
Table of Contents

Support me on Patreon	1.1
Introduction	1.2
Frequently Asked Questions	1.3
Getting Started	1.4
Getting Started - Long Version	1.4.1
Getting Started - Linux TL;DR	1.4.2
Getting Started - Windows TL;DR	1.4.3
Your First Bot	1.4.4
Understanding	1.5
Events and Handlers	1.5.1
Collections	1.5.2
Roles	1.5.3
Async / Await	1.5.4
Coding Guides	1.6
Storing Data in a JSON file	1.6.1
Storing Data in an SQLite file	1.6.2
Using PersistentCollections	1.6.3
A Basic Command Handler	1.6.4
Creating a Music Bot	1.6.5
Cleverbot Integration	1.6.6
Discord Webhooks (Part 1)	1.6.7
Discord Webhooks (Part 2)	1.6.8
Using Emojis	1.6.9
Examples	1.7
Welcome Message every X users	1.7.1
Message Reply Array	1.7.2
Command with arguments	1.7.3
Selfbots, the greatest thing in the universe	1.7.4
Making an Eval command	1.7.5
Using Embeds in messages	1.7.6

Other Guides	1.8
Installing and Using a proper editor	1.8.1
Using Git to share and update code	1.8.2
Hosting on a Raspberry Pi	1.8.3
Hosting Music Bots on a Raspberry Pi	1.8.4
Video Guides	1.9
Episode 1	1.9.1
Episode 2	1.9.2
Episode 3	1.9.3
Episode 4	1.9.4
Episode 5	1.9.5
Episode 6 Part 1	1.9.6
Episode 6 Part 2	1.9.7
Episode 7	1.9.8
Episode 8	1.9.9
Episode 9	1.9.10
Episode 10 Part 1	1.9.11

Discord.js Getting Started

Introduction

This guidebook was originally authored by [eslachance#4611](#) then handed down to me for future updates, but when she started the book originally it was because the examples for discord.js and the documentation were quite daunting for newcomers. There was definitely a space for this kind of document online, made obvious by the recurring questions that pop up almost every day on the support channels.

Seeing as though there needed to be some more detailed explanations as well as code samples, She figured a guide would be a great place to start!

To keep with the style of the official documentation, we will be using full terms (client and message) for those variables.

Updating of this guide

I want to take a moment to thank Evie for everything she's done for the community. Now a lot of people's personal opinions will differ from mine, due to various levels of information that's out there.

But to me Evie is a mentor, she was the one that metaphorically took me under her wing and encouraged me to learn new things and to challenge myself, and hey I've got a decently popular YouTube series because of her, she played a major role in my decision to create a series, so please take a moment and think about how many people Evie has helped with this guide.

And I am honoured to be continuing it.

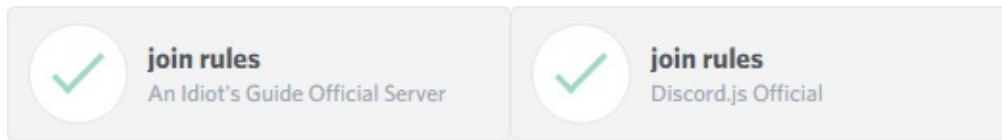


eslachance Today at 3:56 PM

I hereby give permission to York, *The Idiot Himself*, to continue the work I started with the "Discord.js Bot Guide", to modify it in any way he feels necessary, and to publish it through whatever means he desires. This follows the MIT license the guide has been placed under.

Get Support

If you have any questions after reading this guide please join us on the official Discord.js server:



Last Updated: 26/08/2017

On Version: 12.3.5

Frequently Asked Questions

In this page, some very basic, frequently-asked questions are answered. It's important to understand that **these examples are generic** and will most likely not work if you just copy/paste them in your code. You need to **understand** these lines, not just blindly shove them in your code.

Code Examples

Bot and Bot Client

```
// Set the bot's "Playing: " status (must be in an event!)
client.on("ready", () => {
  client.user.setGame("with my code");
});
```

```
// Set the bot's online/idle/dnd/invisible status
client.on("ready", () => {
  client.user.setStatus("online");
});
```

Users and Members

```
// Get a User by ID
client.users.get("user id here");
// Returns <User>
```

```
// Get a Member by ID
message.guild.members.get("user ID here");
// Returns <Member>
```

```
// Get a Member from message Mention
message.mentions.members.first();
// Returns <Member>
```

```
// Send a Direct Message to a user
message.author.send("hello");
```

```
// Mention a user in a message
message.channel.send(`Hello ${user}, and welcome!`);
// or
message.channel.send("Hello " + message.author.toString() + ", and welcome!");
```

Channels and Guilds

```
// Get a Guild by ID
client.guilds.get("the guild id");
// Returns message.guild
```

```
// Get a channel by ID
client.channels.get("the channel id");
// Returns message.channel
```

```
// Get a Channel by Name (note: THIS IS NOT RECOMMENDED as more than one channel can have the same name!)
message.guild.channels.find("name", "channel-name");
// returns message.channel
```

Common Errors & Fixes

Cannot find module `discord.js`

Problem:

You didn't install Discord.js or installed it in the wrong folder

Solution:

- Make sure you are in the **correct** folder where you have your bot's files
- SHIFT+Right-Click in the folder and select **Open command window here**
- Run `npm init -y`, and hit enter until the wizard is complete
- Run `npm i -S discord.js` again to install Discord.

Unexpected End of Input

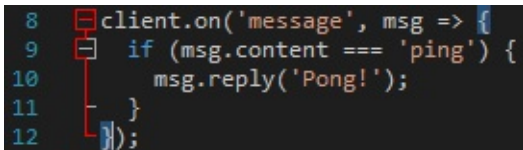
Problem:

```
});  
  ^  
SyntaxError: Unexpected end of input
```

Solution:

Your code has an error somewhere. This is *impossible* to troubleshoot without the **complete** code, since the error can be anywhere (in fact the error stack often tells you it's at the end of your code).

The following trick is a lifesaver, so pay attention: Your code editor is trying to help you. Whatever editor you're using (except notepad++.exe. Don't use notepad++!), clicking on any (and I mean any) special character such as parentheses, square brackets, curly braces, double and single quotes, will automatically highlight the one that matches it. The screenshot below shows this: I clicked on the curly brace at the bottom, it shows me the one on top by highlighting it. Learn this, and how different functions and event handlers "look" like.



```
8 client.on('message', msg => {  
9   if (msg.content === 'ping') {  
10     msg.reply('Pong!');  
11   }  
12 });
```

You can check out [Installing and Using a Proper Editor](#) to help in at least knowing there are errors *before* running your bot code.

Getting Started with Discord.js

So, you want to write a bot and you know some JavaScript, or maybe even node.js. You want to do cool things like a music bot, tag commands, random image searches, the whole shebang. Well you're at the right place!

This is the long version with a whole lot of useless blabbering text, jokes, and explanations.

TL;DR (short) versions: [Windows](#) , [Linux](#)

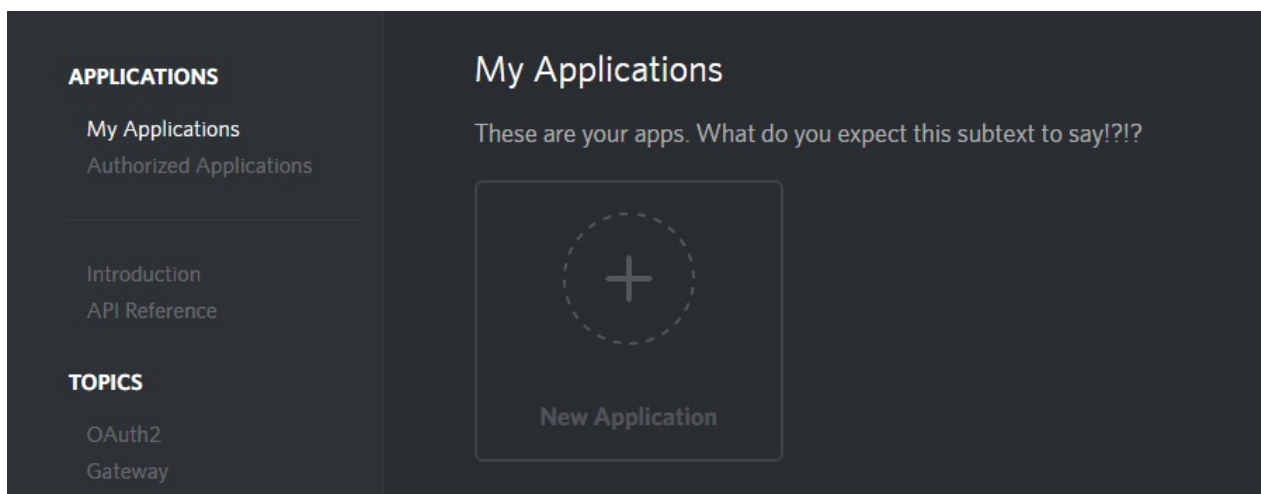
This tutorial will get you through the first steps of creating a bot, configuring it, making it run, and adding a couple of commands to it.

Step 1: Creating your App and Bot account

The first step in creating a bot is to create your own Discord *application*. The bot will use the Discord API, which requires the creation of an account for authentication purposes. Don't worry though, it's super simple.

