Velocity Documentation

Release 1.0.0

the Velocity team

Jun 12, 2019

For Server Administrators

Velocity is the next-generation *Minecraft: Java Edition* proxy. Velocity is built to be highly compatible with server software like Paper, Sponge, and modding platforms such as Minecraft Forge, while also exposing a rich plugin API and providing unparalleled scalability.

Note: This website is a work in progress. More documentation is coming soon.

CHAPTER 1

Getting started with Velocity

Velocity is refreshingly easy to set up.

1.1 Installing Java

Velocity is built on Java, so if you do not already have Java installed, you will need to install it before you continue. A discussion about installing Java is out of scope for the Velocity documentation to cover.

1.2 Downloading Velocity

You will need to download Velocity first. Visit the download page and download the latest proxy build from it. Place the downloaded JAR file into a directory just for your proxy. Afterwards, you can run the JAR using `java -jar velocity-proxy-1.0-SNAPSHOT-all.jar`.

1.3 Configuring Your Servers

Once Velocity is up and running, we can move on to configuring your servers for use with Velocity. For now, we're going to get a basic setup going and improve upon it later.

Open up `velocity.toml` and find the `[servers]` section. This section looks like this:

```
[servers]
lobby = "127.0.0.1:30066"
factions = "127.0.0.1:30067"
minigames = "127.0.0.1:30068"
```

Go ahead and put your servers in this file, and then restart Velocity. Once you've done that, you will need to open the `server.properties` file for each of your servers and set the `online-mode` setting to `false`. This allows

Velocity to connect to your server. Once you're done, you should restart your server. Velocity should now be ready to use.

This is a minimal setup. Since we're not forwarding IPs and player information, the Minecraft server will assume you connected from offline mode and will use a different UUID and display only the default Steve and Alex skins. However, Velocity can forward this information onto your Minecraft servers with some extra configuration. See *Configuring player information forwarding* to learn how to configure this feature.

1.4 What's Next?

In this section, you downloaded and added your servers to the `velocity.toml` file. This file is very important for us, so in the next section we'll cover it in great detail.

CHAPTER 2

Commands

Velocity includes a few commands in the core of the proxy by default. You can gain a richer set of commands by adding plugins.

2.1 The /velocity command

The `/velocity` command contains a number of commands to help manage the proxy.

2.1.1 /velocity plugins

If the user has the `velocity.command.plugins` permission, they can view all the plugins currently active on the proxy.

2.1.2 /velocity version

Displays the Velocity proxy version.

2.1.3 /velocity reload

If the user has the `velocity.command.reload` permission, the proxy will read and reconfigure itself from the `velocity.toml` on disk. If there are problems with parsing the file, no changes will be applied.

2.2 The /server command

If the user has the `velocity.command.server` permission (by default, this is granted to all users), players can use this command to view and change servers.

Executing just `/server` will send the user the name of the server they are currently on, along with options to move to other servers configured on the proxy.

If a server name is specified, Velocity will attempt to connect to the server.

2.3 The /shutdown command

When executed from the console, this will gracefully shut down the Velocity proxy. All players will be disconnected from the proxy and plugins will have a chance to finish up before the proxy shuts down.

CHAPTER 3

Configuring Velocity

Velocity has been designed to be simple and unambigous to configure.

3.1 The configuration file

Velocity is largely configured from the `velocity.toml` file. This file is created in the directory where you started the proxy.

3.1.1 The configuration format

Before we continue, it is useful to take a step back and note that Velocity uses the TOML format for its configuration. TOML was designed to be easy to understand, so you should not have difficulty understanding Velocity's configuration file.

3.1.2 Root section

These settings mostly cover the basic, most essential settings of the proxy.

Setting name	Type	Default	Description
`bind`	Address	`0.0.0.0:25577`	This tells the proxy to accept connections on a specific IP. By default, Velocity will listen for connections on all IP addresses on the computer on port 25577.
`motd`	Chat	`&3A Velocity Server`	This allows you to change the message shown to players when they add your server to their server list. You can use legacy Minecraft color codes or JSON chat.
`show-max-players`	Integer	`500`	This allows you to customize the number of "maximum" players in the player's server list. Note that Velocity doesn't have a maximum number of players it supports.
`player-info-forwarding`	Mode	`modern`	This allows you to customize how player information such as IPs and UUIDs are forwarded to your server. See the "Player info forwarding" section for more information.
`player-info-forwarding-secret`	String	`5up3r53cr3t`	This setting is used as a secret to ensure that player info forwarded by Velocity comes from your proxy and not from someone pretending to run Velocity. See the "Player info forwarding" section for more info.
`announce-forge`	Boolean	`false`	This setting determines whether or Velocity should present itself as a Forge/FML-compatible server. By default, this is disabled.

