Plugins

All you need to know about plugins and related api methods.

Introduction

Maven Project Setup

Create a new maven project in IDE of your choice. We recommend you to use Jetbrains IDE for Java i.e. Intellij IDEA. It comes in two versions that are free and paid.

Open your `pom.xml` file and add the repository in the pom.xml so it can fetch the WaterdogPE Plugin API dependencies

```
<repository>
    <id>waterdog-repo</id>
        <url>https://repo.waterdog.dev/artifactory/main</url>
</repository>
```

Add the dependency for WaterdogPE plugin API

```
<dependency>
    <groupId>dev.waterdog.waterdogpe</groupId>
    <artifactId>waterdog</artifactId>
    <version>1.2.3-SNAPSHOT</version>
    <scope>provided</scope>
</dependency>
```

Add the following build code to allow the plugin.yml and other config files being included in the jar:

```
<build>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
        </resource>
    </resources>
</build>
```

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.

Events Guide

Events allow developers to execute own piece of code when something important happens (fe. player joins). WaterdogPE comes with powerful API which allows developers to create and call own events or handle default events.

Event class

A base event is represented by extending the `dev.waterdog.event.Event` class.

CancellableEvent

If base event class implements `CancellableEvent` interface, the event will be considered as cancellable. We use cancellable events to signalize that some task should be canceled or ignored. CancellableEvent implements these methods:

- `isCancelled() : bool` returns true if the event is cancelled.
- `setCancelled(bool) : void` sets whether the event will be cancelled.

AsyncEvent

Our goal is to use the multi threading feature provided by Java as much as possible. Therefore we have created async events. To mark an event as async, use the `AsyncEvent` annotation. This event will have all the event handlers called asynchronously using a thread pool executor. Async events are supposed to not block the original thread where the event is called from. Using async events is recommended especially for events which don't change any values.
Example of async event:

```
@AsyncEvent
public class TransferCompleteEvent extends Events {
  // Your code here...
```

```
}
```

Event handling

Events are handled using runnables with the event as an argument (consumer). Subscribing to an event is done through the `EventManager` class using the `subscribe(Event, Handler, EventPriority)` method. This will add your method to list of handers which will be executed once event is called. Example of handling an event:

```
public void onEnable() {
  // PlayerChatEvent - the event to be subscribed
  // this::onChat - reference to the method which will be executed
  this.getProxy().getEventManager().subscribe(PlayerChatEvent.class, this::onChat);
}

public void onChat(PlayerChatEvent event) {
  // Getting value provided by event
  ProxiedPlayer player = event.getPlayer();
  // Cancelling event
  event.setCancelled(true);
}
```

Event priority

If the event has more handlers we might want to create, prioritize the order of events. using the `EventPriority` enum we can define priorities per handler. Events with lower priority will be executed first and its values may be changed by handlers with higher priority. By default `EventPriority.NORMAL` is used.
Subscribing with defined priority: `subscribe(PlayerChatEvent.class, this::onChat, EventPriority.HIGHEST)`

Event calling

To call specific event instances we use the `callEvent(Event)` method. This method will schedule

every task from event handlers in prioritized order. If the event has `AsyncEvent` annotation present the method will return `CompletableFuture<Event>` which will be completed once all handlers will be executed. If the event has not annotation preset, `null` will be returned.

Calling non-async event:

```
ProtocolCodecRegisterEvent event = new ProtocolCodecRegisterEvent(protocol, builder);
proxy.getEventManager().callEvent(event);
if (event.isCancelled()){
  // Your code here...
}
```

Calling async event:

```
PlayerLoginEvent event = new PlayerLoginEvent(this);
this.proxy.getEventManager().callEvent(event).whenComplete((futureEvent, exception) -> {
  if (futureEvent.isCancelled()) {
    // Your code here...
  }
});
```

Fallback & Join Handler

Introduction

The `IJoinHandler` and `IReconnectHandler` interfaces are providing plugins with an easy API to change some of the most important parts of your network's behaviour.

Setting the handlers

The ProxyServer object holds one instance of each interface, accessible using `ServerInfo#setReconnectHandler(IReconnectHandler)` and `ServerInfo#setJoinHandler(IJoinHandler)`. **Setting them to null will cause massive issues**. Rather implement NO-OP handlers.