Creating the App account

To create the application, head to the [Discordapp.com Application Page](#). Assuming you're logged in (if not, do so now), you'll reach a page that looks like this:

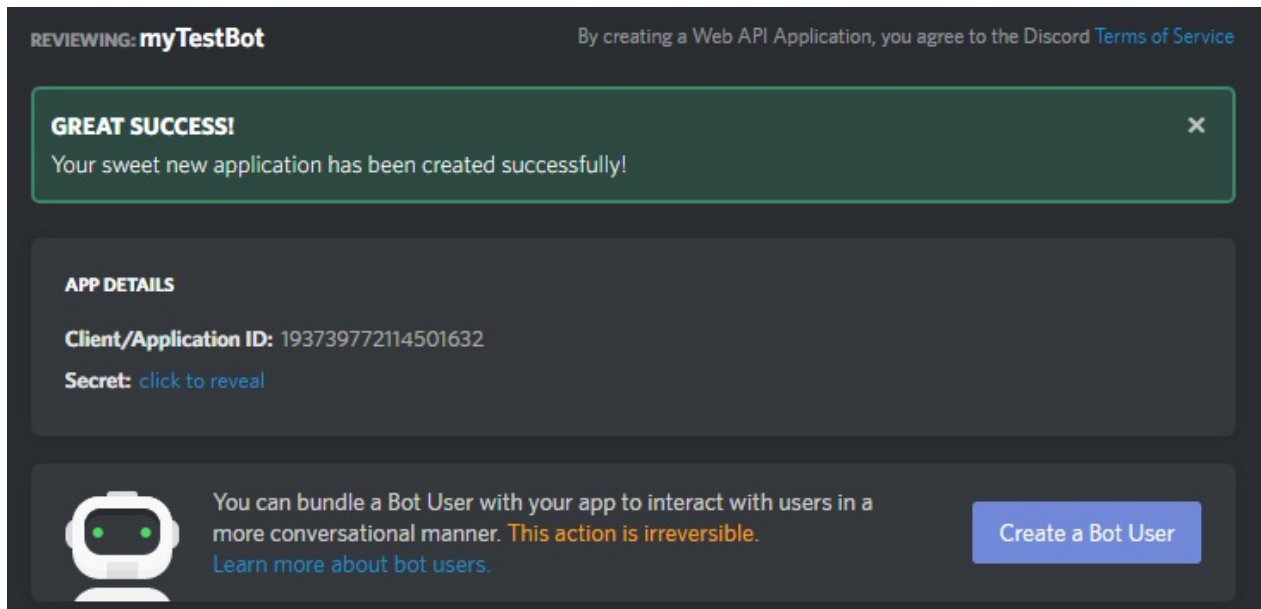


Click on (you guessed it!) **New Application**. This brings you to the following page, in which you should simply enter a name for the *application* (this will be the initial bot username). You don't really need a description or icon, but feel free to add one.

Ignore the "Redirect URI(s)", this section is not useful to you at this moment.

Create the bot account

Once you click on the **Create Application** button, you're brought to the application page, on which you see 2 new sections, once for the App ID (keep this one in mind for later) as well as a section that lets you create the **Bot User**. This is exactly what we want, so go ahead and click **Create a Bot User**, then **Yes, Do it**.



Add your bot to a server

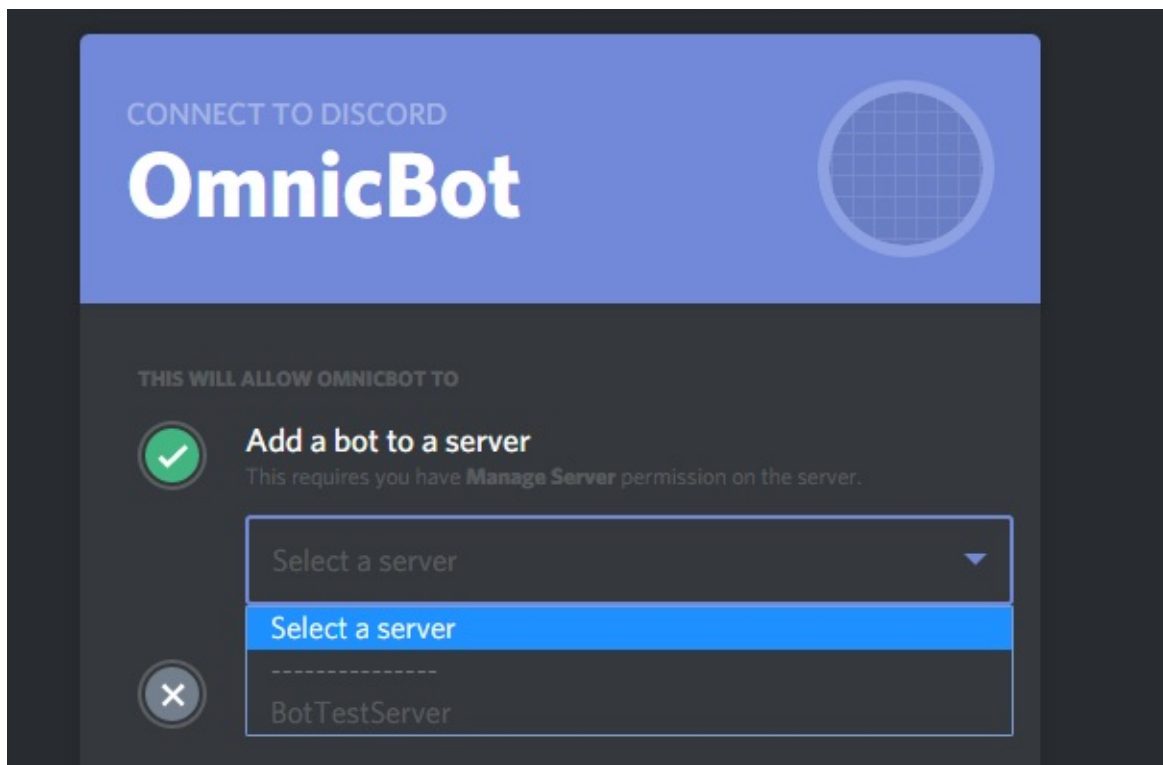
Ok so, this might be a bit early to do this but it doesn't really matter - even if you haven't written a single line of code for your bot, you can already "invite" it to a server. In order to add a bot, you need *Manage Server* permissions on that server. This is the **only** way to add a bot, it cannot use invite links or any other methods.

Unfortunately, there's no cute button here to do this automatically, the link is buried in the API reference, so I'll help you out. You need to visit the following URL, but you have to replace **Client_ID** with the **Client ID** visible in your application page.

https://discordapp.com/oauth2/authorize?client_id=Client_ID&scope=bot

You can also add a bot using specific permissions. To do that, visit the [FiniteReality Permission Link generator](#)!

When you do this, You get shown a window letting you choose the server where to add the bot, simply select the server and click **Authorize**.

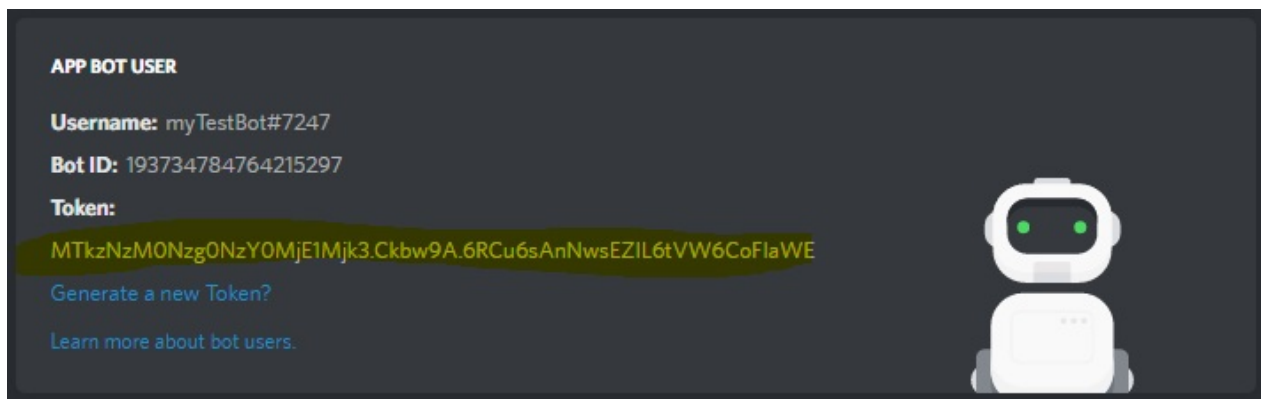


You need to be logged in to Discord on the browser with your account to see a list of servers. You can only add a bot to servers where you have **Manage Server** permissions.

Getting your Bot Token

Alright so, **big flashy warning, PAY ATTENTION**. This next part is really, really important: Your bot's **token** is meant to be **SECRET**. It is the way by which your bot authenticates with the Discord server in the same way that you login to Discord with a username and password. **Revealing your token is like putting your password on the internet**, and anyone that gets this token can use **your** bot connection to do things. Like delete all the messages on your server and ban everyone. If your token ever reaches the internet, **change it immediately**. This includes putting it on pastebin/hastebin, having it in a public github repository, displaying a screenshot of it, anything. **GOT IT? GOOD!**

With that warning out of the way, on to the next step. The Secret Token, as I just mentioned, is the way in which the bot authenticates. To see it, just click on **click to reveal** next to **Token** in the Bot section of the page. You then get this:



You need to be

No, this is not a valid token. Also make **double-sure** you're copying the **Token** and not the **Client Secret**. The latter is not used for bots.

Step 2: Getting your coding environment ready

This might go beyond saying but I'll say it anyway: You can't just start shoving bot code in notepad.exe and expect it to work. In order to use discord.js you will need a couple of things installed. At the very least:

- Get Node.js version 7.6 or higher (earlier versions are not supported). [Download for windows](#) or if you're on a linux distro, via [package manager](#).
- Get an actual code editor. Don't use notepad or notepad++, they are not sufficient. [VS Code](#) , [Sublime Text 3](#) and [Atom](#) are often recommended.

An alternative: [c9.io](#). I personally appreciate c9.io as it's a full VPS with a great editor (Ace) and installing node, discord.js and all dependencies, then running the bot, is easy. You can't host it there, but you can certainly develop there. *This is not an endorsement.*

Once you have the required software, the next step is to prepare a *space* for your code. Please don't just put your files on your desktop it's... unsanitary. If you have more than one hard drive or partition, you could create a special place for your development project. Mine, for example, is `D:\devel\` , and my bot is `d:\devel\omnicbot\` . Once you've created a folder, open your CLI (command line interface) in that folder. Linux users, you know how. Windows users, here's a trick: SHIFT+RightClick in the folder, then choose the "secret" command **Open command window here**. Magic!