3.1.3 `server` section

Setting name	Type	Default	Description
A server name	Address	See the default configuration below.	This makes the proxy aware of a server that it can connect to.
`try`	Array	`["lobby"]`	This specifies what servers (in order Velocity should try to connect to upon player login and when a player is kicked from a server.

3.1.4 `advanced` section

Setting name	Type	Default	Description
`compression-threshold`	Integer	`256`	This is the minimum size (in bytes) that a packet has to be before the proxy compresses it. Minecraft uses 256 bytes by default.
`compression-level`	Integer	`-1`	This setting indicates what `zlib` compression level the proxy should use to compress packets. The default value uses the default zlib level, which is dependent on the zlib version. This number goes from `0` to `9`, where `0` means no compression and `9` indicates maximum compression.
`login-ratelimit`	Integer	`3000`	This setting determines the minimum amount of time (in milliseconds) that must pass before a connection from the same IP address will be accepted by the proxy. A value of `0` disables the rate limit.
`connection-timeout`	Integer	`5000`	This setting determines how long the proxy will wait to connect to a server before timing out.
`read-timeout`	Integer	`30000`	This setting determines how long the proxy will wait to receive data from the server before timing out. If you use Forge, you may need to increase this setting.
`proxy-protocol`	Boolean	`false`	This setting determines whether or not Velocity should receive HAProxy PROXY messages. If you don't use HAProxy, leave this setting off.

3.1.5 `query` section

Setting name	Type	Default	Description
`enabled`	Boolean	`false`	Whether or not Velocity should reply to GameSpy 4 (Minecraft query protocol) requests. You can usually leave this false.
`port`	Number	`25577`	Specifies which port that Velocity should listen on for GameSpy 4 (Minecraft query protocol) requests.
`map`	String	Velocity	Specifies the map name to be shown to clients.
`show-plugins`	Boolean	False	Whether or not Velocity plugins are included in query responses.

3.2 The default configuration

Below is the default configuration file for Velocity, `velocity.toml`.

Listing 1: velocity.toml

```
# What port should the proxy be bound to? By default, we'll bind to all addresses on␣
↪port 25577.
bind = "0.0.0.0:25577"

# What should be the MOTD? Legacy color codes and JSON are accepted.
motd = "&3A Velocity Server"
```

(continues on next page)

(continued from previous page)

```
# What should we display for the maximum number of players? (Velocity does not␣
↪support a cap
# on the number of players online.)
show-max-players = 500

# Should we authenticate players with Mojang? By default, this is on.
online-mode = true

# Should we forward IP addresses and other data to backend servers?
# Available options:
# - "none":   No forwarding will be done. All players will appear to be connecting␣
↪from the proxy
#             and will have offline-mode UUIDs.
# - "legacy": Forward player IPs and UUIDs in BungeeCord-compatible fashion. Use this␣
↪if you run
#             servers using Minecraft 1.12 or lower.
# - "modern": Forward player IPs and UUIDs as part of the login process using Velocity
↪'s native
#             forwarding. Only applicable for Minecraft 1.13 or higher.
player-info-forwarding = "modern"

# If you are using modern IP forwarding, configure an unique secret here.
player-info-forwarding-secret = "5up3r53cr3t"

# Announce whether or not your server supports Forge/FML. If you run a modded server,␣
↪we suggest turning this on.
announce-forge = false

[servers]
# Configure your servers here.
lobby = "127.0.0.1:30066"
factions = "127.0.0.1:30067"
minigames = "127.0.0.1:30068"

# In what order we should try servers when a player logs in or is kicked from a␣
↪server.
try = [
    "lobby"
]

[advanced]
# How large a Minecraft packet has to be before we compress it. Setting this to zero␣
↪will compress all packets, and
# setting it to -1 will disable compression entirely.
compression-threshold = 256

# How much compression should be done (from 0-9). The default is -1, which uses zlib
↪'s default level of 6.
compression-level = -1

# How fast (in miliseconds) are clients allowed to connect after the last connection?␣
↪Default: 3000
# Disable by setting to 0
login-ratelimit = 3000

# Specify a custom timeout for connection timeouts here. The default is five seconds.
connection-timeout = 5000
```

