
BronzeChest,
```

You need a special handler class for your own chest.

```
public class CustomInventory implements BlockInventoryInterface
{
    public void createInventory(BlockId block, BlockVariantId variant,
        Location location, McPlayerInterface player, int initialSize,
        boolean fixed, boolean shared)
    {
        InventoryServiceInterface.instance().getOrCreateInventory...
    }
    public InventoryId getInventory(BlockId block, BlockVariantId variant,
        Location location, McPlayerInterface player)
    {
        return InventoryServiceInterface.instance().getInventory…
    }
    public void onBreak(BlockId block, BlockVariantId variant,
        Location location)
    {
        // maybe delete silently or spawn all items?
    }
}
```

5.18 Skulls and skins

With skins, you are able to spawn player heads (skulls) or even human NPCs having a special skin.

All you need is a valid online player.

1. Receiving skins from players

A skin from players used by skulls can be received through get method.

```
SkinInterface skin = SkinServiceInterface.instance().get(player);
```

2. Building skulls from player skins

After receiving the skin, you can create a skull (itemstack).

```
final ItemStack item = SkinServiceInterface.instance().getSkull(skin, “YOU”);
```

3. Save a skin snapshot (asynchronous)

To use a skin, for example on humans, we need a snapshot of the skin. This loads the skin texture. Note that this can only be done asynchronous because it should not block the server.

```
SkinServiceInterface.instance().getSkinSnapshot(skin, this::saveSkin);
```

4. Saving and loading skins

Now let us save the skin to some data section (for example configuration files) for later use.

```
SkinServiceInterface.instance().save(skin, section, “mySkin”);
```

Loading is done through method “load”.

```
SkinInterface skin = SkinServiceInterface.instance().load(section, “mySkin”);
```

5. Setting human skin (asynchronous)

Now we want to use it on our human entity we spawned by using the human entity builder (see previous chapters).

```
SkinServiceInterface.instance().setToHuman(entity, skin);
```

If the entity is persistent it will even save the skin snapshot in configuration files automatically.

5.19 Schemata services

The schemata service allows you to store and rebuild parts of your world. The main difference from other schematics (f.e. WorldEdit) is that it stores the whole mclib objects too (zones, components, holographics etc.)

5.19.1 Create Schematic

First obtain a builder instance from SchemataServiceInterface.

```
final SchemataBuilderInterface builder =
    SchemataServiceInterface.instance().builder();
```

Set some useful data:

```
builder.setTitle(Locale.ENGLISH, “My Schemata”);
builder.setAuthor(“myself”);
builder.setSchemataUrl(“http://www.mysite.de/schemata/myschemata”);
builder.setSchemataVersion(1);
```

The schemata url and version are optional. They are only important if you like to share your schemata with others. In future versions, we will support some kind of marketplace for minigame arenas. To identify your arenas and newer versions of them you will have to setup the url and version with meaningful values.

After setting the meta data you will have to set parts. Each part is an independent cuboid.

```
builder.addPart(
    “my part #1”,
    new Cuboid(new Location(...), new Location(...)),
    this::onSuccess,
    PartBuildOptions.WithObjects,
    PartBuildOptions.WithAir
);
```

Notice: The cuboid is added asynchronous. You should ensure that players do not change the world during schemata building. In “onSuccess” handler you can add another part or do any other thing you like including file save.

5.19.2 Store Schematic to file

To store your schemata to a file simply invoke “saveToFile”.

```
builder.saveToFile(new File(...));
```

The file is gezipped. Do not try to edit it in editors.

5.19.3 Load Schematic from file

Loading schematas from files is easy:

```
final SchemataInterface schemata =
    SchemataServiceInterface.instance().loadFromFile(new File(...));
```

5.19.4 Paste schematic to world

You can either paste the parts each by each or ionvoke the method “applyToWorld” on the schemata instance. Keep in mind that you need to pass as many locations to your schemata than having party in your schemata.

