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;
                }
            }
        }
    }

    ...
}
```

Revision #2
Created 14 December 2024 15:03:46 by Redned
Updated 14 December 2024 15:27:53 by Redned