MondoCommand Documentation

Release 0.3

James Crasta

September 30, 2014

CHAPTER 1

Getting Started

MondoCommand is a library for Bukkit applications which lets you create a command which dispatches a number of sub-commands along with generating pretty-looking help pages, checking arguments, and so on.

It also abstracts out a lot of the annoying things about doing subcommands, like return values, argument handling, and so on to help you write shorter, cleaner code (and split it up nicely too).

1.1 Basic usage

Do this in your onEnable or something similar:

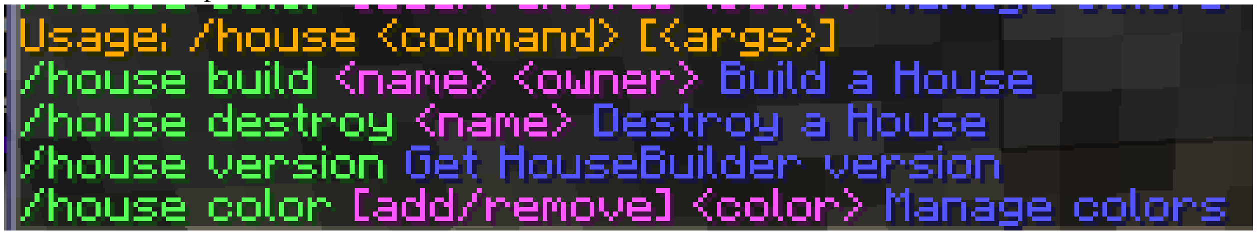
```java
// Basic setup and registration
MondoCommand base = new MondoCommand();
base.autoRegisterFrom(this);
getCommand("housebuilder").setExecutor(base);
```

Now you can add some handlers:

```java
@Sub(description="Build a House", minArgs=2, usage="<owner> <name>")
public void build(CallInfo call) {
    String owner = call.getArg(0);
    String name = call.getArg(1);
    if (houseMap.containsKey(name)) {
        call.reply("House with name %s already exists", name);
    } else {
        // TODO add code to actually make a house
        call.reply("House %s made!", name);
    }
}

@Sub(description="Destroy a House", permission="housebuilder.destroy",
     minArgs=1, usage="<name>", allowConsole=false)
public void destroy(CallInfo Call) {
    // We don't need to check number of args, because we registered the
    // command with minArgs = 1.
    String name = call.getArg(0);
    if (houseMap.containsKey(name)) {
        houseMap.remove(name);
        call.reply("{GREEN}House {GOLD}%s{GREEN} removed", name);
    } else {
        call.reply("{RED}House %s not found", name);
    }
}
```

This creates an output which looks like this:



```
Usage: /house <command> [<args>]
/house build <name> <owner> Build a House
/house destroy <name> Destroy a House
/house version Get HouseBuilder version
/house color [add/remove] <color> Manage colors
```

1.1.1 Things you can do with CallInfo

All MondoCommand handlers take a single argument of type CallInfo which contains all the information you'd normally get from a command handler, plus a bunch of added bits to make writing commands much more convenient:

- `call.getPlayer()` gets a Player object (no more casting from CommandSender) and commands can be registered as allowing console or player-only.
- `call.reply(template, [...])` This is the gem of MondoCommand, it will send a message back to the user that interprets color codes embedded in the string, and lets you also interpolate variables into the string without having to do string concatenation. See below for how to use call.reply.
- `call.getArg(index)` To get a single argument, where the 0th index is the first index which comes after the sub command name (no more argument math!) and furthermore, you don't need to check the length of the args if you registered the subcommand with [B]setMinArgs()[/B], it will show the player a usage message and stop them from running your command.
- `call.getIntArg(index)` - Convenient way to get an argument coerced into an integer.
- `call.getJoinedArgsAfter(index)` - If you need to get a bunch of arguments after a certain index (like say you're accepting a text entry or chat message) this convenience method does that for you.
- `call.numArgs()` - Get the number of arguments.

Using with Maven

MondoCommand is best if used with Maven

Add this repository:

```
<repository>
    <id>crast-repo</id>
    <url>http://maven.crast.us</url>
</repository>
```

And this dependency:

```
<dependency>
    <groupId>us.crast</groupId>
    <artifactId>MondoCommand</artifactId>
    <version>0.3</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
```

And also remember to shade the package so that there's no conflicts with other plugins.

```
<relocation>
    <pattern>mondocommand</pattern>
    <shadedPattern>me.something.your-project-name.mondocommand</shadedPattern>
</relocation>
```

CHAPTER 3

Customizing Formatting

If you don't like the default formatting of MondoCommand, the formatting can be customized in various ways:

3.1 Color Roles

Internally, MondoCommand uses color "roles" to represent various colors. This allows you to write code that, for example, has colors changeable by your users. Also, it lets you customize the colors of output from MondoCommand.