```
schemata.applyToWorld(
    new Location[]{...},
    this::onSuccess,
    this::onFailure);
```

This method is asynchronous. You should wait for invocations of onSuccess or onFailure before going on.

5.20 Things to be done in future releases

Our major TODO list…

- Optimize bungee cord support (have a bungeecord plugin supporting communication even if no one is online on a server)
- Tune Entity-Support
- Optimize and describe client development (forge)
- Support language packs
- Performance tuning
- Tune smart gui widgets
- Simple/Smart Gui for editing messages and config options.
- Testing support (fakeplayer etc.)
- Titles

6 Developing against McLib (Forge clients)

TODO forge clients in details

7 BungeeCord-Clusters and Client/Server

7.1 Communication channels

McLib support communication between multiple servers (BungeeCord networks) or from client to server and vice-versa.

7.1.1 Registering communication channels on server side

To use this feature first create a communication channel. Like other features you will have to create and register an enumeration.

```
public enum MyChannels implements CommunicationEndpointId
{
    SomeChannel
}

// somewhere in onEnable
EnumServiceInterface.instance().registerEnumClass(this, MyChannels.class);
```

After registering the enumeration, you must invoke registerBungeeHandler and/or registerPeerHandler on ServerCommunicationServiceInterface depending whether you are expecting messages from client or other bungee servers.

In your onDisable method you should invoke removeAllCommunicationEndpoints to get a clear shutdown.

7.1.2 Registering communication channels on client side

TODO communication channels on client side

7.1.3 Preparing messages

Messages are created by using data sections. For simplicity, you can use DataFragments with AnnotatedDataFragment base class.

```
final DataFragment myFragment = ...;
final DataSection dataSection = new MemoryDataSection();
dataSection.set("myData", myFragment);
```

7.1.4 Sending messages to the "other side"

The communication endpoint comes with an option to send messages to the other side. Being on client it will send the message to the server. Being on server side it will send the message to the the client of the current player (see chapter 5.8.2.4 getCurrentPlayer).

```
MyChannels.SomeChannel.send(dataSection);
```

Another way to send messages directly to the client is done by invoking sendToClient on McPlayerInterface.

If you want to broadcast to all connected clients, you can use method broadcastClients on McLibInterface.

7.1.5 Sending messages within bungee network

Sending within bungee network is simple. First identify the spigot server you want to send a message to. You will find the servers by name in BungeeServerInterface.

```
final BungeeServerInterface target =
    BungeeServiceInterface.instance().getServer(“lobby”);
target.send(MyChannels.SomeChannel, dataSection);
```

If you want to broadcast to all connected bungee servers, you can use method broadcastServers on McLibInterface.

8 Smart GUI

The smart gui is a feature based on client mods. It helps you to display nicer guis. For example, list views, trees etc.

To work the player must install the McLib client mod. See chapter 4.8 Forge client for installation details.

8.1 Initializing the smart gui.

Within McPlayerInterface invoke the method hasSmartGui for cheking and then method openSmartGui.

```
    if (!player.hasSmartGui())
        player.sendMessage(MyMessges.PleaseInstallSmartGui);
    else
    {
        final GuiSessionInterface gui = player.openSmartGui();
        // do whatever you like (see the following chapters)
    }
```

Note: In the next chapters, we use localized messages. However, the smart gui does not yet support color codes. We strongly recommend to use severity “Neutral” within your messages. For example:

```
    @LocalizedMessage(defaultMessage=”OK”, severity=MessageSeverityType.Neutral)
    OkButton
```

8.2 Buttons

Many sgui elements allow buttons. A button is created by method sguiCreateButton.

```
final okButton = gui.sguiCreateButton(
        MyMessages.OkButton, null);
```

To listen for button clicks add a consumer function taking SGuiInterface and DataSection. The SGuiInterface leads to the gui itself. The DataSection may be null if the button is placed outside forms (error popups etc.) or it contains the form data if invoked within forms.

```
final okButton = gui.sguiCreateButton(
        MyMessages.OkButton, null,
        this::onButtonClicked);
```

It is assumed that the buttons will always close the dialog. If you do not want this behavior pass false at the end.