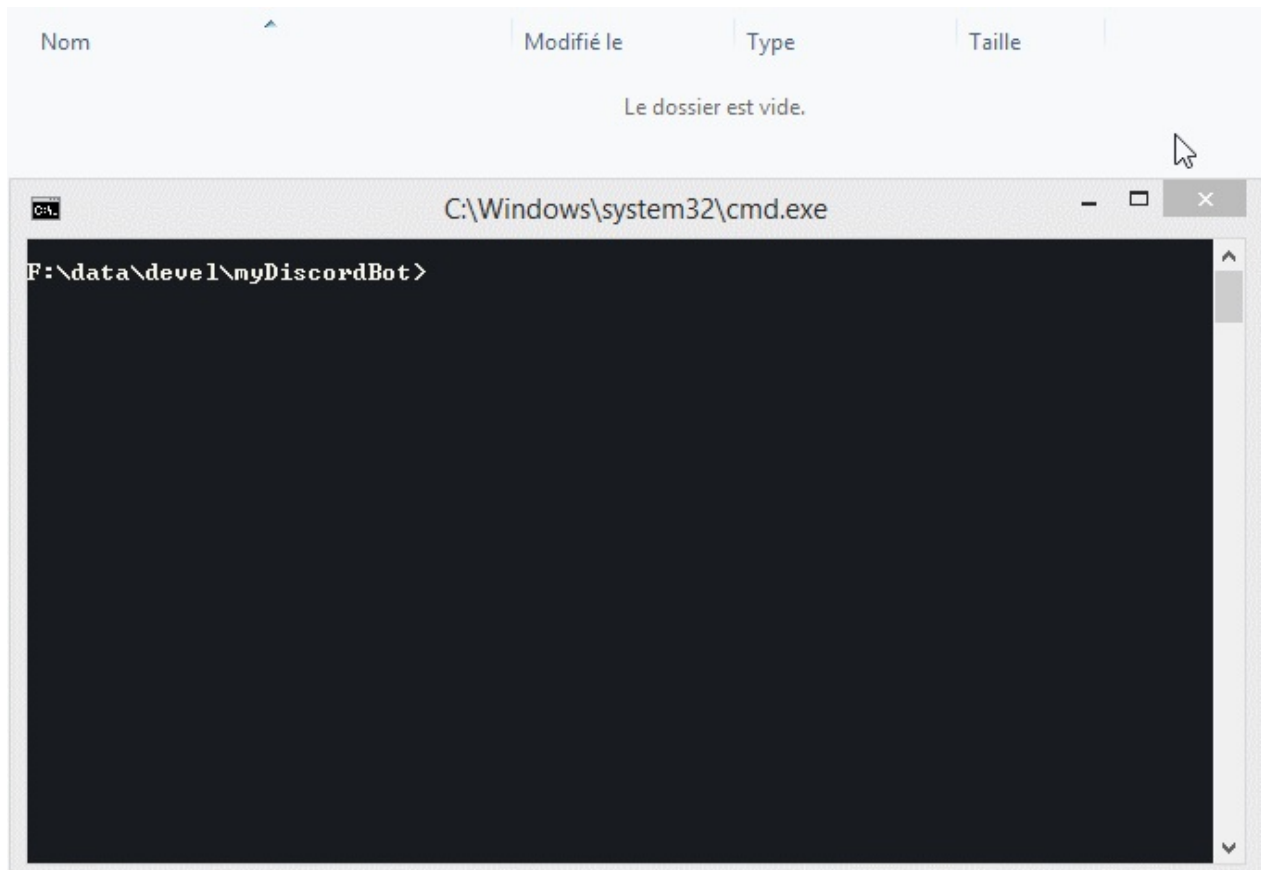
And now ready for the next step!

Installing Discord.js

So you have your CLI ready to go, in an empty folder, and you just wanna start coding. Alright, hold on one last second: let's install discord.js. But first we'll initialize this folder with NPM, which will ensure that any installed module will be here, and nowhere else. Simply run `npm init -y` and then hit Enter. A new file is created called `package.json`, [click here](#) for more info about it.

And now we install Discord.js through NPM, the Node Package Manager:

```
npm i -S discord.js
```



`i` means `install` and `-s` ensures it's saved in the `package.json` we just created!

This will take a couple of heartbeats and display a lot of things on screen. Unless you have a big fat red message saying it didn't work, or package not found, or whatever, you're good to go. If you look at your folder, you'll notice that there's a new folder created here:

`node_modules`. This contains all the installed packages for your project.

Getting your first bot running

Note: I honestly consider that if you don't understand the code you're about to see, coding a bot might not be for you. If you do not understand the following sample, please go to [CodeAcademy](#) and learn Javascript first. I beg of you: stop, drop, and roll.

Ok finally, we're ready to start coding. \o/

Let's take a look at the most basic of examples, the ping-pong bot. Here's the code in its entirety:

```
const Discord = require("discord.js");
const client = new Discord.Client();

client.on("ready", () => {
  console.log("I am ready!");
});

client.on("message", (message) => {
  if (message.content.startsWith("ping")) {
    message.channel.send("pong!");
  }
});

client.login("SuperSecretBotTokenHere");
```

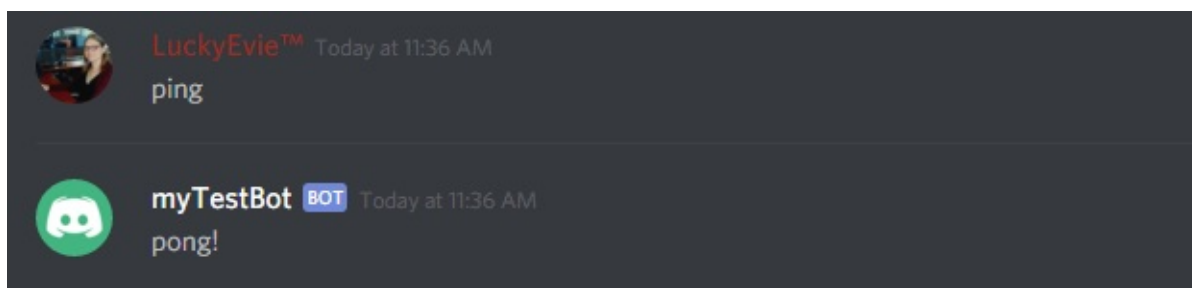
The variable `client` here is used as an example to represent the `<Client>` class. Some people call it `bot`, but you can technically call it whatever you want. I recommend sticking to `client` though!

Ok let's just... actually get this guy to work, because this is literally **a functional bot**. So let's make it run!

1. Copy that code and paste it in your editor.
2. Replace the string in the `client.login()` function with *your* secret token
3. Save the file as `mybot.js`.
4. In the CLI (which should still be in your project folder) type the following command:

```
node mybot.js
```

If all went well (hopefully it did) your bot is now connected to your server, it's in your userlist, and ready to answer all your commands... Well, at least, *one* command: `ping`. In its current state, the bot will reply "pong!" to any message that starts with, *exactly*, `ping`. Let's demonstrate!



Success! You now have a bot running! As you probably realize by now I could probably blabber on from here, showing you a bunch of stuff. But the scope of this tutorial is completed, so I'll shut up now! Ciao!

The Next Step?

Now that you have a basic, functional bot, it's time to start adding new features! Head on over to [Your First Bot](#) to continue on your journey with adding new commands and features!

Addendum: Getting help and Support

Before you start getting support from Discord servers to help you with your bot, I strongly advise taking a look at the following, very useful, resources.

- [Discord.js Documentation](#) : For the love of all that is (un)holy, **read the documentation**. Yes, it will be alien at first if you are not used to "developer documentation" but it contains a whole lot of information about each and every feature of the API. Combine this with the examples above to see the API in context.
- [An Idiot's Guide](#) is another great channel with more material. York's guides are great, and he continues to update them.
- [Evie.Codes on Youtube](#): If you prefer video to words, Evie's youtube series (which is good, though no longer maintained with new videos!) gets you started with bots.
- [An Idiot's Guide Official Server](#): The official server for An Idiot's Guide. Full of friendly helpful users!
- [Discord.js Official Server](#): The official server has a number of competent people to help you, and the development team is there too!

Getting Started - Linux Short Version

This is the **TL;DR** version for Linux. If you wish for a long version with more explanations, please see [this guide](#)

Create App and Bot Account

- Go to the [Discordapp.com Application Page](#)
- Create a **New Application**, and give it a name
- Click **Create a bot account**, then **Yes, do it**
- Visit https://discordapp.com/oauth2/authorize?client_id=APP_ID&scope=bot , replacing **APP_ID** with the **Client/Application ID** from the app page, to add the bot to your server (or ask a server admin to do it for you).
- Copy your bot's **Token** and keep it for later

Pre-requisite software

Install the following through your package manager:

- nodejs (Version 6.X and higher required, see [here](#))

Once you have this all installed, create a folder for your project and install discord.js:

```
mkdir mybot && cd mybot  npm install discord.js
```

For sound support add `npm install opusscript` (ez mode) or `npm install node-opus` (better performance but requires `python 2.7.x` and `build-essential`). BOTH these options require `ffmpeg` to run on your system, installed through `sudo apt-get install ffmpeg` .

Example Code

The following is a simple ping/pong bot. Save as a text file (e.g. `mybot.js`), replacing the string on the last line with the secret bot token you got earlier:

```
const Discord = require("discord.js");
const client = new Discord.Client();

client.on("ready", () => {
  console.log("I am ready!");
});

client.on("message", (message) => {
  if (message.content.startsWith("ping")) {
    message.channel.send("pong!");
  }
});

client.login("SuperSecretBotTokenHere");
```

Launching the bot

In your terminal, from inside the folder where `mybot.js` is located, launch it with:

```
node mybot.js
```

If no errors are shown, the bot should join the server(s) you added it to.

Resources

- [Discord.js Documentation](#) : For the love of all that is (un)holy, **read the documentation**. Yes, it will be alien at first if you are not used to "developer documentation" but it contains a whole lot of information about each and every feature of the API. Combine this with the examples above to see the API in context.
- [An Idiot's Guide](#) is another great channel with more material. York's guides are great, and he continues to update them.
- [Evie.Codes on Youtube](#): If you prefer video to words, Evie's youtube series (which is good, though no longer maintained with new videos!) gets you started with bots.
- [An Idiot's Guide Official Server](#): The official server for An Idiot's Guide. Full of friendly helpful users!
- [Discord.js Official Server](#): The official server has a number of competent people to help you, and the development team is there too!

Getting Started with Discord.js

This is the **TL;DR version for Windows**. If you wish for a long version with more explanations, please see [this guide](#)

Create App and Bot Account

- Go to the [Discordapp.com Application Page](#)
- Create a **New Application**, and give it a name
- Click **Create a bot account**, then **Yes, do it**
- Visit https://discordapp.com/oauth2/authorize?client_id=APP_ID&scope=bot , replacing **APP_ID** with the **Client/Application ID** from the app page, to add the bot to your server (or ask a server admin to do it for you).
- Copy your bot's **Secret Token** and keep it for later

Pre-requisite software

Install the following software in Windows:

- nodejs from [the downloads page](#) (Version 6.X and higher required)

If you need sound support, you'll need 2 more things:

- ffmpeg which is available [on this page](#)
- The new windows build tools:
 - Open an ADMIN command prompt, or PowerShell
 - Run the following command: `npm i -g --production windows-build-tools`
 - This installs Python 2.7 and the C++ Build Tools standalone.

Once you have this all installed, create a folder for your project and install discord.js:

```
md mybot cd mybot npm install discord.js
```

Example Code

The following is a simple ping/pong bot. Save as a text file (e.g. `mybot.js`), replacing the string on the last line with the secret bot token you got earlier:

```
const Discord = require("discord.js");
const client = new Discord.Client();

client.on("ready", () => {
  console.log("I am ready!");
});

client.on("message", (message) => {
  if (message.content.startsWith("ping")) {
    message.channel.send("pong!");
  }
});

client.login("SuperSecretBotTokenHere");
```

Launching the bot

In your command prompt, from inside the folder where `mybot.js` is located, launch it with:

```
node mybot.js
```

If no errors are shown, the bot should join the server(s) you added it to.

