Developer Guide

Contains documentation relating to the BattleArena API.

- Getting Started
- Creating a Custom Arena Gamemode
    - Overview
    - Creating Your Arena Class
    - Adding Configurable Values to Your Arena
    - The Event System
    - Adding Game Logic
    - Per-Competition Code
    - Storing Map Information
    - Creating a Custom Command Executor
- Creating BattleArena Modules or Extensions
- Using the BattleArena API

Getting Started

BattleArena is designed to primarily be a framework to create games from both a user and development standpoint. Nearly everything is data driven, and many of the APIs provided from BattleArena allow you to further extend this.

Follow the steps below to add the API as a dependency.

Adding the BattleArena Repository

Gradle (Kotlin DSL)

```
repositories {
    maven("https://repo.battleplugins.org/releases")
    maven("https://repo.battleplugins.org/snapshots")
}
```

Gradle (Groovy)

```
repositories {
    maven {
        url "https://repo.battleplugins.org/releases"
    }
    maven {
        url "https://repo.battleplugins.org/snapshots"
    }
}
```

Maven

```
<repositories>
  <repository>
    <id>battleplugins-releases</id>
    <name>BattlePlugins Releases</name>
    <url>https://repo.battleplugins.org/releases</url>
  </repository>
  <repository>
    <id>battleplugins-snapshots</id>
```

```
    <name>BattlePlugins Snapshots</name>
    <url>https://repo.battleplugins.org/snapshots</url>
  </repository>
</repositories>
```

Adding the Dependency

Gradle (Kotlin DSL)

```
dependencies {
    compileOnly("org.battleplugins:arena:<version>)
}
```

Gradle (Groovy)

```
dependencies {
    compileOnly "org.battleplugins:arena:<version>"
}
```

Maven

```
<dependencies>
  <dependency>
    <groupId>org.battleplugins</groupId>
    <artifactId>arena</artifactId>
    <version>{version}</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

Creating a Custom Arena Gamemode

An overview on how to create a custom arena gamemode.

Overview

This is an overview of how to use the BattleArena API to create a custom gamemode. Before we start writing code, there's a few things to keep in mind regarding how BattleArena works and manages arenas:

Arena:

The Arena class contains the root logic of a game. This is where global options regarding a specific arena are configured, such as game events, the number of lives a player should have, and anything not linked to a specific competition. Each individual gamemode (i.e. Battlegrounds, SkyWars, etc.) will only have a single Arena instance.

Additionally, an Arena is directly tied to it's respective <arena>.yml file, so most any value accessible in code is directly taken from the YML.

Competition:

A Competition represents an active Arena. Where Arena contains all the actual game logic, a Competition will be responsible for handling anything pertinent to an active game. For instance, all the logic handling which phase a competition is in will be managed here.

CompetitionMap:

A CompetitionMap is the map in which the Competition is active in. In many cases, this is something that will not need to be extended, however for certain gamemodes where it is desired to store additional information (i.e. the layers in a spleef game), it may be useful to use this class.

No game logic is handled in the map class. It simply stores all the information necessary for a Competition to run (i.e. spawnpoints, the game border, the type of map, etc.

Creating Your Arena Class

Creating a Custom Arena Class

The first step in creating a custom gamemode is creating the Arena class for your game. To do this, simply create a new class and extend the **Arena** class.

```
package org.battleplugins.arena.example;

import org.battleplugins.arena.Arena;

public class MyArena extends Arena {

}
```

And to register it, simply run the following in your **onEnable** method inside your plugin:

```
BattleArenaApi.get().registerArena(this, "MyArena", MyArena.class, MyArena::new);
```

Using your Custom Arena

Now that the Arena is registered, you can now reference it from your **<arena>.yml**. For the sake of this tutorial, we will be copying the standard **arena.yml** and using it as a base going forward.

Inside of **plugins/BattleArena/arenas**, simply copy the **arena.yml** and call it **myarena.yml**. Change the name to **Arena** and add a new field: **mode: MyArena** (see the full myarena.yml at the bottom of this page).

On its own, this will not do much as no game logic has been designed yet. However, this will serve as a foundation for the rest of the tutorial.

Full myarena.yml