```
final okButton = gui.sguiCreateButton(
        MyMessages.OkButton, null,
        this::onButtonClicked,
        false);
```

8.3 Display simple message popups

8.3.1 Creating buttons

First you need to create an ok button and then invoke sguiDisplayError.

Code example:

```
final GuiButton okButton = gui.sguiCreateButton(
    MyMessages.OkButton, null);
```

If you want to perform some code once the player clicks the ok button you can pass an event handler method:

```
final GuiButton okButton = gui.sguiCreateButton(
    MyMessages.OkButton, null,
    this::onClickOk);
private void onClickOk(SGuiInterface sgui, DataSection formData)
{
    // do something
    sgui.close();
}
```

If you want the gui to automatically close on click you can simply pass true as last argument:

```
final GuiButton okButton = gui.sguiCreateButton(
    MyMessages.OkButton, null,
    this::onClickOk,
    true);
```

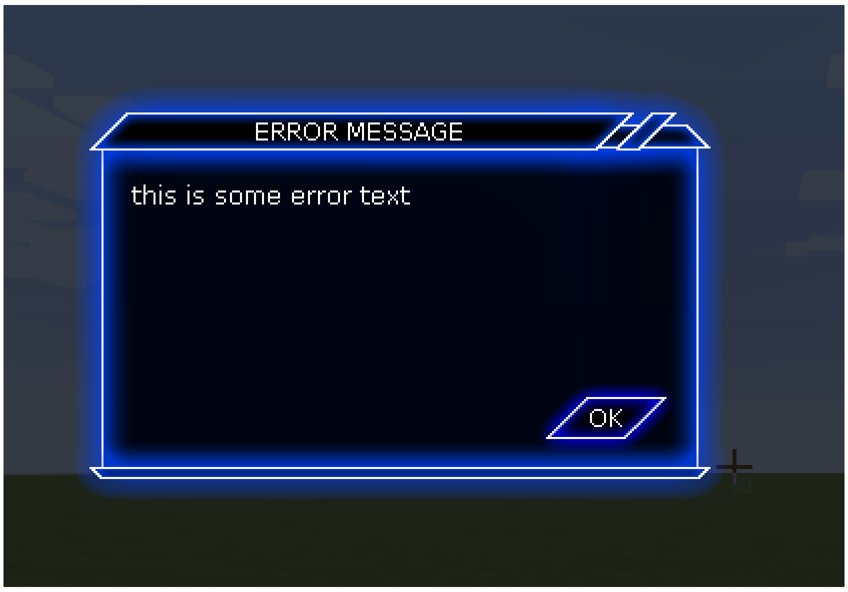
8.3.2 Display error message

After creating the OK-Button you have to invoke sguiDisplayError.

Code example:

```
gui.sguiDisplayError(
    MyMessages.ErrorTitle, null,
    MyMessages.SomeErrorText, null,
    okButton);
```

Now the error message is displayed. Screenshot:



this is some error text

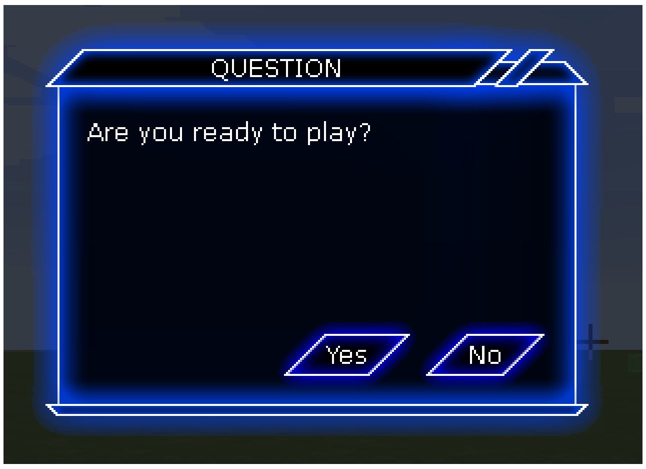
8.3.3 Display yes/no message

If you want the player to choose between two options (yes and no) you can display a popup with two buttons instead of only one button.

Code example:

```
    gui.sguiDisplayYesNo(
        MyMessages.QuestionTitle, null,
        MyMessages.SomeQuestionText, null,
        yesButton,
        noButton);
```

Screenshot:



Are you ready to play?

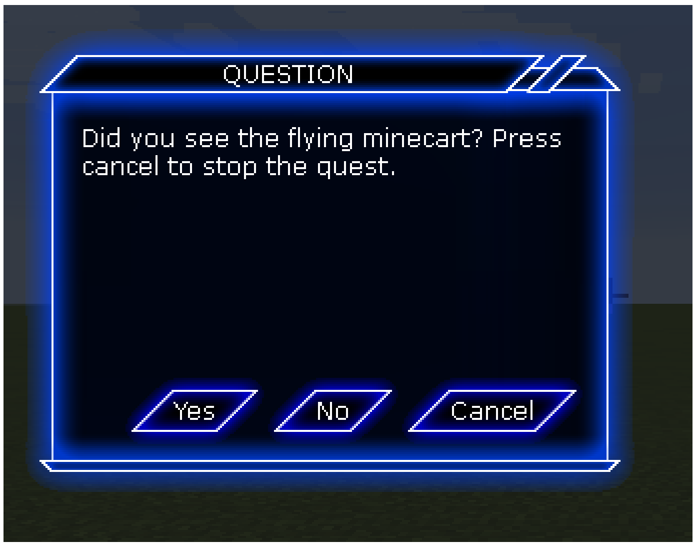
8.3.4 Display Yes/No/Cancel message

If you want the player to choose between three options (yes, no and cancel) you can display a popup with three buttons instead of only one button.

Code example:

```
gui.sguiDisplayYesNoCancel(
    MyMessages.QuestionTitle, null,
    MyMessages.SomeQuestionText, null,
    yesButton,
    noButton,
    cancelButton);
```

Screenshot:



Did you see the flying minecart? Press cancel to stop the quest.

8.4 Object markers

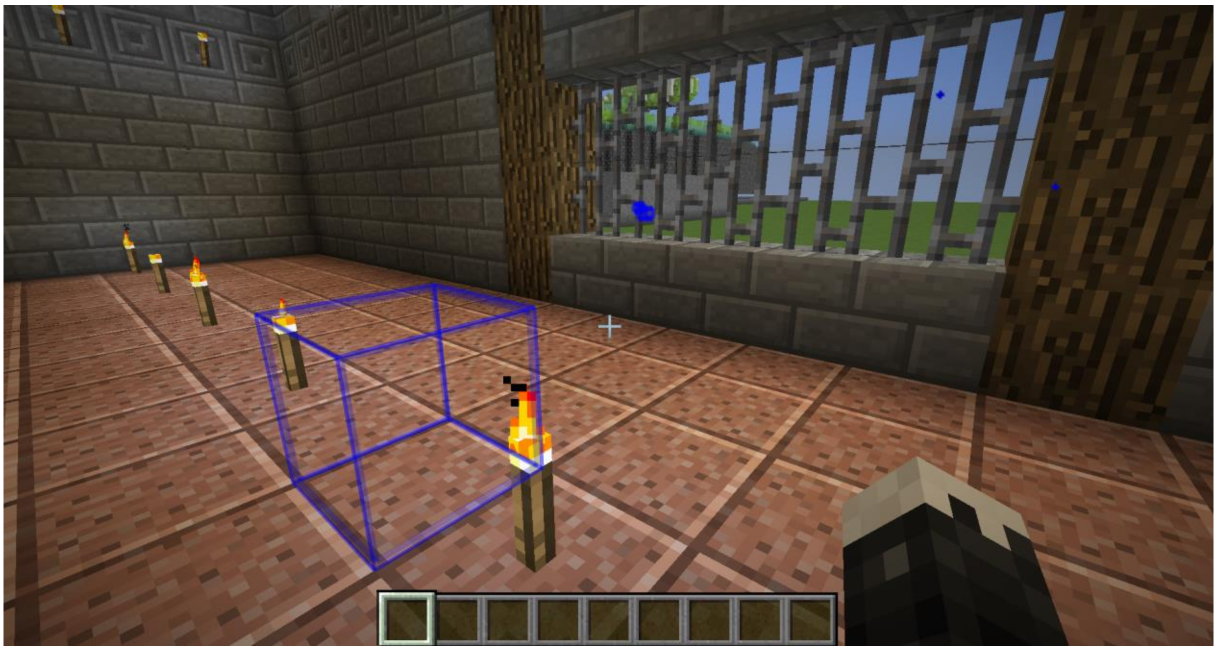
The smart gui is able to display object markers directly in Minecraft clients. The primary use case is to help administrators. But you may display markers for your users too.

A marker displays a cube around a block or zone to show up where your object is located.