Here are the roles which MondoCommand uses:

- *{HEADER}* Heading of any multi-line replies
- *{USAGE}* The usage portion of command help (usually describes arguments). Default: LIGHT_PURPLE
- *{WARNING}* Used when permissions do not match. Default: DARK_RED
- *{ERROR}* Used when there's an error in the form of a MondoFailure
- *{NOUN}* Used to describe an object. (currently unused by MondoCommand) Default: AQUA
- *{VERB}* Used to describe an action. Default: GRAY
- *{MCMD}* Used for the command portion of the usage line. Default: GREEN
- *{DESCRIPTION}* The description of the line in the usage line. Default: BLUE

So one way to change how MondoCommand looks is by changing the role aliases:

```
ChatMagic.registerAlias("{HEADER}", ChatColor.GOLD);
ChatMagic.registerAlias("{USAGE}", ChatColor.LIGHT_PURPLE);
```

You can also use any of these roles in your own code:

```
String action_text = "expand";
call.reply("About to {VERB}%s{RESET} the house belonging to {NOUN}%s",
           action_text, playerName);
```

3.2 Format Usage Strings

In addition, many of the usage strings can be customized using the FormatConfig class:

```
FormatConfig fmt = new FormatConfig()
    .setUsageHeading("{GOLD}Usage: ")
    .setUsageCommandSuffix(" {GREEN}<command> [arg...]")
    .setReplyPrefix("{RED}My{GREEN}App: ")
    .setPermissionWarning("{RED}No permissions to perform this action.");

MondoCommand base = new MondoCommand(fmt);
```

CHAPTER 4

Working with SubCommands

The SubCommand is the basic building block of MondoCommand. Each MondoCommand has many subcommands, and each subcommand has:

name name of this command

description text shown in the help line about this subcommand

usage text explaining the parameters of this subcommand

permission (optional) A string which is the permission to check to run this subcommand

handler A SubHandler which performs the action of this subcommand

4.1 Registering SubCommands

Note: In *Getting Started*, you saw registration of subcommands using the `@Sub` method annotation. The annotation is a very compact and handy way to register commands, but the underlying API allows better control. In actuality, using the @Sub annotation just causes anonymous handlers to be built for the underlying API.

Let's start with a simple example, registering your first command.

```
// Add sub-command "build" which requires permission "housebuilder.build"
base.addSub("build", "housebuilder.build")
    .setDescription("Build a House")      // Description is shown in command help
    .setMinArgs(2)                        // Won't run command without this many args
    .setUsage("<owner> <name>")           // Sets argument usage information
    .setHandler(new HouseBuildHandler());
```

This example references a theoretical HouseBuildHandler which we haven't created yet, but we will get into that soon. The most important things to note are:

1. You build a sub-command by chaining, similar to some other API's like the conversations API.
2. All sub-commands can be gated by an optional permission, if the user doesn't have this permission they can't access the subcommand nor will it show to them in help.
3. You can set useful metadata like the description, usage arguments, and the minimum number of arguments accepted.
4. Sub-commands delegate the action to a SubHandler which handles the actual implementation.

So now let's get to the code of the HouseBuildHandler:

```
class HouseBuildHandler implements SubHandler {
    public void handle(CallInfo call) throws MondoFailure {
        String owner = call.getArg(0);
        String name = call.getJoinedArgsAfter(1);
        if (houseMap.containsKey(name)) {
            call.reply("House with name %s already exists", name);
        } else {
            // TODO add code to actually make a house
            call.reply("House %s made!", name);
        }
    }
}
```

This code is basically the same as the `build` method we wrote in the intro tutorial, except now it's part of a handler class. You may also be wondering about the `throws MondoFailure` part we added. This is a feature which allows for writing code with very good error handling (a must in the world of Minecraft where situations can change rapidly) that still provides good insight to the user. For more on that, jump over to *Better Error Handling*

Handlers can be anything which implement SubHandler, but if you don't want to create a new class for every handler you can also define them inline:

```
// Add a sub-command destroy which requires permissions "housebuilder.destroy"
base.addSub("destroy", "housebuilder.destroy")
    .setDescription("Destroy a House")
    .setMinArgs(1)
    .setUsage("<name>")
    .setHandler(new SubHandler() {
        // This is an example of how to do handlers in-line.
        public void handle(CallInfo call) {
            String houseName = call.getArg(0);
            if (houseMap.containsKey(houseName)) {
                houseMap.remove(houseName);
                // MondoCommand allows you to add messages with color formatting
                call.reply("{GREEN}House {GOLD}%s{GREEN} removed", houseName);
            } else {
                call.reply("{RED}House %s not found", houseName);
            }
        }
    });
```

4.2 Console Commands

Commands can be set to run on the console. When a command is run from the console, `getPlayer()` will return None, but you can use `getSender()` instead if needed.