```
name: MyArena
mode: MyArena
type: Match
team-options:
```

```
  named-teams: false
  team-size: 1
  team-amount: 2
  team-selection: none
modules:
  - arena-restoration
  - classes
  - duels
  - scoreboards
lives:
  enabled: false
victory-conditions:
  teams-alive:
    amount: 1
  time-limit:
    time-limit: 5m
events:
  on-join:
    - store{types=all}
    - change-gamemode{gamemode=adventure}
    - flight{enabled=false}
    - teleport{location=waitroom}
  on-spectate:
    - store{types=all}
    - change-gamemode{gamemode=spectator}
    - flight{enabled=true}
    - teleport{location=spectator}
  on-leave:
    - clear-effects
    - restore{types=all}
    - remove-scoreboard
  on-death:
    - clear-inventory
    - respawn
    - delay{ticks=1}
    - teleport{location=waitroom}
options:
  - block-break{enabled=false}
  - block-place{enabled=false}
  - block-interact{enabled=false}
```

```
  - damage-entities{option=never}
  - keep-inventory{enabled=true}
  - keep-experience{enabled=true}
  - class-equip-only-selects{enabled=true}
initial-phase: waiting
phases:
  waiting:
    allow-join: true
    next-phase: countdown
    options:
      - damage-players{option=never}
      - class-equipping{enabled=true}
    events:
      on-start:
        - apply-scoreboard{scoreboard=waiting}
      on-join:
        - apply-scoreboard{scoreboard=waiting}
  countdown:
    allow-join: false
    allow-spectate: true
    revert-phase: true
    next-phase: ingame
    countdown-time: 5s
    options:
      - damage-players{option=never}
      - class-equipping{enabled=true}
    events:
      on-start:
        - apply-scoreboard{scoreboard=countdown}
  ingame:
    allow-join: false
    allow-spectate: true
    next-phase: victory
    options:
      - damage-players{option=other_team}
    events:
      on-start:
        - equip-class{class=warrior}
        - teleport{location=team_spawn}
        - give-effects{effects=[speed{duration=300;amplifier=1}]}
```

```
      - play-sound{sound=block.note_block.pling;pitch=2;volume=1}
      - apply-scoreboard{scoreboard=ingame-list}
  victory:
    allow-join: false
    allow-spectate: false
    next-phase: waiting
    duration: 5s
    events:
      on-complete:
        - leave
        - restore-arena
        - remove-scoreboard
      on-victory:
        - send-message{message=<green>Congrats, you won!</green>}
        - play-sound{sound=entity.player.levelup;pitch=1;volume=1}
      on-lose:
        - send-message{message=<red>Sorry, you lost!</red>}
        - play-sound{sound=block.anvil.place;pitch=0;volume=1}
      on-draw:
        - send-message{message=<yellow>It's a draw!</yellow>}
        - play-sound{sound=block.beacon.deactivate;pitch=0;volume=1}
```

Adding Configurable Values to Your Arena

Now that you have a custom Arena instance and a corresponding YML file, you are now ready to add some configurable options to your game!

The Configuration System - in a nutshell

One of the main features BattleArena includes a fully custom configuration system. This allows for very easy loading of data from config files. The Arena instance and YML file go hand-in-hand, so adding configurable options is very simple.

@ArenaOption annotation

This is an annotation that lets you load a value from the config. It has the following options

- **name**: The name of the option
- **description:** The description of the option
- **required:** Whether the option is required to be specified
- **contextProvider:** A provider to allow for more complex loading of an option (this option is covered in a more advanced page)

MyArena example

```
public class MyArena extends Arena {


    @ArenaOption(name = "infection-time", description = "How long a player should be infected once hit.")
    private Duration infectionTime = Duration.ofSeconds(5);
}
```

In this example, an infection time option has been added which can now be pulled from the **myarena.yml:**