```
final SGuiMarkerInterface marker = gui.sguiShowMarker(
    component,
    MyMessages.MarkerLabel);

// later on after some ticks or if the user invokes a clear command:
marker.remove();
```

To remove all markers, you can simply invoke “sguiRemoveAllMarkers”.

Screenshot



8.5 Form building

TODO form building

9 Modded worlds

9.1 Technical background/ Server setup

Modding game worlds means to create your own items and blocks. Vanilly Minecraft does not really support it in a clean way. Both, client and server, need to know what they are talking about.

While clients may support more blocks or items than servers the other way is not supported. If a server talks about blocks or items a client does not know it will simply cash.

Mclib solves this problem by introducing generic blocks and items that can be anything. If a client does not know about it will simply display some texture errors but not crash. However even this setup requires your users to install the forge client plugin.

So, will always keep in mind:

- Never use modded blocks or items in your lobby worlds because it will crash people not having the client mod installed.
- Before let the players join your modded servers over Bungee or your modded worlds check if the client mod is installed. Block the world or server if the client mod is not installed.

Second important notice is a mapping from custom item/block to the code used for it. As an experienced plugin developer, you may already know it: Minecraft internally uses numeric ids for items and blocks. If you install plugin A that introduces two weapons they may use id “2000” and “2001”. After installing the next plugin, it introduces a shovel with id “2002”.

Seems to be OK. But what if someone installs the second plugin before the first plugin? Now the shovel has id “2000” and the weapons have id “2001 and 2002”. Your world got broken and you cannot fix it without manipulating it on binary level. That’s the worst case.

There is one single file that you ALWAYS need to backup along with your worlds. And this even should even be identical over all your servers in the whole bungeecord network:

```
plugins/mclib/moddedworld.yml
```

This file keeps all the secret. It contains the mapping from minecraft items and blocks to your worlds binary code/ Minecraft ids. You should never lose it.

9.2 Custom items

Custom items are created by enumerations.

```
public enum MyItems implements ItemId
{
    @ItemData(
        modEnabled = true,
        textures = “some/path/bronzenugget.png”,
        modelJson = “...”)
    BronzeNugget
}
```

9.2.1 Furnace recipe

Use the FurnaceRecipe annotation to setup the item you get from smelting this item.

```
@FurnaceRecipe(BronzeFurnaceRuleToNugget.class)
BronzeIngot

class BronzeFurnaceRuleToNugget implements FurnaceRecipeInterface
{
    public ItemStack getRecipe(
        ItemId item, BlockId block, BlockVariantId variant)
    {
        return
            ItemServiceInterface.instance().createItem(MyItems.BronzeNugget);
    }
    public float getExperience(
        ItemId item, BlockId block, BlockVariantId variant)
    {
        return 0.1f
    }
}
```

9.2.2 Crafting recipes

Use the CraftingRecipes annotation to setup the crafting recipe to build your item.