IJoinHandler

The `IJoinHandler` interface only requires one method to be implemented, namely the `determineServer(ProxiedPlayer)` method. This method is called whenever a player connects **to the proxy**.
This method can only return an instance of `ServerInfo` or `null`. If the method returns a ServerInfo object, the player's initial connection while be established to that ServerInfo.
If null is returned, the player will be disconnected from the proxy as there is no server available for it to use.

Use case

This method can be used perfectly if you are having multiple lobby-instances in your network (for example). You can then implement f.e. a Round-Robin determination model to evenly distribute players over your lobby-instances. You could also send player to servers depending on where they

were last, or depending on any other set of conditions that you would like to enforce.

IReconnectHandler

The `IReconnectHandler` is called whenever a player is disconnected from a downstream server. This can be caused by a kick, a server shutdown or even a Proxy <-> Downstream timeout. This method will then be called in order to determine the future of the player.
The interface only requires the implementation of the `getFallbackServer(ProxiedPlayer, ServerInfo, String)` method, where the ServerInfo is the information holder of the downstream server the player was disconnected from, and the String is the reason the player was disconnected with (if given).

Use case

In some network concepts, you could want to prevent players from being kicked from the network. This could be the case f.e. for minigames-servers where servers might crash / close down, but you'd still want the player to stay on the proxy but instead be sent to your lobby. In that case you'd just return the ServerInfo of the lobby to transfer the player to.

Important is that you can filter this input by using the `kickMessage` method parameter. With that you could catch players which are being kicked for "Internal Server Error" or "Server closed", but still completly disconnect players that are kicked for "You are banned" or "You have been kicked". If you are returning a ServerInfo object, you can also send the player titles, text messages or other types of output to notify him of the disconnect.

Important note

This note is regarding the performance of this system. You **should not** execute any time-expensive code in either of these methods, as that causes some players to lag while the code is running. Instead, try to run f.e. SQL queries periodically in the background, store the results easily usable in-memory and access those results in the method.

Examples

Setting the handlers

```
ProxyServer server = ProxyServer.getInstance();
IReconnectHandler reconnectHandler = new MyCustomReconnectHandler();
IJoinHandler joinHandler = new MyCustomJoinHandler();
server.setJoinHandler(joinHandler);
server.setReconnectHandler(reconnectHandler);
```

Custom IJoinHandler

A simple example plugin

```
public class VanillaJoinHandler implements IJoinHandler {

    private final ProxyServer server;

    public VanillaJoinHandler(ProxyServer server) {
        this.server = server;
    }

    @Override
    public ServerInfo determineServer(ProxiedPlayer player) {
        return
this.server.getServer(this.server.getConfiguration().getPriorities().get(0));
    }
}
```

Excerpt of the WaterdogPE code.

Returns the first server from the server priority list.

Custom IReconnectHandler

```
public class TestFallbackHandler implements IReconnectHandler {
    @Override
    public ServerInfo getFallbackServer(ProxiedPlayer proxiedPlayer, ServerInfo serverInfo,
String s) {
        for(ServerInfo i : proxiedPlayer.getProxy().getServers()){
            if(!i.getServerName().equals(serverInfo.getServerName())){
                return i;
            }
        }
        return null;
    }
}
```

Tries to find a server which is not the server the player was kicked from. If none was found, return null.

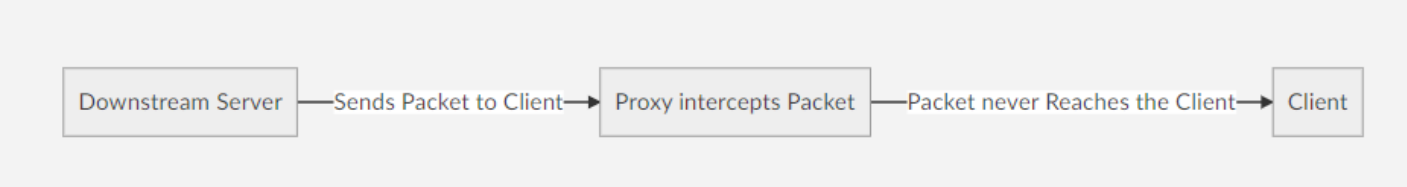
Proxy Communication

Its common that bigger networks requires some synchronization and communication between downstream servers and proxy. There are different ways how communication can be implemented.

Plugin Messages

This type of communication is not supported in WDPE. But understanding how this method works may be useful.

The downstream server sends a packet thought client connection to the proxy. Proxy will then intercept the packet and handle it as it likes, parsing the payload to get request data or whatever the downstream server sent. In specific situation proxy could send packet to client as an additional data exchange (but since 1.16 client is crashing if it receives this packet).