```
name: MyArena
mode: MyArena
type: Match
infection-time: 3s
```

```
    ...
```

In this example, if no infection time if specified, then the default value of 5 seconds will be used as set in the code. However, if a config value with the same name from the ArenaOption is set, it will override the default value. In the case that a value should be required in order to load the arena, the **required** option can be set in the ArenaOption annotation.

On its own, this game will still do nothing. While we have an infection time option set that if referenced, will pull from the corresponding arena YML, we don't have any game logic. Before we dive into that, the next page in this tutorial will discuss the event system which is the heart of BattleArena and how you will deal with the vast majority of your game logic.

The Event System

The event system is at the heart of BattleArena, and is how game logic and most features will be configured. Most classes you will extend (i.e. Arena, Competition, Map, etc.) do not have (m)any overridable methods, but instead use events.

The BattleArena event system is an extension of Bukkit's event system, which should be familiar to most people reading this documentation. However, it comes with a set of additional features that let you scope events for specific arenas.

Arena Events

When specifying an event for BattleArena, rather than using Bukkit's @EventHandler annotation, BattleArena's @ArenaEventHandler annotation is available, letting you listen for events that **only** happen in your arena:

```
public class MyArena extends Arena {

    @ArenaOption(name = "infection-time", description = "How long a player should be infected
once hit.")
    private Duration infectionTime = Duration.ofSeconds(5);

    @ArenaEventHandler
    public void onInteract(PlayerInteractEvent event) {
        event.getPlayer().sendMessage("Interact while in Arena!");
    }
}
```

In this example, the PlayerInteractEvent is listened for, but what makes this different from using @EventHandler is this event will only be called for players **inside** a MyArena. This also means players in say a Battlegrounds arena would never see this message. If a random player playing survival were to click a block, nothing would happen, but if a player had ran **/myarena join** and started clicking blocks, you would see messages in console.

Custom Parameters

In addition to simply scoping out events per-Arena, BattleArena also lets you specify additional values in your event method that would otherwise not work in Bukkit. These two options are a **Competition** and an **ArenaPlayer**. For events that are per-player (i.e. PlayerInteractEvent), you

can specify both, however some events may not be per-player, but may only be called for an Arena (i.e. ArenaPhaseStartEvent). Here are two examples:

Player Event

```
@ArenaEventHandler
public void onInteract(PlayerInteractEvent event, ArenaPlayer player) {
    event.getPlayer().sendMessage(player.getPlayer().getName() + " interacted in arena: " +
player.getArena().getName());
}
```

Arena Event

```
@ArenaEventHandler
public void onPhaseStart(ArenaPhaseStartEvent event, Competition<?> competition) {
    System.out.println("The phase " + event.getPhase().getType().getName() + " was started in
competition " + competition.getMap().getName());
}
```

The second parameter is dynamic, meaning you can also specify the competition for your arena. So if you had a **SpleefCompetition** for instance, you could replace the **Competition<?>** reference with that.

Registering Listeners

By default, placing custom event listeners inside an **Arena** class will automatically register them for you without any additional code needed. However, if you wish to segregate this logic and instead have it in a separate class, a few things will need to be done.

Creating an ArenaListener

Rather than implementing Bukkit's listener, you will need to implement **ArenaListener**. Here is an example:

```
public class MyArenaListener implements ArenaListener {

    @ArenaEventHandler
    public void onInteract(PlayerInteractEvent event) {
        event.getPlayer().sendMessage("Interact while in Arena!");
    }
}
```

Registering Your Listener

In order to register your listener, you need to register it against the Arena in which you want to capture events for. In the example of MyArena, it is best done in the constructor like so:

```
public class MyArena extends Arena {

    @ArenaOption(name = "infection-time", description = "How long a player should be infected once hit.")
    private Duration infectionTime = Duration.ofSeconds(5);

    public MyArena() {
        super();

        this.getEventManager().registerEvents(new MyArenaListener());
    }
}
```

It is also important to note that if you are using standard @EventHandler annotations in your MyArenaListener, **they will not be registered using the above code.** You will need to run the standard **Bukkit#getPluginManager#registerEvents** method to do this. However, it is **not recommended** to mix these two together, and standard Bukkit events should be done in a separate listener, registered in the main class of your plugin.

Limitations

It is worth noting that not all events can be scoped by the @ArenaEventHandler annotation. This is mainly because the event system needs to have a pathway to extract an Arena player from an event. For instance, all events that are an instance of a **PlayerEvent** will be usable in BattleArena, since BattleArena can pull an ArenaPlayer from a standard Bukkit player. However, an event such as the **WeatherChangeEvent** cannot be referenced back to an Arena, because it is a global event not tied to a player. If you attempted to use the @ArenaEventHandler, nothing would happen.

Adding Game Logic

Now that you have a basic understanding of how the event system works, along with a base to work off of. It's time to implement some game logic!

Game logic can be implemented in two places: the Arena, or your Competition class. While we do not have a Competition class yet, it is typically recommended to leave most code inside of your Arena class, with a custom Competition class being responsible for storing values that change throughout the game (i.e. the number of blocks broken). We will get to this later.

Using Events

Going from our previous example of a grace period, let's add some listeners to make this functional!