Resources

- [Discord.js Documentation](#) : For the love of all that is (un)holy, **read the documentation**. Yes, it will be alien at first if you are not used to "developer documentation" but it contains a whole lot of information about each and every feature of the API. Combine this with the examples above to see the API in context.
- [An Idiot's Guide](#) is another great channel with more material. York's guides are great, and he continues to update them.
- [Evie.Codes on Youtube](#): If you prefer video to words, Evie's youtube series (which is good, though no longer maintained with new videos!) gets you started with bots.
- [An Idiot's Guide Official Server](#): The official server for An Idiot's Guide. Full of friendly helpful users!
- [Discord.js Official Server](#): The official server has a number of competent people to help you, and the development team is there too!

Your First Bot

This chapter assumes you've followed the Getting Started chapter and your bot code compiles. Also, I have to repeat: if you don't understand the code you're about to see, coding a bot might not be for you. Go to [CodeAcademy](#) and learn Javascript.

In this chapter I'll guide you through the development of a simple bot with some useful commands. We'll start with the example we created in the first chapter:

```
const Discord = require("discord.js");
const client = new Discord.Client();

client.on("ready", () => {
  console.log("I am ready!");
});

client.on("message", (message) => {
  if (message.content.startsWith("ping")) {
    message.channel.send("pong!");
  }
});

client.login("SuperSecretBotTokenHere");
```

Introducing Events

Before we dive into any further coding, we need to first understand what an *Event* is.

This is an event:

```
client.on("message", (message) => {
  // This code runs when the event is triggered
});
```

This is, specifically, an event in *discord.js* but it's similar to how other APIs handle events. This event triggers *every time the bot sees a message*. This includes every channel the bot has access to as well as any direct or private message it receives. If someone sends 5 messages on a channel, this event fires 5 times.

Why is this important? Well, if you intend to use your bot on a large server, or if you want it to be on multiple servers, this becomes a large number of events triggering at every moment. I don't want to go into too much optimization talk, but for a single point: **use a**

single event function for each event.

Discord.js contains a large number of events that can trigger under certain situations. For instance, the `ready` event triggers when the bot comes online. The `guildMemberAdd` event triggers when a new user joins a server shared with the bot. For a full list of events, see [Events in the documentation](#). We will come back to some of those later in this chapter.

Adding a second command

One of the first useful things you might want to learn is how to add a second command to your bot. While there are *better* ways than what I'm about to show you, for the time being this will be enough.

From now on I will omit the code that requires and initiates the discord.js and concentrate on specific parts of the code.

```
client.on("message", (message) => {
  if (message.content.startsWith("ping")) {
    message.channel.send("pong!");
  } else

  if (message.content.startsWith("foo")) {
    message.channel.send("bar!");
  }
});
```

Save your code and reload your bot. To do so, use `CTRL+C` in the command line, and re-run `node mybot.js`. Yes, there are better ways to reload the code, as you will see later in this book.

You can test your new command by saying `foo` in a channel you share with the bot. You can also confirm that `ping` still returns `pong` !

Using a Prefix

You might have noticed that a lot of bots respond to commands that have a prefix. This might be an exclamation mark (!), a dot (.), a question mark(?), or another character. This is useful for two things.

First, if you don't use a unique prefix and have more than one bot on a server, both will respond to the same commands. On developer servers, typing `!help` leads to a flood of replies and private messages which is something to avoid.

Second, in the example above we respond when the message *starts with* the 3 characters, `foo`. In its current state, this means the following sentence will trigger the bot's response: **fool, you have not heard the last of me!**. Yes, that's an odd example, but it's still valid - say this on your bot's channel and he will respond.

To work around this, we'll be using prefix, which we will store in a variable. This way we get the prefix as well as the ability to change it for all commands in one place. Here's an example code that does this:

```
// Set the prefix
const prefix = "!";
client.on("message", (message) => {
  // Exit and stop if it's not there
  if (!message.content.startsWith(prefix)) return;

  if (message.content.startsWith(prefix + "ping")) {
    message.channel.send("pong!");
  } else
  if (message.content.startsWith(prefix + "foo")) {
    message.channel.send("bar!");
  }
});
```

The changes to the code are still simple. Let's go through them:

- `const prefix = "!";` defines the prefix as the exclamation mark. You can change it to something else, of course.
- The line `if(!msg.content.startsWith(prefix)) return;` is a small bit of optimization which reads: "If the message does not start with my prefix, stop what you're doing". This prevents the rest of the function from running, making your bot faster and more responsive.
- The commands have changed so use this prefix, where `startsWith(prefix + "ping")` would only be triggered when the message starts with `!ping`.

The second point is just as important as having a single `message` event handler. Let's say the bot receives a hundred messages every minute (not much of an exaggeration on popular bots). If the function does not break off at the beginning, you are processing these hundred messages in each of your command conditions. If, on the other hand, you break off when the prefix is not present, you are saving all these processor cycles for better things. If commands are 1% of your messages, you are saving 99% processing power...

OK I'm sorry, I'm bullshitting a little. It's not 99%, that is an exaggeration. It *is*, however, true that you save a ton on processor and RAM power.

Preventing Botception

We're pretty much done with the basic bot. There's one last thing that I want to talk about: bots answering each other. Let's pretend for a moment that you have two bots on your server and each can respond to the same prefixed command, `!help`. But when that command is called, it replies: `!help commands: Type !help followed by one of the following to see details: ping , foo .`

Now, one person types `!help` in a channel, and both bots respond. But, they will also see the **other** bot saying `!help commands: [...]`, will see that as a request for help, answer each other... in an infinite loop. To prevent that from happening, we can add a second condition inside our `message` event handler, right below the one that checks for the prefix:

```
const prefix = "!";
client.on("message", (message) => {
  // our new check:
  if (!message.content.startsWith(prefix) || message.author.bot) return;
  // [rest of the code]
});
```

That condition contains an *OR* (`||`) operator, which reads as the following:

If there is no prefix or the author of this message is a bot, stop processing. This includes this bot, itself.

And now, we have a bot that only responds to 2 commands and does not waste any power trying to figure out anything else. Is this a complete basic bot? Sure! So let's end this page here and we'll take a look at some new concept next.

The full bot code would now be:

```
const Discord = require("discord.js");
const client = new Discord.Client();

// Set the prefix
let prefix = "!";
client.on("message", (message) => {
  // Exit and stop if the prefix is not there or if user is a bot
  if (!message.content.startsWith(prefix) || message.author.bot) return;

  if (message.content.startsWith(prefix + "ping")) {
    message.channel.send("pong!");
  } else
  if (message.content.startsWith(prefix + "foo")) {
    message.channel.send("bar!");
  }
});

client.login("SuperSecretBotTokenHere");
```

Adding a `config.json` file to your bot?

Now that you have a bot up and running, we can start splitting it into some more useful parts. And the first part of this is separating some of the variables we have defined into a configuration file, `config.json`. We'll be loading this file on boot.

Why a config file?

One of the advantages of having a configuration file is that you can safely copy your bot's code into, say, hastebin.com to show people, and your token won't be in there. A second advantage is that you can upload the code to a repository like github and, as long as you ignore the config file, your bot can be shared but remain secure. We'll see that in action in a future walkthrough.

Step 1: The config file

The 2 things that we can add to the config file are:

- The bot's token
- The prefix

Simply take the following example, and create a new file in the same folder as your bot's file, calling it `config.json`:

```
{
  "token": "insert-bot-token-here",
  "prefix": "!"
}
```

Step 2: Require the config file

At the top of your bot file, you need to add a line that will load this configuration, and put it in a variable. This is what it looks like:

```
const Discord = require("discord.js");
const client = new Discord.Client();
const config = require("./config.json");
```

This means that now, `config` is your configuration object. `config.token` is your token, `config.prefix` is your prefix! Simple enough.

Step 3: Using `config` in your code

So let's use what we did earlier, and use the token from the config file, instead of putting it directly in the file. The last line of our bot looks like this:

```
client.login("SuperSecretBotTokenHere");
```

And we simply need to change it to this:

```
client.login(config.token);
```

The other thing we have, is of course the prefix. Again from before, we have this line in our message handler:


```
const prefix = "!";
client.on("message", (message) => {
  if (!message.content.startsWith(prefix) || message.author.bot) return;

  if (message.content.startsWith(prefix + "ping")) {
    message.channel.send("pong!");
  } else
  if (message.content.startsWith(prefix + "foo")) {
    message.channel.send("bar!");
  }
});
```

We're using `prefix` in a few places, so we need to change them all. Here's how it looks like after the changes:

```
client.on("message", (message) => {
  if (!message.content.startsWith(config.prefix) || message.author.bot) return;

  if (message.content.startsWith(config.prefix + "ping")) {
    message.channel.send("pong!");
  } else
  if (message.content.startsWith(config.prefix + "foo")) {
    message.channel.send("bar!");
  }
});
```

NOTE: the removal of the line that sets the prefix. We don't need it anymore!

Changing the config

You're probably wondering 'But how do I modify my prefix with a command?', right? You're in luck, that part is actually fairly easy!

This requires, first of all, the native `fs` module. At the top of your bot file, add:

NOTE `fs` is a native module, you do **not** need to install it.

```
const fs = require("fs")
```

Now, let's say you wanted a prefix-changing command. This would take the shape of:

```
if(message.content.startsWith(config.prefix + "prefix")) {
  // Gets the prefix from the command (eg. "!prefix +" it will take the "+" from it)
  let newPrefix = message.content.split(" ").slice(1, 2)[0];
  // change the configuration in memory
  config.prefix = newPrefix;

  // Now we have to save the file.
  fs.writeFile("./config.json", JSON.stringify(config), (err) => console.error);
}
```

If you want to understand what `args` is, please read [Command with arguments](#).

Awesome! Now the configuration has been changed, and we've edited `config.json` so that next time the bot restarts, the new prefix will be used! There's a *lot* more you can do with JSON files though, for more of this check out : [Storing Data in a JSON file](#). This example does an awesome *points* system just like the horrible Mee6 bot.

Extending the idea

So is there anything else you could put in that config file? Absolutely. One thing I use it for is to store my personal user ID, so that my bot can use it to recognize me and give me exclusive access to some commands.

```
{
  "token": "insert-bot-token-here",
  "prefix": "!",
  "ownerID": "your-user-ID"
}
```

Then, in a protected command, I could use the following line to prevent access to all the users that think they can use it!:

```
if(message.author.id !== config.ownerID) return;
```

What's next *now*?