```
@CraftingRecipes(
    shapeless = @CraftingShapelessRecipe(
        amount = 9,
        items = CraftingBronzeIngot.class
    )
)
BronzeNugget

class CraftingBronzeIgnot implements CraftingItemInterface
{
    public ItemStack item()
    {
        return
            ItemServiceInterface.instance().createItem(MyItems.BronzeIngot);
    }
}
```

This recipe means: You use one bronze ingot to receive 9 bronze nuggets.

9.2.3 Default itemstack size

Each item has its individual item stack size. Typical vanilla sizes are 64 or 16 or 1.

Use the ItemStackSize annotation to setup your preferred item stack size.

9.2.4 Tooling items

You can use several annotations to create tooling items. Only one annotation is allowed per item. In the following chapters, we give you sample values for bronze tooling's. Use them as a starting point for your development.

9.2.4.1 Swords

Sample for bronze swords:

```
@ItemSword(
    damage = 4.5f,
    damageVsEntity = 1.5f,
    speed = -2.4000000953674316D,
    getItemEnchantability = 5,
    durability = 190,
    digRule = DefaultSwordDigRule.class,
    dmgRule = DefaultWeaponDmg.class,
    repairRule = SelfRepair.class
    )
```

9.2.4.2 Shovels

Sample for bronze shovels:

```
@ItemShovel(
    durability = 190,
    digRule = DefaultShovelDigRule.class,
    dmgRule = DefaultToolDmg.class,
    repairRule = SelfRepair.class,
    damage = 3.0f,
    speed = -3.0D,
    getItemEnchantability = 5
    )
```

9.2.4.3 Axes

Sample for bronze axes:

```
@ItemAxe(
    durability = 190,
    digRule = DefaultAxeDigRule.class,
    dmgRule = DefaultToolDmg.class,
    repairRule = SelfRepair.class,
    damage = 8.0f,
    speed = -3.2D,
    getItemEnchantability = 5)
```

9.2.4.4 Hoes

Sample for bronze hoes:

```
@ItemHoe(
    durability = 190,
    digRule = DefaultHoeDigRule.class,
    dmgRule = DefaultToolDmg.class,
    repairRule = SelfRepair.class,
    damage = 0.0f,
    speed = -1.5D,
    getItemEnchantability = 5)
```

9.2.4.5 Pickaxes

Sample for bronze pickaxes:

```
    @ItemPickaxe(
        durability = 190,
        digRule = DefaultPickaxeDigRule.class,
        dmgRule = DefaultToolDmg.class,
        repairRule = SelfRepair.class,
        damage = 4.5f,
        speed = -2.8D,
        getItemEnchantability = 5)
```

9.2.4.6 Helmets

Sample for bronze helmets:

```
    @ItemArmor(
        durability = 7*13,
        dmgReduceAmount = 1,
        getItemEnchantability = 9,
        toughness = 0.0f,
        slot = ArmorSlot.Helmet,
        repairRule = SelfRepair.class
        )
```

9.2.4.7 Chestplates

Sample for bronze chestplates:

```
    @ItemArmor(
        durability = 7*15,
        dmgReduceAmount = 4,
        getItemEnchantability = 9,
        toughness = 0.0f,
        slot = ArmorSlot.Chestplate,
        repairRule = SelfRepair.class
        )
```

9.2.4.8 Boots

Sample for bronze boots:

```
    @ItemArmor(
        durability = 7*11,
        dmgReduceAmount = 2,
        getItemEnchantability = 9,
        toughness = 0.0f,
        slot = ArmorSlot.Boots,
        repairRule = SelfRepair.class
        )
```

9.2.4.9 Leggins

Sample for bronze leggins:

```
@ItemArmor(
    durability = 7*16,
    dmgReduceAmount = 5,
    getItemEnchantability = 9,
    toughness = 0.0f,
    slot = ArmorSlot.Leggins,
    repairRule = SelfRepair.class
    )
```

9.3 Custom blocks

Custom blocks are created by enumerations.

```
public enum MyBlocks implements BlockId
{
    @BlockData
    @BlockVariantData(
        textures = “some/path/bronzeore.png,
        modelJson = “...”)
    @BlockMeta(hardness = 3.0F, resistance = 5.0F)
    @BlockDropRule(DropSelf.class)
    BronzeOre
}
```

9.3.1 Furnace recipe

Use the FurnaceRecipe annotation to setup the item you get from smelting this item.