```
public class MyArena extends Arena {
    private static final String INFECTED_METADATA = "infected";

    @ArenaOption(name = "infection-time", description = "How long a player should be infected
once hit.")
    private Duration infectionTime = Duration.ofSeconds(5);

    @ArenaEventHandler
    public void onDamageEntity(EntityDamageByEntityEvent event) {
        if (event.getDamager() instanceof Player damager && event.getEntity() instanceof
Player player) {
            // Player is not infected, let's infect them :)
            if (!player.hasMetadata(INFECTED_METADATA)) {
                player.setMetadata(INFECTED_METADATA, new
FixedMetadataValue(MyPlugin.getInstance(), true));
                player.sendMessage("You have been infected!");
                damager.sendMessage("You have infected " + player.getName() + "!");

                // Infect the player for the given duration
                Bukkit.getScheduler().runTaskLater(MyPlugin.getInstance(), () -> {
                    player.removeMetadata(INFECTED_METADATA, MyPlugin.getInstance());
                    player.sendMessage("You are no longer infected!");
```

```
                }, this.infectionTime.toMillis() / 50);
            }
        }
    }

    @ArenaEventHandler
    public void onMove(PlayerMoveEvent event) {
        if (event.getPlayer().hasMetadata(INFECTED_METADATA)) {
            event.getPlayer().sendMessage("You are infected! You cannot move!");
            event.setCancelled(true);
        }
    }
}
```

In this example, if a player is hit by another player while in a MyArena, they will be "infected" for a short period of time. The duration for which they are infected is pulled from the **infectionTime** variable specified in your arena YML. While this is a very simple example, it demonstrates how to use the event system in BattleArena.

As a reminder, **no game specific variables** should be stored in this Arena class. As an example, this would be **wrong:**

```
    // Do NOT do this - MyArena only exists ONCE, and this map will leak
    // across ALL competitions of type MyArena
    private final Set<UUID> infectedPlayers = new HashSet<>();

    @ArenaEventHandler
    public void onDamageEntity(EntityDamageByEntityEvent event) {
        if (event.getDamager() instanceof Player damager && event.getEntity() instanceof
Player player) {
            // Player is not infected, let's infect them :)
            if (!this.infectedPlayers.contains(player.getUniqueId())) {
                this.infectedPlayers.add(player.getUniqueId());
                player.sendMessage("You have been infected!");
                damager.sendMessage("You have infected " + player.getName() + "!");

                // Infect the player for the given duration
                Bukkit.getScheduler().runTaskLater(MyPlugin.getInstance(), () -> {
                    this.infectedPlayers.remove(player.getUniqueId());
                    player.sendMessage("You are no longer infected!");
```

```
            }, this.infectionTime.toMillis() / 50);
        }
    }
}

@ArenaEventHandler
public void onMove(PlayerMoveEvent event) {
    if (this.infectedPlayers.contains(event.getPlayer().getUniqueId())) {
        event.getPlayer().sendMessage("You are infected! You cannot move!");
        event.setCancelled(true);
    }
}
```

Up next is creating a custom map and competition class, which will further extend the functionality above and demonstrate how to store per-game logic the correct way for **each** game individually.

Per-Competition Code

Now that you have an Arena class with functioning game logic, it's time to expand on that.

Creating Competition Classes

As mentioned in the Adding Game Logic page, you will need to create a custom Competition class. The following code below is how to create a custom Competition:

```
public class MyCompetition extends LiveCompetition<MyCompetition> {

    public MyCompetition(MyArena arena, CompetitionType type, ArenaMap map) {
        super(arena, type, map);
    }

}
```

As seen above, you will need to extend the **LiveCompetition** class, which is created for competitions that are live on the server BattleArena is running on. On it's own, this will do nothing, so the next step is to create a custom **ArenaMap** which is responsible for creating the competition.