More about plugin messages can be found on Tobias's gist.

Custom socket communication

Creating extra connection between downstreams and proxy can be done thought TCP/UDP sockets. Usually this is the most effective way of data synchronization.
If you are looking for socket solution we recommend to check StarGate, a project developed by one of the WaterdogPE developers, which allows exactly such communication. You can get support with this project on our Discord server.

There are more architectures how could communication be done:

Proxy-Server

This is the simpler implementation which is usually good enough for all networks. In this architecture proxy acts as server for downstream socket clients - manages connections, forwards data between clients. All clients are connected to specific proxy.

Pros:

- Easy data exchange between proxy and downstream.
- Handled data can be easily proceed by proxy itself.

Cons:

- Proxy acts as the master. If proxy went down downstream clients will be disconnected.
- Implementation of custom clients which would be part of the backend system may be harder.
- Using multiple proxy instances would mean multiple open downstream client connections.

Backend-Server

Networks which are using more proxies or needs synchronization with backend system would probably prefer this architecture. Custom application running in backend acts as server. Proxy and downstream servers are in this clients to the application. This application (server) handles received data from downstream client, process it and if needed sends to destination proxy.

Pros:

- Ability to communicate with multiple proxies using one downstream client.
- Ability to communicate between proxies.
- Proxy can be terminated without disconnecting any of downstream clients.

Cons:

- Maintenance of whole stack may be harder.
- Communication between downstream and proxy may be bit slower.

Not recommended solutions

Due to lack of knowledge many users tend to use "poor" solutions which will do their job. Periodically scanning database table, querying results isn't the correct way how to exchange data between downstreams and proxy. We do not recommend any of this methods:

- Using Minecraft query methods to get information from downstreams.
- Any response system based on SQL or database queries.
- Shared file databases (and SQLite) between proxy and downstreams

If you really want to use any of this non-recommended methods consider using Redis instead of file, database or sockets solution.

Scheduling Task

To submit custom repeating, delayed, async tasks WDPE has implemented `WaterdogScheduler`.

Accessing the scheduler

Scheduler can be accessed from proxy instance using `ProxyServer.getScheduler()` or using `WaterdogScheduler.getInstance()`.

Scheduling new task

You have several options to execute task. Here are all methods:

- `scheduleAsync(Runnable task)`: Submit the task to task pool and execute it async (using executor service).
- `scheduleTask(Runnable task, boolean async)`: Submit the task and execute it. Executes task asynchronously if `async = true`.
- `scheduleDelayed(Runnable task, int delay, boolean async)`: Submit the task to queue and execute it after specified delay (in ticks, 1 tick = 50ms). If `async = true` task will be executed asynchronously.
- `scheduleDelayed(Runnable task, int delay)`: This is alias to `scheduleDelayed(Runnable task, int delay, boolean async)` where `async = false`.
- `scheduleRepeating(Runnable task, int period, boolean async)`: Schedule repeatiting task with given period (in ticks). If `async = true` task will be executed every time asynchronously. **Note that task may mot be executed on same thread!**
- `scheduleRepeating(Runnable task, int period)`: Alias to `scheduleRepeating(Runnable task, int period, boolean async)` where `async = false`.
- `scheduleDelayedRepeating(Runnable task, int delay, int period, boolean async)`: Schedule task which will start first time after given delay. Task will repeate in given period. If `async = true` task will be executed every time asynchronously.

- `scheduleDelayedRepeating(Runnable task, int delay, int period)` : Alias to `scheduleDelayedRepeating(Runnable task, int delay, int period, boolean async)` where `async = false`.

Every method return `TaskHandler` which can be used to change executing behaviour. To cancel the task, you can use `TaskHandler.cancel()`. You can also pass `Task` class which implements `Runnable` and contains `onRun(int currentTick)`, `onCancel()` methods. Function `onCancel()` will be called once the task is completed or canceled.

When should I use async task?

We recommend to use asynchronous tasks when your task does not require to be completed after previously submited task. Multi threaded model comes with several benefits such as non-blocking executing. WaterdogPE fully supports multi threaded features and even uses it.
Even is executing task using non-async task will not be executed on main thread. We use single threaded executor which ticks and executes non-async tasks. Asynchronous tasks are executed using `ThreadPoolExecutor` which can execute tasks on any of the threas from the pool.