Once you're done with this basic bot, there's a couple of things you should take a look at in this guide.

References and Information

- [Understanding Events and Handlers](#) gives you more details about events (things that trigger code) and handlers (the code that triggers).
- [Understanding Roles](#) is an important overview of Roles and Collections.

More code samples

For more code samples you can use in your bot, check out:

- [Welcome Message every X users](#) : it's easy to welcome users as they come in, but can you do it every 10 users? I'll show you how.
- [Message Reply Array](#) : instead of a dozen conditions with simple text replies, we'll use an array to make things easier to look at and add to!
- [Command with Arguments](#) : How do I do `!echo a message` or `!kick @xXx_phantom_sniper_xXx` ? Follow this to learn how.
- [Selfbots, the Awesomest thing in the universe](#) : If you want to use lenny make personal tags just like me, follow along, young apprentice.

Understanding Events and Handlers

We already explored one event handler in [Your Basic Bot](#), the `message` handler. Now let's take a look at some of the most important handlers that you will use, along with an example.

DO NOT NEST EVENTS

One important point: Do not nest any events (aka "put one inside another"). Ever.

Events should be at the "root" level of your code, *beside* the `message` handler and not within it.

The `ready` event and its importance

Ah, asynchronous coding. So awesome. So hard to grasp when you first encounter it. The reality of discord.js and many, many other libraries you will encounter, is that code is not executed one line at a time, one after the other.

It should have been made obvious with the user of `client.on("message")` which triggers for each message. To explain how the `ready` event is important, let's look at the following code:

```
const Discord = require("discord.js");
const client = new Discord.Client();

client.user.setGame("Online!");

client.login("SuperSecretBotTokenHere");
```

This code will not work, because `client` is not immediately available after it's been initialized. `client.user` will be undefined in this case, even if we flipped the `console.log` and `login` lines. This is because it takes a small amount of time for discord.js to load its servers, users, channels, and all that jazz. The more servers the bot is on, the longer it takes.

To ensure that `client` and all its "stuff" is ready, we can use the `ready` event. Any code that you want to run on bootup that requires access to the `client` object, will need to be in this event.

Here's a simple example of using the `ready` event handler:

```
client.on("ready", () => {
  client.user.setGame(`on ${client.guilds.size} servers`);
  console.log(`Ready to serve on ${client.guilds.size} servers, for ${client.users.size} users.`);
});
```

Detecting New Members

Another useful event is `guildMemberAdd` which triggers whenever someone joins any of the servers the bot is on. You'll see this on smaller servers: a bot welcomes every new member in the #general channel. The following code does this.

```
client.on("guildMemberAdd", (member) => {
  console.log(`New User "${member.user.username}" has joined "${member.guild.name}"` );
  ;
  member.guild.defaultChannel.send(`"${member.user.username}" has joined this server`)
  ;
});
```

The objects available for each event are important: they're only available within these contexts. Calling `message` from the `guildMemberAdd` would not work - it's not in context. `client` is always available within all its callbacks, of course.

Errors, Warn and Debug messages

Yes, bots fail sometimes. And yes, the library can too! There's a little trick we can use, however, to prevent complete crashes sometimes: Capturing the `error` event.

The following small bit of code (which can be anywhere in your file) will catch all output message from discord.js. This includes all errors, warning and debug messages.

NOTE: The debug event **WILL** output your token, so exercise caution when handing over a debug log.

```
client.on("error", (e) => console.error(e));
client.on("warn", (e) => console.warn(e));
client.on("debug", (e) => console.info(e));
```

Testing Events

So now you're wondering, how do I test those events? Do I have to join a server with an alternate account to test the `guildMemberAdd` event? Isn't that, like, super annoying?

Actually, there's an easy way to test almost any event. Without going into too many details, `client`, your Discord Client, extends something called the `EventHandler`. Any time you see `client.on("something")` it means you're handling an event called `"something"`. But `EventHandler` has another function other than `on`. It has `emit`. `Emit` is the counterpart for `on`. When you `emit` an event, it's handled by the callback for that event in `on`.

So what does it *mean*??? It means that if *you* emit an event, your code can capture it. I know I know I'm rambling without giving you an example and you're here for examples. Here's one:

```
client.emit("guildMemberAdd", message.member);
```

This emits the event that normally triggers when a new member joins a server. So it's *pretending* like this particular member has rejoined the server even if they have not. This obviously works for any event but you have to provide the proper arguments for it. Since `guildMemberAdd` requires only a member, any member will do (see [FAQ](#) to know how to get another member). I can trigger the `ready` event again by using `client.emit("ready")` (the `ready` event does not take any parameter).

What about other events? Let's see. `guildBanAdd` takes 2 parameters: `guild` and `user`, to simulate that a user was banned. So, you could `client.emit("guildBanAdd", message.guild, message.author)` to simulate banning the person sending a message. Again, getting those things (Guilds and Users) is in the [FAQ](#).

You can do all this in a "test" command, or you can do what I do: use `eval`. [Check the Eval command](#) when you're ready to go that route.

Understanding Collections

In this page we will explore Collections, and how to use them to grab data from various part of the API.

A **Collection** is a *utility class* that stores data. Collections are the Javascript Map() data structure with additional utility methods. This is used throughout discord.js rather than Arrays for anything that has an ID, for significantly improved performance and ease-of-use.

Examples of Collections include:

- `client.users` , `client.guilds` , `client.channels`
- `guild.channels` , `guild.members`
- message logs (in the callback of `fetchMessages`)
- `client.emojis`

Getting by ID

Very simply, to get anything by ID you can use `Collection.get(id)` . For instance, getting a channel can be `client.channels.get("81385020756865024")` . Getting a user is also trivial: `client.users.get("139412744439988224")`

Finding by key

If you don't have the ID but only some other property, you may use `find()` to search by property:

```
let guild = client.guilds.find("name", "Discord.js Official");
```

The `.find()` method also accepts a function. The *first* result that returns `true` within the function, will be returned. The generic idea of this is:

```
let result = <Collection>.find(item => item.property = "a value")
```

Obviously this looks a lot like the key/value find above. However, using a custom function means you can also be looking at other data, properties not a the top level, etc. Your imagination is the limit.

Want a great example? Here's getting the first role that matches one of 4 role names:

```
const acceptedRoles = ["Mod", "Moderator", "Staff", "Mod Staff"];
const getModRole = member.roles.find(role => acceptedRoles.includes(role.name));
if(!modRole) return "No role found";
```

Don't need to return the actual role? `.some()` might be what you need. It's faster than `find`, but will only return a boolean `true/false` if it finds something:

```
const hasModRole = member.roles.some(r => acceptedRoles.includes(role.name));
// hasModRole is boolean.
```

Custom filtering

Collections also have a custom way to filter their content with an anonymous function:

```
let large_guilds = client.guilds.filter(g => g.memberCount > 100);
```

`filter()` returns a new collection containing only items where the filter returned `true`, in this case guilds with more than 100 members.

Mapping Fields

One great thing you can do with a collection is to grab specific data from it with `map()`, which is useful when listing stuff. `<Collection>.map()` takes a function which returns a string. Its result is an array of all the strings returned by each item. Here's an example: let's get a complete list of all the guilds a bot is in, by name:

```
const guildNames = client.guilds.map(g => g.name).join("\n")
```

Since `.join()` is an array method, which links all entries together, we get a nice list of all guilds, with a line return between each. Neat!

We can also get a most custom string. Let's pretend the `user.tag` property doesn't exist, and we wanted to get all the `user#discrim` in our bot. Here's how we'd do it (using awesome template literals):

```
const tags = client.users.map(u=> `${u.username}#${u.discriminator}`).join(", ");
```

Combining and Chaining

In a lot of cases you can definitely chain methods together for really clean code. For instance, this is a comma-delimited list of all the small guilds in a bot:

```
const smallGuilds = client.guilds.filter(g => g.memberCount < 10).map(g => g.name).join("\n");
```

More Data!

To see **all** of the Discord.js Collection Methods, please [refer to the docs](#). Since Collection extends Map(), you will also need to refer to [this awesome mdn page](#) which describe the native methods - most notably `.forEach()`, `.has()`, etc.

Understanding Roles

Roles are a powerful feature in Discord, and admittedly have been one of the hardest parts to master in discord.js. This walkthrough aims at explaining how roles and permissions work. We'll also explore how to use roles to protect your commands.

Role hierarchy

Let's start with a basic overview of the hierarchy of roles in Discord.

... or actually not, they already explain it better than I care to: [Role Management 101](#). Read up on that, then come back here. I'll wait. (Yeah I know that's cheesy, so sue me).

Role code

Let's get down to the brass tax. You want to know how to use roles and permissions in your bot.

Get Role by Name or ID

This is the "easy" part once you actually get used to it. It's just like getting any other Collection element, but here's a reminder anyway!

```
// get role by ID
let myRole = message.guild.roles.get("264410914592129025");

// get role by name
let myRole = message.guild.roles.find("name", "Moderators");
```

Check if a member has a role

In a `message` handler, you have access to checking the `GuildMember` class of the message author:

```
// assuming role.id is an actual ID of a valid role:
if(message.member.roles.has(role.id)) {
  console.log(`Yay, the author of the message has the role!`);
} else {
  console.log(`Nope, noppers, nadda.`);
}
```

To grab members and users in different ways see the [FAQ Page](#).

Get all members that have a role

```
let roleID = "264410914592129025";
let membersWithRole = message.guild.roles.get(roleID).members;
console.log(`Got ${membersWithRole.size} members with that role.`);
```

Add a member to a role

Alright, now that you have roles, you probably want to add a member to a role. Simple enough! Discord.js provides 2 handy methods to add, and remove, a role. Let's look at them!

```
let role = message.guild.roles.find("name", "Team Mystic");

// Let's pretend you mentioned the user you want to add a role to (!addrole @user Role Name):
let member = message.mentions.members.first();

// or the person who made the command: let member = message.member;

// Add the role!
member.addRole(role).catch(console.error);

// Remove a role!
member.removeRole(role).catch(console.error);
```

Alright I feel like I have to add a *little* precision here on implementation:

- You can **not** add or remove a role that is higher than the bot's. This should be obvious.
- The bot requires `MANAGE_ROLES` permissions for this. You can check for it using the code further down this page.
- Because of global rate limits, you cannot do 2 role "actions" immediately one after the other. The first action will work, the second will not. You can go around that by using `<GuildMember>.setRoles([array, of, roles])`. This will overwrite all existing roles and only apply the ones in the array so be careful with it.