```
public class MyCompetitionMap extends LiveCompetitionMap {
    public static final MapFactory FACTORY = MapyFactory.create(MyCompetitionMap.class,
MyCompetitionMap::new);

    public MyCompetitionMap() {
    }

    public MyCompetitionMap(String name, Arena arena, MapType type, String world, @Nullable
Bounds bounds, @Nullable Spawns spawns) {
        super(name, arena, type, world, bounds, spawns);
    }

    // Override this method in order to use your custom competition class
    @Override
    public LiveCompetition<?> createCompetition(Arena arena) {
        if (!(arena instanceof MyArena myArena)) {
```

```
            throw new IllegalArgumentException("Arena must be an instance of MyArena!");
        }

        return new MyCompetition(myArena, arena.getType(), this);
    }
}
```

As mentioned in an earlier segment of this documentation, maps can exist without necessarily having a competition bound to them, meaning they are responsible for actually creating a live competition. In the map class above, it can be seen the **createCompetition** method is overridden to instead create an instance of our **MyCompetition** class.

Additionally, a **MapFactory** is specified at the top of the class - this is important for the next step of linking this to your **MyArena** so BattleArena knows which ArenaMap (and therefore, Competition) to create for your Arena. It is also **very important** that both constructors are specified as seen in the example above.

And finally, the last step is to override the **getMapFactory** method in **MyArena**, and specify your factory like so:

```
public class MyArena extends Arena {
    @ArenaOption(name = "infection-time", description = "How long a player should be infected
once hit.")
    private Duration infectionTime = Duration.ofSeconds(5);

    private final Set<UUID> infectedPlayers = new HashSet<>();

    @Override
    public MapFactory getMapFactory() {
        return MyCompetitionMap.FACTORY;
    }

    ...
  }
```

Per-Competition Code

Now that we have all the necessary classes created, you can now start creating code that will exist on a per-competition level. If we wanted to infect a random player every minute for instance the following could be done like so:

```
public class MyCompetition extends LiveCompetition<MyCompetition> {

    private BukkitTask tickTask;

    public MyCompetition(MyArena arena, CompetitionType type, LiveCompetitionMap map) {
        super(arena, type, map);
    }

    public void startInfectTask() {
        this.tickTask = Bukkit.getScheduler().runTaskTimer(this.getArena().getPlugin(),
this::infectPlayer, 0, 60 * 60 * 20);
    }

    public void stopInfectTask() {
        if (this.tickTask != null) {
            this.tickTask.cancel();
        }

        this.tickTask = null;
    }

    private void infectPlayer() {
        MyArena arena = (MyArena) this.getArena();

        // Infect a random player
        List<ArenaPlayer> uninfectedPlayers = this.getPlayers().stream().filter(player ->
!arena.isInfected(player)).toList();
        if (uninfectedPlayers.isEmpty()) {
            return;
        }

        ArenaPlayer player = uninfectedPlayers.get((int) (Math.random() *
uninfectedPlayers.size()));
        arena.infect(player.getPlayer());
    }
}
```

And with those changes, we also need to update **MyArena** to actually call the start and stop methods. Here is what the updated **MyArena** class would look like:

```
public class MyArena extends Arena {
    private static final String INFECTED_METADATA = "infected";

    @ArenaOption(name = "infection-time", description = "How long a player should be infected once hit.")
    private Duration infectionTime = Duration.ofSeconds(5);

    @Override
    public MapFactory getMapFactory() {
        return MyCompetitionMap.FACTORY;
    }

    @ArenaEventHandler
    public void onDamageEntity(EntityDamageByEntityEvent event) {
        if (event.getDamager() instanceof Player damager && event.getEntity() instanceof Player player) {
            // Player is not infected, let's infect them :)
            if (!player.hasMetadata(INFECTED_METADATA)) {
                this.infect(player);

                damager.sendMessage("You have infected " + player.getName() + "!");
            }
        }
    }