(continues on next page)

(continued from previous page)

```
# Specify a read timeout for connections here. The default is 30 seconds.
read-timeout = 30000

# Enables compatibility with HAProxy.
proxy-protocol = false

[query]
# Whether to enable responding to GameSpy 4 query responses or not.
enabled = false

# If query is enabled, on what port should the query protocol listen on?
port = 25577

# This is the map name that is reported to the query services.
map = "Velocity"

# Whether plugins should be shown in query response by default or not
show-plugins = false
```

CHAPTER 4

Configuring player information forwarding

Velocity supports forwarding information about your players to your servers, such as IP addresses, UUIDs, and skins. Velocity supports two different methods for forwarding player information to your servers:

- `modern` forwarding is a Velocity-native format. It forwards all player information in an efficient binary format and ensures that nobody tries to trick the server into impersonating your Velocity proxy. However, it is only available for Minecraft 1.13 or higher.
- `legacy` forwarding is the player information forwarding protocol used by BungeeCord. This is extremely compatible across all Minecraft versions that Velocity supports, but requires proper configuration to ensure that nobody pretends to be your proxy by using a firewall or a plugin like IPWhitelist.

4.1 Configuring modern forwarding

Currently, only build 377 and above of Paper 1.13.1+ support Velocity's modern forwarding.

To use modern forwarding with any supported server implementation, set the `player-info-forwarding` setting in `velocity.toml` to `modern`. You must also change the `forwarding-secret` setting to a unique secret. You then need to ensure your server is properly configured to use modern Velocity forwarding.

4.1.1 Paper

To allow Paper to understand the forwarded player data, in your `paper.yml`, set `settings.velocity-support.enabled` to `true` and `settings.velocity-support.secret` to match the secret in your `velocity.toml`. You must also set `settings.velocity-support.online-mode` to the `online-mode` setting in your `velocity.toml`. Once you're done editing `paper.yml`, reboot your server.

4.2 Configuring legacy BungeeCord-compatible forwarding

If you need to use legacy BungeeCord-compatible forwarding, simply set your `player-info-forwarding` setting in `velocity.toml` to `legacy`. You will also need to make sure your server is properly configured to under-

stand the data.

Caution: Legacy BungeeCord-compatible forwarding allows anyone to pretend they are your proxy and allow them to log in under any username or IP address! You must make sure that you have a firewall set up on your servers or use a plugin such as IPWhitelist to make sure your servers are protected.

4.2.1 Spigot / Paper

To make Spigot or Paper understand the data forwarded from Velocity, set `settings.bungeecord` to `true` in your `spigot.yml` and then reboot your server.

4.2.2 Sponge

To configure Sponge to understand the data forwarded from Velocity, set `modules.bungeecord` to `true` and `bungeecord.ip-forwarding` to `true` in your `config/sponge/global.conf` file, and then restart your Sponge server.

CHAPTER 5

Frequently asked questions

5.1 What versions of Minecraft does Velocity support?

Velocity supports Minecraft 1.8-1.14. It is important to note, however, that Velocity does not translate between protocol versions - most packets from the client and server are passed through the proxy unchanged. If you need a multi-protocol solution for your Minecraft server, please consider installing ProtocolSupport or ViaVersion on your backend servers.

5.2 What server software is supported by Velocity?

Velocity aims to support Paper, Sponge, and Minecraft Forge. As of September 7, 2018, Forge support is available and the proxy has been most extensively tested against Paper, although Sponge also runs well.

5.3 Is Velocity compatible with my Forge mod(s)?

Velocity is compatible with Minecraft Forge (1.8-1.12.2) and its legacy player information forwarding is compatible with SpongeForge. Most mods should work without issue and with less issues than with BungeeCord or Waterfall.