Permission code

Check specific permission of a member on a channel

To check for a single permission override on a channel:

```
// Getting all permissions for a member on a channel.  
let perms = message.channel.permissionsFor(message.member);  
  
// Checks for Manage Messages permissions.  
let can_manage_chans = message.channel.permissionsFor(message.member).hasPermission("MANAGE_MESSAGES");  
  
// View permissions as an object (useful for debugging or eval)  
message.channel.permissionsFor(message.member).serialize()
```

Get all permissions of a member on a guild

Just as easy, wooh!

```
let perms = message.member.permissions;  
  
// Check if a member has a specific permission on the guild!  
let has_kick = message.member.hasPermission("KICK_MEMBERS");
```

ezpz, right?

Now get to coding!

ADDENDUM: Permission Names

This is the list of internal permission names, used for `.hasPermission(name)` in the above examples:

```
{  
  CREATE_INSTANT_INVITE: true,  
  KICK_MEMBERS: true,  
  BAN_MEMBERS: true,  
  ADMINISTRATOR: true,  
  MANAGE_CHANNELS: true,  
  MANAGE_GUILD: true,  
  ADD_REACTIONS: true,  
  READ_MESSAGES: true,  
  SEND_MESSAGES: true,  
  SEND_TTS_MESSAGES: true,  
  MANAGE_MESSAGES: true,  
  EMBED_LINKS: true,  
  ATTACH_FILES: true,  
  READ_MESSAGE_HISTORY: true,  
  MENTION_EVERYONE: true,  
  EXTERNAL_EMOJIS: true,  
  CONNECT: true,  
  SPEAK: true,  
  MUTE_MEMBERS: true,  
  DEAFEN_MEMBERS: true,  
  MOVE_MEMBERS: true,  
  USE_VAD: true,  
  CHANGE_NICKNAME: true,  
  MANAGE_NICKNAMES: true,  
  MANAGE_ROLES_OR_PERMISSIONS: true,  
  MANAGE_WEBHOOKS: true,  
  MANAGE_EMOJIS: true  
}
```

Async/Await

Description

When an async function is called, it returns a [Promise](#). When the async function returns a value, the Promise will be resolved with the returned value. When the async function throws an exception or some value, the Promise will be rejected with the thrown value.

An async function can contain an [await](#) expression, that pauses the execution of the async function and waits for the passed Promise's resolution, and then resumes the async function's execution and returns the resolved value.

Promise

But, what are Promises? Promise Object is used for asynchronous computations/calls, but unlike functions, they don't return the value immediately, as they have three states:

- **Pending**: initial state, not fulfilled or rejected.
- **Fulfilled**: meaning that the operation completed successfully.
- **Rejected**: meaning that the operation failed.

A function returns a Promise when you call an asynchronous method, it means, as explained above, the **final** value isn't available when the function has been called, but when the object Promise resolves.

An example in the real world would be:

You get a bottle of water, you open it, turn over and drain it. Then dispose it.

The problem above is, when you get the bottle of water, you can immediately open it (sync function), but when you turn it over and drain it, you have to **await** until the bottle gets empty. This is, an **AsyncFunction**.

AoDude#8676 proposed the following example:

```
const perrier = require("PerrierBrandWater");
const bottle = new perrier.BottleOfWater();

bottle.open(); // sync operation
bottle.turnOverAndDrain() // async operation
  .then(emptyBottle => emptyBottle.dispose())
  .catch((err) => {
    console.error(err);
    runForYourLives();
  });

const runForYourLives = () => process.exit();
```

In the example above, you open the bottle (sync operation, you can do that in the code execution), then you call `turnOverAndDrain`, in which is an async operation. You don't know if the operation will be executed successfully (the water gets completely drained) or it'll fail (something happened and the water couldn't get completely drained).

When you call an asynchronous function, in **ES6**, you can use the keywords `then` and `catch`. At some point they make some logic.

You open the bottle, turn it over and drain, **then** you dispose it. But if something failed, there's an `error`, you **catch** it, display a console error and run for your lives.

Inside `then` and `catch` there are functions, inside then, we use the variable **emptyBottle**, in which is the value that the function **turnOverAndDrain** returns when it resolves. And inside catch, **err** is the error object that the function returns when it fails.

Don't get it? There's an example with Discord.JS:

Some practise, the method `client.fetchUser` returns `Promise<User>`. It means, when you call that method, it'll return a **Promise**, resolving with a **User** object (but it can also throw an error).

```
client.fetchUser(id)
  .then((User) => {
    // Do something with the User object
  })
  .catch((err) => {
    // Do something with the Error object, for example, console.error(err);
  })
```

The code above requires a UserID, but you don't have the User object since you are going to retrieve information from Discord to get the User object, that takes a while, once you get the data, Discord.JS will resolve the method, running the function inside the `then`, passing the object `User`, described in the docs.

Async/Await usage

Once we know how to use the `Object Promise` and we know how to work with it, it's now time to learn how to use ES8 Promises, with `async / await`.

```
async () => {  
  const User = await client.fetchUser(id);  
  // Do something with the User object  
}
```

WAIT WHAT? THAT'S ALL? Yes, it is. in the code above, you're defining the constant `User` as the result of the Promise, hence the keyword `await`. In this context, your code (when it executes), calls the method `client.fetchUser()`, but it'll stop there, once the promise resolves, the returned value (User Object) is assigned to the constant `User`.

Wait, we have the replacement for `then`, but what if the method fails? An advantage of ES8 Async/Await is that, you can call multiple AsyncFunctions, and catch them all once. As in the following example:

```
async () => {  
  try {  
    const User = await client.fetchUser(id);  
    const member = await guild.fetchmember(User);  
    const role = guild.roles.find("name", "Idiot Subscribers");  
    await member.addRole(role);  
    await channel.send("Success!");  
  } catch (e) {  
    console.error(e);  
  }  
}
```

In the example above, you fetch a user, once you have the `User` object, declare it as the constant `User`, then you fetch a member with the `User`, if it's found, it'll get a role (sync method, doesn't return a `Object Promise`, so you don't need the `await` keyword), add the role to the member, and send a message to the channel.

If **ONE** of the promises fail, the code stops executing the rest of the methods inside the `try` 's block and will run the block inside `catch`, with the Error object, and will send an error to the console.

The example executes a **Promise Chain** (runs promises, one after the previous), you don't want to see them with ES6 `async/await`. Seriously. It's something we call **Callback Hell** (a lot of indentation levels, code that is very hard to follow, hence much harder to work with...).

Important!

To use the `await` keyword, you **MUST** have written the `async` keyword in the function whose block contains the code. For example:

```
const EditMessage = async (id, content) => {
  const Message = await channel.fetchMessage(id); // Async
  return Message.edit(content);
}
```

```
async function EditMessage(id, content) {
  const Message = await channel.fetchMessage(id); // Async
  return Message.edit(content);
}
```

However, if you have a function inside another, for example:

```
const EditMessage = async (id, content) => {
  const Message = await channel.fetchMessage(id);
  setTimeout(() => {
    await Message.edit(content);
    Message.channel.send("Edited!");
  }, 5000);
}
```

That will throw an error:

```
    await Message.edit(content);
      ^^^^^^^
SyntaxError: Unexpected identifier
```

This example failed because the function inside `setTimeout` doesn't have the `async` keyword.

REQUIREMENTS You must have Node.js v7.x to use Async/Await, since v7.6 you are not required to use the flag `--harmony` when you run your bot. And Discord.JS v12 will **require** Node.js v8.x.

To know your **Node version**, run `node -v` OR `node --version`.

Download Node.JS v7.x [here](#), but if you want to install it with the package manager (console commands), follow the instructions [here](#).

Documentation

The following links are from MDN (Mozilla Developer Network), they provide syntax, description and examples.

- [JavaScript Reference](#)
- [AsyncFunction \(Statement\)](#)
- [AsyncFunction \(Operator\)](#)
- [await \(Operator\)](#)
- [Promise](#)

Storing Data in a JSON file

In this example we're going to read and write data to and from a JSON file. We'll keep it simple by using this JSON file for a points system. Yes, that's like Mee6 - admittedly that's a piece of shit bot, but people seem to love it so here we are.

NOTE: It should be noted that JSON is not the best storage system for this. It's prone to corruption if you do a lot of read/writes. We'll have an SQLite version coming up soon!

The basis of this system is the `fs` system, for reading and writing the file. And we'll also need the native `JSON.stringify()` and `JSON.parse()` functions to convert between the Object and JSON version of our data structure. All these words spinning your head around? Get a breath of fresh air and try again!

Basic Data Structure

So here's an example of an object that contains a list of users, along with their points and level.

```
{
  "139412744439988224" : { "points": 42, "level": 0 },
  "145978637517193216" : { "points": 3, "level": 0 },
  "90997305578106880" : { "points": 122, "level": 1},
  "173547401905176585" : { "points": 999, "level": 3}
}
```

Simple enough, right? Nothing to it. It's a [JavaScript Object](#)! But, it's just an example. don't write this just yet.

Instead, create a file in your bot folder called `points.json` with as only content 2 characters: `{}` . Save it and you now have an *empty* JSON file we can read and write.

Reading the file

Reading a JSON file is simply a question of loading the file with `fs` module:

```
const fs = require("fs");

let points = JSON.parse(fs.readFileSync("./points.json", "utf8"));
```

So at this moment, we have an object called `points` from which we can read any property. So if we pretend for a second that we loaded the above example, we could access `points["139412744439988224"].points` and that would return the number `42`. Great!

You only need to read the file *once*, when originally loading your bot file, and then as you update it you're just writing to it. This is because the `points` object continues to be updated anyway, we're only using JSON for persistence between reboots!