    @ArenaEventHandler
    public void onMove(PlayerMoveEvent event) {
        if (event.getPlayer().hasMetadata(INFECTED_METADATA)) {
            event.getPlayer().sendMessage("You are infected! You cannot move!");
            event.setCancelled(true);
        }
    }

    @ArenaEventHandler
    public void onPhaseStart(ArenaPhaseStartEvent event, MyCompetition competition) {
        // Ensure we are ingame
        if (!CompetitionPhaseType.INGAME.equals(event.getPhase().getType())) {
            return;
        }
```

```
        competition.startInfectTask();
    }

    @ArenaEventHandler
    public void onPhaseComplete(ArenaPhaseCompleteEvent event, MyCompetition competition) {
        // Ensure we are ingame
        if (!CompetitionPhaseType.INGAME.equals(event.getPhase().getType())) {
            return;
        }

        competition.stopInfectTask();
    }

    public boolean isInfected(ArenaPlayer player) {
        return player.getPlayer().hasMetadata(INFECTED_METADATA);
    }

    public void infect(Player player) {
        player.sendMessage("You have been infected!");
        player.setMetadata(INFECTED_METADATA, new FixedMetadataValue(MyPlugin.getInstance(),
true));

        // Infect the player for the given duration
        Bukkit.getScheduler().runTaskLater(MyPlugin.getInstance(), () -> {
            player.removeMetadata(INFECTED_METADATA, MyPlugin.getInstance());
            player.sendMessage("You are no longer infected!");
        }, this.infectionTime.toMillis() / 50);
    }
}
```

As seen above, when the game enters the **in-game** phase, we start running our task in the active **MyCompetition** to infect a random player every minute. Once the competition is no longer ingame, we stop the task.

Storing Map Information

In some competitions, it may be necessary to store additional information that varies on a per-map basis. An example of this is predefined layers in spleef. The game needs to know where the layers exist in order to handle logic such as only allowing block break for the layers, or allowing the layers to decay. ArenaSpleef makes use of this functionality, and will be used as a reference point at various points in this tutorial.

Storing Information in Your Map

As mentioned earlier in this set of documentation, no game logic should exist in your **CompetitionMap** class. Like **Arena**, it should only contain information and configuration. In the example of spleef, SpleefMap contains all the stored layers, but the actual logic to decay the layers, or handle block breaking is handled in the relevant SpleefCompetition or SpleefArena.

In our example, we will store a set of infection points which will infect players when they walk into them:

```
public class MyCompetitionMap extends LiveCompetitionMap {
    public static final MapFactory FACTORY = MapFactory.create(MyCompetitionMap.class,
MyCompetitionMap::new);

    @ArenaOption(name = "infection-points", description = "The points where players can be
infected.")
    private List<Bounds> infectionPoints;

    public MyCompetitionMap() {
    }

    public MyCompetitionMap(String name, Arena arena, MapType type, String world, @Nullable
Bounds bounds, @Nullable Spawns spawns) {
        super(name, arena, type, world, bounds, spawns);
    }

    public List<Bounds> getInfectionPoints() {
        return this.infectionPoints == null ? List.of() : List.copyOf(this.infectionPoints);
    }
```

```
    public void addInfectionPoint(Bounds bounds) {
        this.infectionPoints.add(bounds);
    }

    public void removeInfectionPoint(Bounds bounds) {
        this.infectionPoints.remove(bounds);
    }

    @Override
    public LiveCompetition<?> createCompetition(Arena arena) {
        if (!(arena instanceof MyArena myArena)) {
            throw new IllegalArgumentException("Arena must be an instance of MyArena!");
        }

        return new MyCompetition(myArena, arena.getType(), this);
    }
}
```

With the code above, BattleArena is now capable of loading a list of **Bounds** from the maps YML file. **Bounds** is a class provided by BattleArena, which simply let you store a minimum and maximum position. It will be used in our case, as we will be able to store a list of bounds throughout various points in the map to infect players.

Now that we have the necessary code in order for this to work, we will want to create a way for a user to add bounds to the map. For this, we will be creating a custom command executor.

Creating a Custom Command Executor

BattleArena provides support for creating custom command executors for individual arenas. Each arena has its own executor that is utilized when running the **/<arena>** command (i.e. /arena, /battlegrounds, etc.). Continuing from this tutorial, there is also a **/myarena** executor used, and we will be expanding upon this in order to add a custom command that lets users specify infection points.

Creating the Executor

The first step in creating custom commands is actually extending the executor. This can simply be done by extending the **ArenaCommandExecutor** class like so:

```
public class MyCommandExecutor extends ArenaCommandExecutor {

    public MyCommandExecutor(MyArena arena) {
        super(arena);
    }
}
```

Now that the class is created, inside your **MyArena** you will want to override the **createCommandExecutor** and provide your own like so:

```
public class MyArena extends Arena {
    ...