However, there are certain mods that are incompatible with the server-switching behavior Velocity employs. These are issues that only the author of the mod can fix, and are not issues with Velocity.

5.4 What is Velocity's performance profile?

On a Velocity server without plugins, most CPU time is spent processing packets (especially decompressing and recompressing) and waiting on network events. Velocity has been tuned for throughput: given enough resources, a single proxy should be able to handle a large number of Minecraft players online.

There are several ways to increase the throughput of the proxy.

5.4.1 Keep an eye on your plugins

The biggest performance killer by far are your plugins! Velocity implements several measures to attempt to reduce issues caused by misbehaving plugins, but these measures are imperfect. It is important you monitor your plugins to ensure they are not hurting your proxy throughput.

5.4.2 Disable compression between the proxy and your backend server

If your backend server has compression enabled (by default, Minecraft servers compress packets larger than 256 bytes), then Velocity is forced to decompress the packets from servers so it can process them, usually only to compress then shortly afterwards because it did not find anything interesting. To eliminate this inefficiency, you should disable compression on your backend server, so that only Velocity is responsible for compressing packets.

To disable compression, simply set `network-compression-threshold=-1` in your `server.properties`, and then reboot your server.

5.4.3 Keep up to date

The Velocity team constantly seeks to improve the throughput of the proxy, and you can only benefit from our efforts if you keep the proxy regularly up-to-date.

CHAPTER 6

Creating your first plugin

So you've decided to take the plunge and create your first Velocity plugin? That's awesome! This page will help you get you going.

6.1 Set up your environment

You're going to need the JDK and an IDE (we like IntelliJ IDEA, but any IDE will work).

6.2 I know how to do this. Give me what I need!

6.2.1 Maven repository

Name	`velocity`
URL	`https://repo.velocitypowered.com/snapshots/`

6.2.2 Dependency

Group ID	`com.velocitypowered`
Artifact ID	`velocity-api`
Version	`1.0.0-SNAPSHOT`

6.3 Setting up your first project

If you need help setting up your project, don't worry!

6.3.1 Set up your build system

You will need to set up a build system before you continue. Discussing how to set up a build system for your project is out of scope for this page, but you can look at the Gradle or Maven documentation for assistance.

Setting up the dependency with Gradle

Add the following to your `build.gradle`:

```
repositories {
    maven {
        name 'velocity'
        url 'https://repo.velocitypowered.com/snapshots/'
    }
}

dependencies {
    compile 'com.velocitypowered:velocity-api:1.0.0-SNAPSHOT'
}
```

Note: As of Gradle 5, you must also specify the API dependency as an annotation processor, otherwise plugin annotations won't be processed into the `velocity-info.json` file.

```
dependencies {
    compile 'com.velocitypowered:velocity-api:1.0.0-SNAPSHOT'
    annotationProcessor 'com.velocitypowered:velocity-api:1.0.0-SNAPSHOT'
}
```

Setting up the dependency with Maven

Add the following to your `pom.xml`:

```
<repositories>
    <repository>
        <id>velocity</id>
        <url>https://repo.velocitypowered.com/snapshots/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>com.velocitypowered</groupId>
        <artifactId>velocity-api</artifactId>
        <version>1.0.0-SNAPSHOT</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

6.3.2 Create the plugin class

Create a new class (let's say `com.example.velocityplugin.VelocityTest` and paste this in:

```
package com.example.velocityplugin;

import com.google.inject.Inject;
import com.velocitypowered.api.plugin.Plugin;
import com.velocitypowered.api.proxy.ProxyServer;
import org.slf4j.Logger;

@Plugin(id = "myfirstplugin", name = "My First Plugin", version = "1.0-SNAPSHOT",
        description = "I did it!", authors = {"Me"})
public class VelocityTest {
    private final ProxyServer server;
    private final Logger logger;

    @Inject
    public VelocityTest(ProxyServer server, Logger logger) {
        this.server = server;
        this.logger = logger;

        logger.info("Hello there, it's a test plugin I made!");
    }
}
```

What did you just do there? There's quite a bit to unpack, so let's focus on the Velocity-specific bits:

```
@Plugin(id = "myfirstplugin", name = "My First Plugin", version = "1.0-SNAPSHOT",
        description = "I did it!", authors = {"Me"})
public class VelocityTest {
```

This tells Velocity that this class contains your plugin (`myfirstplugin`) so that it can be loaded once the proxy starts up. Velocity will detect where the plugin will reside when you compile your plugin.

