Commands Guide

Proxy Commands

Command class

A base proxy command is represented by a class extending `dev.waterdog.command.Command`.

Constructor

The constructor of the class requires two arguments:

`name: string` `settings: dev.waterdog.command.CommandSettings`.

The name argument is the name of the command, as it would be written when you want to execute it. The settings object contains information like command description, usage message, required permission etc.

A `CommandSettings` object can be simply created using the CommandSettings.builder() function. Using the `Builder` class, you can create the CommandSettings object in a single line of code, finishing with `Builder#build()`. This can look like this:

```
public InfoCommand() {
   super("wdinfo", CommandSettings.builder()
       .setDescription("waterdog.command.info.description")
       .setUsageMessage("waterdog.command.info.usage")
       .setPermission("waterdog.command.info.permission")
       .build());
}
```

That can be passed aswell when creating the command object without overwriting the constructor. That said, overwriting the constructor looks more clean.

Execute Function

The central function every Command needs to inherit and overwrite is `boolean onExecute(CommandSender sender, String alias, String[] args)`. Sender represents the proxy entity which executed the command. As both the `ProxiedPlayer` and the `ConsoleCommandSender` class implement that interface, it is required to check if the sender provided is a player or not, at least when you are performing any player-specific actions (sending popups, transferring..).

Alias is provided when the player used an alias of the command to trigger the command. If he uses the base command name, that parameter will be `null`.

Args is an array of all the arguments provided with a command, excluding the command itself. For the command `/test 123 Hello true`, the args array would look like this: `{"123", "Hello", "true"}`. As you can see, you still need to perform manual user input sanatizing and proper type casting, as we provide all the arguments as strings.

Client-Sided autocompletion

Many people might want to use client-sided autocompletion to reduce the time to type a command.

Warning: You need to enable `inject_proxy_commands` in your `config.yml` in order for this to work

Doing that looks comp 

nal string argument by default. This behaviour can be overwritten. Every command class has the method `public CommandData craftNetwork()`. This method is defining the above behaviour by default using the following code

```
public CommandData craftNetwork() {
        CommandParamData[][] parameterData = new CommandParamData[][]{{
                new CommandParamData(this.name, true, null, CommandParamType.TEXT, null,
Collections.emptyList())
        }};
        return new CommandData(this.name, this.getDescription(), Collections.emptyList(),
(byte) 0, null, parameterData);
}
```

Now, I won't go into detail what every single argument does. The two-dimensional

CommandParamData array contains multiple optional overloads (for example: string, int int; or int, int int). Using the protocol class `com.nukkitx.protocol.bedrock.data.command.CommandParamType`, you can define data-types for parameters, which will be displayed client-sided. Every CommandParamData object represents one argument has the following constructor arguments:

- *name: string*: the name of the parameter, as displayed to the client
- *optional: bool*: whether the parameter is optional or not
- *enumData: CommandEnumData*: Client enum data. This can be for example a list of items or other options. Can be tab-completed.
- *type: CommandParamType*: The data type of the command argument. Examples: string, int, player, boolean..
- *postfix: string*: Command postfix, should be null in most cases
- *options: List*: List of additional parameter options, for example disable auto completion

We can now take an example from the Waterdog default `/server` command.

```
    @Override
    public CommandData craftNetwork() {
        CommandParamData[][] parameterData = new CommandParamData[][]{{
                new CommandParamData("server", false, null, CommandParamType.TEXT, null,
Collections.emptyList()),
                new CommandParamData("player", true, null, CommandParamType.TARGET, null,
Collections.emptyList())
        }};
        return new CommandData(this.getName(), this.getDescription(),
Collections.emptyList(), (byte) 0, null, parameterData);
    }
```

Here two parameters are added to the command:

- "server": the server that the you want to transfer to. has the type text, and is optional.
- "player": the player you want to transfer. Has the type tagret(player) and is optional

Command Maps

Command Maps are the "storage" for commands. You register commands and aliases there aswell as unregistering them. They also take care of interpreting command messages. In Waterdog command maps are represented by the `dev.waterdog.command.CommandMap` interface. The ProxyServer holds an instance of an class implementing that interface.

SimpleCommandMap

The `SimpleCommandMap` class implements default behaviour for command registering, unregistering aswell as parsing as we know it in vanilla. It uses the command prefix / and takes care of permission checking and success checking.

DefaultCommandMap

The `DefaultCommandMap` class extends the previously mentioned SimpleCommandMap and does nothing special except registering the very basic commands that are shipped with Waterdog.

Command Senders

In the Waterdog API, entities which are able to execute commands are represented by the `CommandSender` interface. It requires the ability to check for permission, the ability to send messages, get the name etc.. When handling a command in onExecute(), you are not passed a `ProxiedPlayer` but a `CommandSender`.

Defaults

By default there are two types of Command Senders: `ProxiedPlayer` and `ConsoleCommandSender`.

ProxiedPlayer

The ProxiedPlayer class or any base class extending it also implements the CommandSender interface. `sendMessage` will send the message to the player InGame and `hasPermission` will check the permission map for the requested permission. `getName` will simply return the players name as received in the LoginPacket.

ConsoleCommandSender

ConsoleCommandSender is used by the Proxy Console. It has all permissions, meaning `hasPermission` returns true in any case, `getName` returns "Console", and `sendMessage` will simply send the message in the console with the `INFO` log level.

CommandSender in onExecute

In onExecute, you get passed a CommandSender instance, but you don't know if its a player or the console or any 3rd party command sender. Because of that you should check, for example using `instanceof`, if the CommandSender is a player. If you don't do that but simply cast the `CommandSender` to `ProxiedPlayer`, the command will throw an exception when executed by the

console. Especially when the command attempts to invoke player-specific actions, for example transfers, checking is required.

Revision #8

Created Tue, Nov 17, 2020 8:23 PM by TobiasDev

Updated Fri, Feb 12, 2021 2:54 PM by Alemiz