Example:

```
// Example of a command which works on the console
base.addSub("version")
    .allowConsole()
    .setDescription("Get HouseBuilder version")
    .setHandler(new SubHandler() {
        public void handle(CallInfo call) {
            call.reply("HouseBuilder Version {RED}1.0.5");
        }
    });
```

Note you can use `call.reply()` as usual, including color codes (though some consoles will ignore colors)

4.3 Nested Sub-Commands

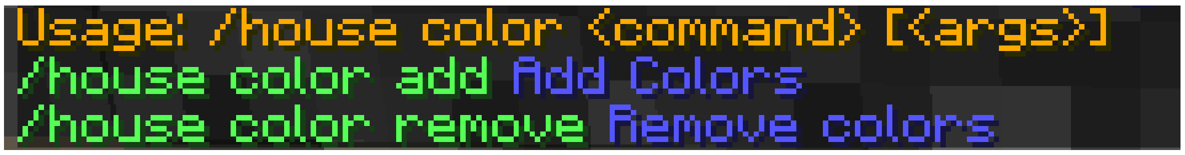
MondoCommand supports doing sub-sub commands by nesting one MondoCommand as a sub of the base (and in theory can handle as many levels as you want)

```
MondoCommand colorSub = new MondoCommand();
base.addSub("color")
    .setDescription("Manage colors")
    .setUsage("[add/remove] <color>")
    .setHandler(colorSub);

colorSub.addSub("add")
    .setDescription("Add Colors");

colorSub.addSub("remove")
    .setDescription("Remove colors");
```

The output of running sub-sub commands looks lke this:



```
Usage: /house color <command> [<args>]
/house color add Add Colors
/house color remove Remove colors
```

4.4 Better Error Handling

One thing MondoCommand is able to do is provide a better error handling flow and also encourage re-usable components by using exceptions to handle the flow and provide descriptive errors back to the users. This is done by use of the *MondoFailure* exception.

To illustrate how this can change your code flow, let's begin first with some basic code which manipulates houses in our theoretical HouseBuilder plugin:

```
@Sub(description="Destroy a House", minArgs=1, usage="<name>")
public void destroy(CallInfo Call) {
    String name = call.getArg(0);
    if (houseMap.containsKey(name)) {
        houseMap.get(name).destroy();
        houseMap.remove(name);
        call.reply("{GREEN}House {GOLD}%s{GREEN} removed", name);
    } else {
        call.reply("{RED}House %s not found", name);
    }
}

@Sub(description="Expand House", minArgs=2, usage="<name> <size>")
public void grow(CallInfo Call) {
    String name = call.getArg(0);
    if (houseMap.containsKey(name)) {
        houseMap.get(name).expand(call.getIntArg(1));
        call.reply("{GREEN}House {GOLD}%s{GREEN} expanded", name);
    } else {
        call.reply("{RED}House %s not found", name);
    }
}
```

You'll notice above that you've more or less duplicated the pieces which deal with finding a house by name in both of those methods, and while you could define a helper, it makes the control flow with providing a clean message to the user much harder. There is another way you can do it, using the *MondoFailure* exception. All handlers for MondoCommand are allowed to throw a MondoFailure exception. The intention of this is to allow you to propagate sensible messages outwards.

```
private House getHouse(String name) throws MondoFailure {
    House house = houseMap.get(name.toLowerCase());
    if (house == null) {
        throw new MondoFailure("House {NOUN}%s{ERROR} not found", name);
    }
    return house;
}

@Sub(description="Destroy a House", minArgs=1, usage="<name>")
public void destroy(CallInfo Call) throws MondoFailure {
    String name = call.getArg(0);
    // MondoFailure is propagated so we don't need to deal with not found situation
    House house = getHouse(name);
    house.destroy();
    houseMap.remove(name);
    call.reply("{GREEN}House {NOUN}%s{GREEN} removed", name);
};

@Sub(description="Expand House", minArgs=2, usage="<name> <size>")
public void grow(CallInfo Call) throws MondoFailure {
    String name = call.getArg(0);
    House house = getHouse(name);
    house.expand(call.getIntArg(1));
    call.reply("{GREEN}House {NOUN}%s{GREEN} expanded", name);
}
```

By using MondoFailure and allowing it to propagate through handlers, now your code is considerably flattened versus the original way it was designed, allowing you to have cleaner command handlers and less duplicated code.