```
@FurnaceRecipe(BronzeOreFurnaceRule.class)
BronzeOre

class BronzeOreFurnaceRule implements FurnaceRecipeInterface
{
    public ItemStack getRecipe(
        ItemId item, BlockId block, BlockVariantId variant)
    {
        return
            ItemServiceInterface.instance().createItem(MyItems.BronzeIngot);
    }
    public float getExperience(
        ItemId item, BlockId block, BlockVariantId variant)
    {
        return 0.5f
    }
}
```

9.3.2 Crafting recipes

Use the CraftingRecipes annotation to setup the crafting recipe to build your item.

```
    @CraftingRecipes(
        shaped = @CraftingShapedRecipe(
            shape = {“iii”, “iii”, “iii”},
            items = @CraftingSpaedItem(
                shape = ‘i’, item = CraftingBronzeIngot.class)
        )
    )
    BronzeBlock

    class CraftingBronzeIgnot implements CraftingItemInterface
    {
        public ItemStack item()
        {
            return
                ItemServiceInterface.instance().createItem(MyItems.BronzeIngot);
        }
    }
```

This recipe means: You set all slots with bronze ingots to get one bronze block.

9.3.3 Default itemstack size

Each item has its individual item stack size. Typical vanilla sizes are 64 or 16 or 1.

Use the ItemStackSize annotation to setup your preferred item stack size.

9.4 World generation/ mineables

For world support, you can create bronze minables. There are three steps. First of all create a bukkit block populator and invoke the block service to create minables.

```
public class WorldGen extends BlockPopulator
{

    /** minimum y */
    private int minBronzeY = 12;
    /** maximum y */
    private int maxBronzeY = 64;

    /** chances to generate bronze */
    private int bronzeChances = 20;

    /** size of bronze minables */
    private int bronzeSize = 20;

    @Override
    public void populate(World world, Random random, Chunk chunk) {
        if (world.getEnvironment() == Environment)
        {
            final int cx = chunk.getX();
            final int cz = chunk.getZ();
            final int x = cx * 16;
            final int z = cz * 16;
            final int deltaY = this.maxBronzeY - this.minBronzeY;
            for (int i = 0; i < this.bronzeChances; i++)
            {
                final Location loc = new Location(world, x + random.nextInt(16),
this.minBronzeY + random.nextInt(deltaY), z + random.nextInt(16));
                BlockServiceInterface.instance().createMinable(random, loc,
MyBlocks.BronzeOre, BlockData.CustomVariantId.DEFAULT, this.bronzeSize);
            }
        }
    }
```

The world generator will create bronze ore. It uses the block service to access the original vanilla code already generating iron etc.

You may want to change the parameters or read them from configuration values. Keep in mind that you first need to setup configuration enums in your onEnable method before registering the block populator.

Second step is to register the block populator for target worlds. Best solution is to listen for world init events.

```
    @EventHandler
    public void onWorldInit(WorldInitEvent evt)
    {
        evt.getWorld().getPopulators().add(new WorldGen());
    }
```

The third step ensures that you have a chance to init the worlds correctly. Edit your plugin.yml and let your plugin be loaded during startup:

```
    load: startup
```

10 Advanced testing

10.1 spigot-testsupport

TODO spigot testsupport

10.2 fakeplayer

TODO fakeplayer

11 Chat commands

TODO chat commands

12 Configuration

TODO configuration

13 Supported third party plugins

13.1 Prequisites

If you install mclib-thirdparty jar it will connect several third-party plugins to mclib and vice versa. Simply add it to your plugins folder. There is no additional setup needed.

13.2 Placeholder-API

Spigot link:	https://www.spigotmc.org/resources/placeholderapi.6245/
Description:	Supports various placeholders. Can be extended by other plugins to provide more placeholders.

- Mclib will support the placeholders from Placeholder API as long as they match the mclib regex pattern
- Placeholders installed into mclib can be used in placeholder API