```
@Inject
public VelocityTest(ProxyServer server, Logger logger) {
    this.server = server;
    this.logger = logger;

    logger.info("Hello there, it's a test plugin I made!");
}
```

This looks like magic! How is Velocity doing this? The answer lies in the `@Inject`, which indicates that Velocity should inject a `ProxyServer` and the `Logger` when constructing your plugin. These two interfaces will help you out as you begin working with Velocity. We won't talk too much about dependency injection: all you need to know is that Velocity will do this.

All you need to do is build your plugin, put it in your `plugins/` directory, and try it! Isn't that nice? In the next section you'll learn about how to use the API.

6.3.3 A word of caution

In Velocity, plugin loading is split into two steps: construction and initialization. The code in your plugin's constructor is part of the construction phase. There is very little you can do safely during construction, especially as the API does not specify which operations are safe to run during construction. Notably, you can't register an event listener in your constructor, because you need to have a valid plugin registration, but Velocity can't register the plugin until the plugin has been constructed, causing a "chicken or the egg" problem.

To break this vicious cycle, you should always wait for initialization, which is indicated when Velocity fires the `ProxyInitializeEvent`. We can do things on initialization by adding a listener for this event, as shown below.

Note that Velocity automatically registers your plugin main class as a listener.

```
@Subscribe
public void onProxyInitialization(ProxyInitializeEvent event) {
    // Do some operation demanding access to the Velocity API here.
    // For instance, we could register an event:
    server.getEventManager().register(this, new PluginListener()):
}
```

CHAPTER 7

The Command API

The Command API lets you create commands that can be executed on the console or via a player connected through the proxy.

7.1 Create the command class

Each command class must implement the Command interface, which has two methods: one for when the command is executed and one to provide suggestions for tab completion. Let's see an example of a simple command that will tell whoever executes the command "Hello World" in light blue text.

```
package com.example.velocityplugin;

import com.velocitypowered.api.command.Command;
import com.velocitypowered.api.command.CommandSource;
import net.kyori.text.TextComponent;
import net.kyori.text.format.TextColor;
import org.checkerframework.checker.nullness.qual.NonNull;

public class CommandTest implements Command {

    @Override
    public void execute(@NonNull CommandSource source, String[] args) {
        source.sendMessage(TextComponent.of("Hello World!").color(TextColor.AQUA));
    }
}
```

Now that we have created the command, we need to register it in order for it to work. To register commands, you use the Command Manager. We get the command manager by executing `proxyServer.getCommandManager()` with the proxy instance, or by injecting it using the `@Inject` annotation in our main class. The register method requires two parameters, the command object and the command aliases which is a varargs parameter.

```
commandManager.register(new CommandTest(), "test");
```

If we assemble it all into our main class created on the first tutorial, it'll look something like this

```
package com.example.velocityplugin;

import com.google.inject.Inject;
import com.velocitypowered.api.command.CommandManager;
import com.velocitypowered.api.plugin.Plugin;
import org.slf4j.Logger;

@Plugin(id = "myfirstplugin", name = "My First Plugin", version = "1.0-SNAPSHOT",
        description = "I did it!", authors = {"Me"})
public class VelocityTest {

    @Inject private VelocityTest(CommandManager commandManager, Logger logger) {
        commandManager.register(new CommandTest(), "test");
        logger.info("Plugin has enabled!");
    }
}
```

As you can see we're injecting the commandManager instance but we can also obtain it by injecting the `ProxyServer` and getting it from there.

7.2 How command arguments work

The execute method has a `String[]` which represents the arguments of the command. The arguments don't include the base command. It is important to note that in the event that no arguments are specified, an empty array will be passed, rather than a null array.

If a player or a console executes the following command: `/stats Player2 kills`, the first argument will be `Player2`, which we can access using `args[0]` and the second argument will be `kills`.

Let's create a command that will return how many kills a player has (which are stored in a local hashmap for the purposes of this tutorial).

The command will be `/stats <player>`