    @Override
    public ArenaCommandExecutor createCommandExecutor() {
        return new MyCommandExecutor(this);
    }

    ...
}
```

Adding Commands

Adding commands using BattleArena's command system is designed to be dead simple. The system is annotation-based, and supports a number of built-in parameter types. An example command (without any logic yet) for adding infection points is shown below:

```
@ArenaCommand(commands = "point", subCommands = "add", description = "Adds an infection point
to a MyArena competition.", permissionNode = "point.add")
public void addPoint(Player player, CompetitionMap map, Position min, Position max) {
    if (!(map instanceof MyCompetitionMap myMap)) {
        return; // Should not happen but just incase
    }

    ...
  }
```

To break it down further:

@ArenaCommand

This is an annotation added over a method to designate the method as a command method. This contains the command name, sub commands, the command description and the permission node. This means when running **/myarena point add**, the method shown above will be used. Additionally, the permission designated to this command will be **battlearena.command.myarena.point.add**.

Method Parameters

BattleArena dynamically reads which argument to use from the method parameters. The parameter, Player, is not specified when running the command, but is the player who invoked the command. Everything from that point forward is built into the command, meaning that a user would need to run **/myarena point add <map> <min pos> <max pos>** to successfully invoke the command.

Parsing the Parameters

You might be thinking, how does BattleArena know to read a position from a string, or a map from a string. The simplest answer to that is BattleArena has custom logic for a various number of classes. As this command system is designed to be dynamic, it will handle most of the busy work for you, so instead of having to read a value from a string, or parse a number, BattleArena does that automatically and so the command example above needs no additional work from you.

Adding the Logic

Now that the command method exists, it's time to add some logic to it. Fortunately, this is very easily done:

```
public class MyCommandExecutor extends ArenaCommandExecutor {

    public MyCommandExecutor(MyArena arena) {
        super(arena);
    }

    @ArenaCommand(commands = "point", subCommands = "add", description = "Adds an infection
point to a MyArena competition.", permissionNode = "point.add")
    public void addPoint(Player player, CompetitionMap map, Position min, Position max) {
        if (!(map instanceof MyCompetitionMap myMap)) {
            return; // Should not happen but just incase
        }

        myMap.addInfectionPoint(new Bounds(min, max));

        try {
            myMap.save();
        } catch (ParseException | IOException e) {
            BattleArena.getInstance().error("Failed to save map file for arena {}",
this.arena.getName(), e);
            Messages.MAP_FAILED_TO_SAVE.send(player, map.getName());
            return;
        }
    }
}
```

As we already created a method in the previous page for adding an infection point, it simply just needs to be invoked here. Additionally a save method is then called to save the value we just added to the config. Since we now have a method to add an infection point, we may also want to add a method to remove it. This can simply be done by adding the following:

```
@ArenaCommand(commands = "point", subCommands = "remove", description = "Removes an infection
point from a MyArena competition.", permissionNode = "point.remove")
public void removePoint(Player player, CompetitionMap map, Position min, Position max) {
    if (!(map instanceof MyCompetitionMap myMap)) {
        return; // Should not happen but just incase
    }

    myMap.removeInfectionPoint(new Bounds(min, max));
```

```
    try {
        myMap.save();
    } catch (ParseException | IOException e) {
        BattleArena.getInstance().error("Failed to save map file for arena {}",
this.arena.getName(), e);
        Messages.MAP_FAILED_TO_SAVE.send(player, map.getName());
        return;
    }
}
```

Tying it all Together

Now that we have infection points that players can add in-game using a command, it's time to make those infection points useful. Inside our **MyCompetition** class, let's add a task that uses these infection points:

```
public class MyCompetition extends LiveCompetition<MyCompetition> {

    private BukkitTask tickTask;
    private BukkitTask infectTask;

    public MyCompetition(MyArena arena, CompetitionType type, LiveCompetitionMap map) {
        super(arena, type, map);
    }

    public void startInfectTask() {
        this.tickTask = Bukkit.getScheduler().runTaskTimer(this.getArena().getPlugin(),
this::infectPlayer, 0, 60 * 60 * 20);
        this.infectTask = Bukkit.getScheduler().runTaskTimer(this.getArena().getPlugin(),
this::checkInfectionPoints, 0, 20);
    }

    public void stopInfectTask() {
        if (this.tickTask != null) {
            this.tickTask.cancel();
        }

        if (this.infectTask != null) {
            this.infectTask.cancel();
        }
```

```
        this.tickTask = null;
        this.infectTask = null;
    }

    private void checkInfectionPoints() {
        MyArena arena = (MyArena) this.getArena();
        MyCompetitionMap map = (MyCompetitionMap) this.getMap();

        for (ArenaPlayer player : this.getPlayers()) {
            if (arena.isInfected(player)) {
                continue;
            }

            for (Bounds bounds : map.getInfectionPoints()) {
                if (bounds.isInside(player.getPlayer().getLocation())) {
                    arena.infect(player.getPlayer());
                    break;
                }
            }
        }
    }

    ...
}
```

Creating BattleArena Modules or Extensions

An overview on how to create custom modules or extensions for BattleArena.

Using the BattleArena API

BattleArena offers an API for developers to use if they wish to interface with the plugin.

The main entrypoint into BattleArena is the **BattleArenaApi** class, which can be accessed by running **BattleArenaApi.get()**.

The more important methods are documented below, but the full javadoc can be found here.

- **isInArena(Player)** - Returns whether the given Player is in an Arena
- **arena(String) / getArena(String)** - Returns the Arena from the given name, or null if not found
- **getArenas()** - Returns all the Arenas for the plugin
- **registerArena(Plugin, String, Class, Supplier)** - Registers the given Arena
- **getCompetitions(Arena)** - Returns all the existing Competitions for the Arena
- **getCompetitions(Arena, String)** - Returns all the Competitions for the given Arena and map name
- **findJoinableCompetition(List<Competition>, Player, PlayerRole)** - Finds a joinable competition for the given player with a player role

Useful Methods

A few methods that may be useful to plugin developers is also documented below.

Getting all players in an Arena

This shows how to get all players for a specific game across all maps and active competitions.

```
public List<Player> getAllPlayers(Arena arena) {
    List<Player> players = new ArrayList<>();

    for (Competition<?> competition : BattleArenaApi.get().getCompetitions(arena)) {
        // Ensure that the competition is occurring on this server
        if (!(competition instanceof LiveCompetition<?> liveCompetition)) {
            continue;
        }

        for (ArenaPlayer arenaPlayer : liveCompetition.getPlayers()) {
            players.add(arenaPlayer.getPlayer());
        }
```

```
    }

    return players;
}
```

Getting all maps

This shows how to get all maps that exist for an arena.

```
public List<? extends CompetitionMap> getMaps(Arena arena) {
    return BattleArenaApi.get().getMaps(arena);
}
```

Joining an arena

This shows how to make a player join an arena.

```
public void joinArena(Arena arena, Player player) {
    if (BattleArenaApi.get().isInArena(player)) {
        // Player is already in an arena
        return;
    }

    // Get all the competitions currently running
    List<Competition<?>> competitions = BattleArenaApi.get().getCompetitions(arena);

    // Search to see if there is a joinable competition for the player
    BattleArenaApi.get().findJoinableCompetition(competitions, player,
PlayerRole.PLAYING).thenAccept(result -> {
        JoinResult joinResult = result.result();

        // Success! They can join the competition
        if (joinResult.canJoin()) {
            result.competition().join(player, PlayerRole.PLAYING);
            return;
        }

        // The player cannot join the competition for some reason. This could be for any
reason:
        // - All competitions are full/running
        // - No maps have been set up
        // - No competitions are running
```

```
        // It is usually best practice to send the player a message as to why they cannot join
        joinResult.message().send(player);
    });
}
```