Writing to the file

So every time an action happens, we simply increments the proper element in the `points` array and save it. Now let's pretend we go the unoriginal route, and every time someone posts a message, we give them a point!

```
const fs = require("fs");
let points = JSON.parse(fs.readFileSync("./points.json", "utf8"));

client.on("message", message => {
  if (message.author.bot) return; // always ignore bots!

  // if the points don't exist, init to 0;
  if (!points[message.author.id]) points[message.author.id] = {
    points: 0,
    level: 0
  };
  points[message.author.id].points++;

  // And then, we save the edited file.
  fs.writeFile("./points.json", JSON.stringify(points), (err) => {
    if (err) console.error(err)
  });
});
```

And now, we can access the points of a user by grabbing from the object. However, if a user doesn't have points we want to show 0 instead, obviously. So, `let userpoints = points[message.author.id] ? points[message.author.id] : 0;` (if you don't know what the `:` and `?` shenanigans means, look up [Ternary Operator Assignment!](#))

Calculating Levels

So I put in the `levels` property in there because why else would you have points, right? Here, we have a little bit of math. Don't be scared, it's pretty simple!

```
let userPoints = points[message.author.id] ? points[message.author.id].points : 0;
let curLevel = Math.floor(0.1 * Math.sqrt(userPoints));
```

Alright, so we have a level. Let's do like all the lame bots out there and output a message when a new level is reached! yay.

```
let userLevel = points[message.author.id] ? points[message.author.id].level : 0;
if(userLevel < curLevel) {
  // Level up!
  message.reply(`You've leveled up to level **${curLevel}**! Ain't that dandy?`);
}
```

Letting a user see their level

Ok I'm certainly not going to give you the secret recipe to show a full profile like Tatsumaki. But, I can at least show you how to return a really basic command that loads and shows it.

```
if(message.content.startsWith(prefix + "level")) {
  message.reply(`You are currently level ${curLevel}, with ${userPoints} points.`);
}
```

Putting it all together

Ok so we've got a bunch of little bits of code, and your head is probably spinning wonder in what order it goes, right? Well let's fix that now. On top of which we'll simplify a few things. Follow along, now!

```
const Discord = require("discord.js");
const fs = require("fs");
const client = new Discord.Client();

let points = JSON.parse(fs.readFileSync("./points.json", "utf8"));
const prefix = "+";

client.on("message", message => {
  if (!message.content.startsWith(prefix)) return;
  if (message.author.bot) return;

  if (!points[message.author.id]) points[message.author.id] = {
    points: 0,
    level: 0
  };
  let userData = points[message.author.id];
  userData.points++;

  let curLevel = Math.floor(0.1 * Math.sqrt(userData.points));
  if (curLevel > userData.level) {
    // Level up!
    userData.level = curLevel;
    message.reply(`You've leveled up to level **${curLevel}**! Ain't that dandy?`);
  }

  if (message.content.startsWith(prefix + "level")) {
    message.reply(`You are currently level ${userData.level}, with ${userData.points} points.`);
  }
  fs.writeFile("./points.json", JSON.stringify(points), (err) => {
    if (err) console.error(err)
  });
});

client.login("SuperSecretBotTokenHere");
```

Now take this, and make it **better than Mee6!** Go ahead, I challenge you ;)

Storing Data in an SQLite file

As mentioned in the [Storing Data in a JSON file](#) guide, JSON files could get corrupted due to *race conditions*. However SQLite doesn't suffer from that and is a better method of storing data between boot ups than JSON.

That is the focus of this guide: we'll be recreating the points system with SQLite instead of JSON. The core of this system is using the `sqlite` (not `sqlite3`) package that you can get from npmjs.com.

NOTE: The reason I stated `sqlite` and not `sqlite3` is because `sqlite` is a promise wrapper for `sqlite3` and with Discord.js being promise based, it's obviously the best choice.

Setting the table

Like in the JSON guide you had a data structure, SQLite is no different, but it's called a table. Now what do we want in our table? I'll tell ya!

For this example points system we want the user's ID, points and level, I'm not going to go deep in to the jargon surrounding SQL and SQLite, but the tables are made from rows and columns of data. Got it? Good, moving on!

Let's take the core elements from the example bot on [getting started](#).

```
const Discord = require("discord.js");
const client = new Discord.Client();

client.on("ready", () => {
  console.log("I am ready!");
});

client.on("message", (message) => {
  if (message.content.startsWith("ping")) {
    message.channel.send("pong!");
  }
});

client.login("SuperSecretBotTokenHere");
```

Now we've got that we should `require` `sqlite` and make use of it, put the following under `const client`

```
const sql = require("sqlite");
sql.open("./score.sqlite");
```

NOTE: Don't worry about the file, we'll be doing a special conditional shortly that'll do some fancy magic!

Alright now that's required correctly we want to prevent people trying to DM the bot to increase their points. Add the following code below the ignore bots line.

```
if (message.channel.type === "dm") return; // Ignore DM channels.
```

Your code should look like this now;

```
const Discord = require("discord.js");
const client = new Discord.Client();
const sql = require("sqlite");
sql.open("./score.sqlite");

client.on("ready", () => {
  console.log("Ready!");
});

client.on("message", message => {
  if (message.author.bot) return; // Ignore bots.
  if (message.channel.type === "dm") return; // Ignore DM channels.
  if (message.content.startsWith("ping")) {
    message.channel.send("pong!");
  }
});

client.login("SuperSecretBotTokenHere");
```

Alright, time to get down to business.

We need to start the sqlite chain, we don't have to worry about opening the database, as it's opened at the top of our file, so it's loaded when we need it. With sqlite being promise based, we need to start off with a `get` query then follow it up with a `catch`

```
sql.get(`SELECT * FROM scores WHERE userId = "${message.author.id}"`).then(row => {

}).catch(() => {

});
```


Right, now we've got that out of the way, we need to add our logic to it, but let's work on the catch first so, remember when I mentioned about some magic? Well, when the code attempted to open the database it in fact created the file for us since there was no file. However the file doesn't have a table, that's where the catch comes in, add the following code to the catch.

```
console.error; // Gotta log those errors
sql.run("CREATE TABLE IF NOT EXISTS scores (userId TEXT, points INTEGER, level INTEGER)").then(() => {
  sql.run("INSERT INTO scores (userId, points, level) VALUES (?, ?, ?)", [message.author.id, 1, 0]);
});
```

Let's break that code down shall we?

Obviously the first thing we want to do on errors is to log them, but we also want to do some magic... what the code does is creates the *scores* table in the database file IF it does not exist, then it inserts the same data you would get if it found the file, otherwise it just opens and inserts the data just like normal.

On to the next bit of logic, inside the `then` you should notice we defined `row`, now we'll put it into good use with the following code.

```
if (!row) { // Can't find the row.
  sql.run("INSERT INTO scores (userId, points, level) VALUES (?, ?, ?)", [message.author.id, 1, 0]);
} else { // Can find the row.
  sql.run(`UPDATE scores SET points = ${row.points + 1} WHERE userId = ${message.author.id}`);
}
```

Now, that will either insert (if no row is found), or update the authors points if it found a row... perfect!

Your code should now look like this.

```
const Discord = require("discord.js");
const client = new Discord.Client();
const sql = require("sqlite");
sql.open("./score.sqlite");

client.on("message", message => {
  if (message.author.bot) return;
  if (message.channel.type !== "text") return;
  if (message.content.startsWith("ping")) {
    message.channel.send("pong!");
  }
  sql.get(`SELECT * FROM scores WHERE userId = "${message.author.id}"`).then(row => {
    if (!row) {
      sql.run("INSERT INTO scores (userId, points, level) VALUES (?, ?, ?)", [message.author.id, 1, 0]);
    } else {
      sql.run(`UPDATE scores SET points = ${row.points + 1} WHERE userId = ${message.author.id}`);
    }
  }).catch(() => {
    console.error;
    sql.run("CREATE TABLE IF NOT EXISTS scores (userId TEXT, points INTEGER, level INTEGER)").then(() => {
      sql.run("INSERT INTO scores (userId, points, level) VALUES (?, ?, ?)", [message.author.id, 1, 0]);
    });
  });
});

client.login("SuperSecretBotTokenHere");
```

DING Level up!

Now, what's the point of having all of these points? To level up of course!

But first we need to calculate the level, so I'm going to take the code from the original article and rework it slightly.

This is the original code.

```
let userLevel = points[message.author.id] ? points[message.author.id].level : 0;
if(userLevel < curLevel) {
  // Level up!
  message.reply(`You've leveled up to level **${curLevel}**! Ain't that dandy?`);
}
```

Now, we need to change a couple of things, such as using sql row names, but through the magic of tutorials, here's what the code should look like after our tweaks

```
let curLevel = Math.floor(0.1 * Math.sqrt(row.points + 1));
if (curLevel > row.level) {
  row.level = curLevel;
  sql.run(`UPDATE scores SET points = ${row.points + 1}, level = ${row.level} WHERE userId = ${message.author.id}`);
  message.reply(`You've leveled up to level **${curLevel}**! Ain't that dandy?`);
}
```

Place that code inside our `row` conditional, above the `update` query.

Let a user view their level & points.

Now we've got the core of this code done, we need to add a few commands, so as normal we'll add a prefix and ignore messages without a prefix.

Place this above your message handler

```
const prefix = "+";
```

And this just below the `sql` code block

```
if (!message.content.startsWith(prefix)) return; // Ignore messages that don't start with the prefix

if (message.content.startsWith(prefix + "level")) {

} else

if (message.content.startsWith(prefix + "points")) {

}
```

Now for the commands. We want to view the row `level` and `points`, so we need to do another sql query like we did above, then respond to the user with their level or points.

All you'll need to do is swap `level` for `points` and the response message and you're set!

```
sql.get(`SELECT * FROM scores WHERE userId = "${message.author.id}"`).then(row => {
  if (!row) return message.reply("Your current level is 0");
  message.reply(`Your current level is ${row.level}`);
});
```

Conclusion

After this guide, your code should look like this;

```
const Discord = require("discord.js");
const client = new Discord.Client();
const sql = require("sqlite");
sql.open("./score.sqlite");

const prefix = "+";
client.on("message", message => {
  if (message.author.bot) return;
  if (message.channel.type !== "text") return;

  if (message.content.startsWith(prefix + "ping")) {
    message.channel.send("pong!");
  }

  sql.get(`SELECT * FROM scores WHERE userId ="${message.author.id}"`).then(row => {
    if (!row) {
      sql.run("INSERT INTO scores (userId, points, level) VALUES (?, ?, ?)", [message.author.id, 1, 0]);
    } else {
      let curLevel = Math.floor(0.1 * Math.sqrt(row.points + 1));
      if (curLevel > row.level) {
        row.level = curLevel;
        sql.run(`UPDATE scores SET points = ${row.points + 1}, level = ${row.level} WHERE userId = ${message.author.id}`);
        message.reply(`You've leveled up to level **${curLevel}**! Ain't that dandy?`)
      }
      sql.run(`UPDATE scores SET points = ${row.points + 1} WHERE userId = ${message.author.id}`);
    }
  }).catch(() => {
    console.error;
    sql.run("CREATE TABLE IF NOT EXISTS scores (userId TEXT, points INTEGER, level INTEGER)").then(() => {
      sql.run("INSERT INTO scores (userId, points, level) VALUES (?, ?, ?)", [message.author.id, 1, 0]);
    });
  });

  if (!message.content.startsWith(prefix)) return;

  if (message.content.startsWith(prefix + "level")) {
    sql.get(`SELECT * FROM scores WHERE userId ="${message.author.id}"`).then(row => {
      if (!row) return message.reply("Your current level is 0");
      message.reply(`Your current level is ${row.level}`);
    });
  } else
```

```
if (message.content.startsWith(prefix + "points")) {  
    sql.get(`SELECT * FROM scores WHERE userId ="${message.author.id}"`).then(row => {  
        if (!row) return message.reply("sadly you do not have any points yet!");  
        message.reply(`you currently have ${row.points} points, good going!`);  
    });  
}  
});  
  
client.login("SuperSecretBotTokenHere");
```

Now, when ever anyone in your guild talks, the code will either create a new table row for them, or update their table role by taking the current amount of points and simply adding 1 to it. My challenge for you dear reader, is to make this multi-guild friendly.