```
package com.example.velocityplugin;

import com.google.common.collect.ImmutableList;
import com.velocitypowered.api.command.Command;
import com.velocitypowered.api.command.CommandSource;
import net.kyori.text.TextComponent;
import net.kyori.text.format.TextColor;
import org.checkerframework.checker.nullness.qual.NonNull;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class TabCompleteTest implements Command {

    private final Map<String, Integer> playerKills = new HashMap<>();

    public TabCompleteTest() {
```

(continues on next page)

(continued from previous page)

```
        playerKills.put("Tux", 58);
        playerKills.put("Player2", 23);
        playerKills.put("Player3", 17);
    }

    @Override
    public void execute(@NonNull CommandSource source, String[] args) {
        if (args.length != 1) {
            source.sendMessage(TextComponent.of("Invalid usage!").color(TextColor.
↪RED));
            source.sendMessage(TextComponent.of("Usage: /stats <player>").
↪color(TextColor.RED));
            return;
        }

        String playerName = args[0];
        if (playerKills.containsKey(playerName)) {
            source.sendMessage(TextComponent
                    .of(playerName + " has " + playerKills.get(playerName) + " kills.
↪")
                    .color(TextColor.GREEN));
        } else {
            source.sendMessage(TextComponent.of("Player not found").color(TextColor.
↪RED));
        }
    }
}
```

Let's break down the command.

```
private final Map<String, Integer> playerKills = new HashMap<>();

public TabCompleteTest() {
    playerKills.put("Tux", 58);
    playerKills.put("Player2", 23);
    playerKills.put("Player3", 17);
}
```

We create a simple map where we'll store dummy players with kills as an example for this tutorial. If you were to create a stat plugin, these players would be loaded from the database or from another file.

```
@Override
public void execute(@NonNull CommandSource source, String[] args) {
    if (args.length != 1) {
        source.sendMessage(TextComponent.of("Invalid usage!").color(TextColor.RED));
        source.sendMessage(TextComponent.of("Usage: /stats <player>").color(TextColor.
↪RED));
        return;
    }
```

We first check that the arguments are equal to 1, meaning they specified a player.

```
String playerName = args[0];
```

We get the player name that was provided in the command. `/stats Player2`, the `playerName` would be `Player2`.

```
if (playerKills.containsKey(playerName)) {
    source.sendMessage(TextComponent
            .of(playerName + " has " + playerKills.get(playerName) + " kills.")
            .color(TextColor.GREEN));
} else {
    source.sendMessage(TextComponent.of("Player not found").color(TextColor.RED));
}
```

Finally do a simple check to see if the player has kills and display them if they do have, or otherwise send them a message that the player is not found.

7.3 Creating a simple tab complete

Tab completion is when a player or the console presses the tab key while writing a command, in which the plugin will automatically give suggestions according to the context of the command. Let's say you're typing `/kill` and then press the tab key, the plugin would suggest the names of the players who are online.

We'll base on the last command example, but will add one thing. The player names who have kills will be able to be completed using the tab key.

```
@Override
public List<String> suggest(@NonNull CommandSource source, String[] currentArgs) {
    if (currentArgs.length == 0) {
        return new ArrayList<>(playerKills.keySet());
    } else if (currentArgs.length == 1) {
        return playerKills.keySet().stream()
                .filter(name -> name.regionMatches(true, 0, currentArgs[0], 0,␣
↪currentArgs[0].length()))
                .collect(Collectors.toList());
    } else {
        return ImmutableList.of();
    }
}
```

Let's break down the `suggest` method.

```
if (currentArgs.length == 0) {
    return new ArrayList<>(playerKills.keySet());
```

If the player hasn't entered anything other than the command, we will suggest all the names in the map.

```
} else if (currentArgs.length == 1) {
    return playerKills.keySet().stream()
            .filter(name -> name.regionMatches(true, 0, currentArgs[0], 0,␣
↪currentArgs[0].length()))
            .collect(Collectors.toList());
```

Now the player has typed something, so we will suggest all the player names that start with the characters that the player has typed. For instance, if the player has typed `Pla` or `Player`, it will suggest `Player2` and `Player3`. If the player has typed `T`, it will suggest `Tux`.

```
} else {
   return ImmutableList.of();
}
```

If the player tries to autocomplete more than one argument, we return an empty list since our command only has one argument.