PersistentCollection Guide

Persistent Collections are a data structure that can be used to store data in memory that is also saved in a database behind the scenes. The data is synchronized to the database automatically, seamlessly, and asynchronously so it should not adversely affect your performance compared to regular Discord.js Collections

So why use this when you have JSON and SQLite? Because with a PersistentCollection you don't have to write any code to read or write from files, no SQL queries, no complications. You just insert data inside a Collection, and it's saved in the database. That's it!

Installing

PersistentCollections can be installed from `npm` :

```
npm install --save djs-collection-persistent
```

They can be imported into your bot code using the following:

```
const PersistentCollection = require('djs-collection-persistent');
```

Creating a "Table"

Most of us understand the concept of a "table", basically a structure that holds multiple rows and columns. This applies to concepts from Databases to Excel Spreadsheets so to simplify this how-to I'll go ahead and use those terms.

To create a new table, then, you need to "initialize" a new PersistentCollection:

```
const myTable = new PersistentCollection({name: "myTable"});
```

When this code is run, one of 2 things can happen:

- **If there is no table of that name**, this table is created in the database, and it's empty.
- **If the table exists**, that table and *all its contents* is loaded into the Collection itself.

So what does this mean? It means that "initializing the database" and "loading all its data" is taken care of, for you. You don't need to do anything else for this to happen.

While it's possible to put the initialization line in multiple files, it's recommended not to do so as each load duplicates the whole data set into memory. That is to say, the more files load a table, the more memory you're taking. To prevent that, simply attach the PersistentCollection to your client variable: `client.myTable = myTable;` for instance.

Inserting Data

Inserting data is super simple once you've initialized the table. Please note however that inserting data has its limits:

- **Keys** must be either a string or a number. Any other value is rejected.
- **Values** must be simple data types on which `JSON.stringify()` can be applied. Arrays, Objects, Strings, Numbers, those are fine. However, complex types like `Map()` and `Set()` (and, of course, other collections) cannot be inserted into the database.

Collections are a simple key/value pair storage, so there is no concept of a "column" here. Essentially, when inserting data, you're "setting" the value of a *key* to a single *value*. Here is the simplest example possible:

```
myTable.set("foo", "bar");
```

However, you can of course have something approximating rows by inserting either an array, or an object. For instance, here's how I do my per-server configuration for a bot:

```
const guildSettings = new PersistentCollection({name: 'guildSettings'});

const defaultSettings = {
  prefix: "!",
  modLogChannel: "mod-log",
  modRole: "Moderator",
  adminRole: "Administrator",
  welcomeMessage: "Say hello to {{user}}, everyone! We all need a warm welcome sometimes :D"
}

client.on("guildCreate", guild => {
  guildSettings.set(guild.id, defaultSettings);
});
```

Getting Data

Grabbing data from the PersistentCollection is just as simple as writing to it. You only need to know the key!

```
myTable.get("foo"); // outputs "bar"
```

If your value is more complex, such as with guildSettings above, properties of the values are accessible as they would be normally. For instance:

```
const thisConf = guildSettings.get(message.guild.id);

const prefix = thisConf.prefix;

if(!message.content.startsWith(prefix)) return;
```

You could also access arrays in the same way. Here's an example, which also shows that this whole thing is synchronous:

```
myTable.set("myArray", ["blah", "foo", "thingamajig", "goobbledigook"]);

myTable.get("myArray")[2]; // outputs "thingamajig"
```

Editing Data

There's no "edit" feature in Maps. Really what you need to do is to load the data, change it in memory, and re-write it. So, here's an example of loading a guild's settings, changing a value, and saving that change:

```
const thisConf = guildSettings.get(message.guild.id);

thisConf.prefix = "+";

guildSettings.set(message.guild.id, thisConf);
```

Some Use Cases

So, want to know what you can do with PersistentCollection? Here's a couple of ideas for ya!

- **Per-Server Configuration:** As shown higher up. [A full example is available as a gist](#)
- **Tags:** Either per-server or global, they're simple strings so why not? I use them in my selfbot, [initializing in app.js](#) and [controlling in a tags command](#).
- **Points system:** Both "JSON" and "SQLite" tutorials use this as an example, it could

easily be adapted with PersistentCollection. You could load points with
`points.get(message.author)` for example.

These are Collections after all

So, since PersistentCollection extends Collection, it means that *all* the Collection features you know and love. Want to grab all the guildSettings that have the default prefix?

`guildSettings.filter(c=>c.prefix === "!")` . Want to get all the tag names from a tag collection? `myTags.map(t=>t.name).join(", ")` ! The possibilities are endless ^_^

A Basic Command Handler Example

A *Command Handler* is essentially a way to separate your commands into different files, instead of having a bunch of `if/else` conditions inside your code (or a `switch/case` if you're being fancy).

In this case, the code shows you how to separate each command into its own file. This means that each command can be *edited* separately, and also *reloaded* without the need to restart your bot. Yes, really!

Want a better, updated version of this code? We're now maintaining this command handler at the community level. [Guide Bot is on Github](#) and not only can you use the code, you can also contribute if you feel proficient enough!

App.js Changes

Without going into all the details of the changes made, here is the modified app.js file:

```

const Discord = require("discord.js");
const client = new Discord.Client();
const fs = require("fs");

const config = require("./config.json");

// This loop reads the /events/ folder and attaches each event file to the appropriate
// event.
fs.readdir("./events/", (err, files) => {
  if (err) return console.error(err);
  files.forEach(file => {
    let eventFunction = require(`./events/${file}`);
    let eventName = file.split(".")[0];
    // super-secret recipe to call events with all their proper arguments *after* the
    // `client` var.
    client.on(eventName, (...args) => eventFunction.run(client, ...args));
  });
});

client.on("message", message => {
  if (message.author.bot) return;
  if (message.content.indexOf(config.prefix) !== 0) return;

  // This is the best way to define args. Trust me.
  const args = message.content.slice(config.prefix.length).trim().split(/ +/g);
  const command = args.shift().toLowerCase();

  // The list of if/else is replaced with those simple 2 lines:
  try {
    let commandFile = require(`./commands/${command}.js`);
    commandFile.run(client, message, args);
  } catch (err) {
    console.error(err);
  }
});

client.login(config.token);

```

Fair Warning: The `require()` command here relies on *user input* to call a file, which can be dangerous. Your homework is to find a way to make sure *only* commands from the `./commands/` folder are called, and anything else such as `../../../../../../etc/passwd` can't be loaded in memory because someone knows linux!

Example commands

This would be the content of the `./commands/ping.js` file, which is called with `!ping` (assuming `!` as a prefix)

```
exports.run = (client, message, args) => {  
  message.channel.send("pong!").catch(console.error);  
}
```

Another example would be the more complex `./commands/kick.js` command, called using `!kick @user`

```
exports.run = (client, message, [mention, ...reason]) => {  
  const modRole = message.guild.roles.find("name", "Mods");  
  if (!modRole)  
    return console.log("The Mods role does not exist");  
  
  if (!message.member.roles.has(modRole.id))  
    return message.reply("You can't use this command.");  
  
  if (message.mentions.users.size === 0) {  
    return message.reply("Please mention a user to kick");  
  }  
  
  if (!message.guild.me.hasPermission("KICK_MEMBERS"))  
    return message.reply("");  
  
  const kickMember = message.mentions.members.first();  
  
  kickMember.kick(reason.join(" ")).then(member => {  
    message.reply(`${member.user.username} was successfully kicked.`);  
  });  
}
```

Notice the structure on the first line. `exports.run` is the "function name" that is exported, with 3 arguments: `client` (the client), `message` (the message variable from the handler) and `args`. Here, `args` is replaced by fancy destructuring that captures the `reason` (the rest of the message after the mention) in an array. See [Commands with Arguments](#) for details.

Example Events

Events are handled almost exactly in the same way, except that the number of arguments depends on which event it is. For example, the `ready` event:

```
exports.run = (client) => {  
  console.log(`Ready to server in ${client.channels.size} channels on ${client.guilds.size} servers, for a total of ${client.users.size} users.`);  
}
```

Note that the `ready` event normally doesn't have any arguments, it's just `()`. But because we're in separate modules, it's necessary to "pass" the `client` variable to it or it would not be accessible.

Here's another example with the `guildMemberAdd` event:

NOTE: `defaultChannel` is no longer a thing, use this work around instead.

```
exports.run = (client, member) => {
  const defaultChannel = member.guild.channels.find(c=> c.permissionsFor(guild.me).has(
    "SEND_MESSAGES"));
  defaultChannel.send(`Welcome ${member.user} to this server.`).catch(console.error);
}
```

Now we have `client` and also `member` which is the argument provided by the `guildMemberAdd` event.

BONUS: The "reload" command

Because of the way `require()` works in node, if you modify any of the command files in `./commands`, the changes are not reflected immediately when you call that command again - because `require()` *caches* the file in memory instead of reading it every time. While this is great for efficiency, it means we need to clear that cached version if we change commands.

The *Reload* command does just that, simply deletes the cache so the next time that specific command is run, it'll refresh its code from the file.

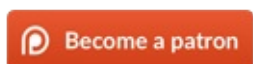
```
exports.run = (client, message, args) => {
  if(!args || args.size < 1) return message.reply("Must provide a command name to reload.");
  // the path is relative to the *current folder*, so just ./filename.js
  delete require.cache[require.resolve(`./${args[0]}.js`)];
  message.reply(`The command ${args[0]} has been reloaded`);
};
```

Coding a Music Bot

Everyone including their grandparents want to create a music bot, I myself have created a music bot and believe me it's not as easy as you would think.

However, if you insist on creating a music bot you can either check out the example bots below, or you could support me on Patreon for \$5+, and you'll get a step by step tutorial on how to make a multi-guild compatible music bot and access to the completed source code to a custom music bot (When the series concludes).

As well as the music bot tutorial you also get a `patron` role on my guild, a private patron only channel to discuss with other patrons, priority d.js bot help, behind the scenes pictures, videos and vlogs, and early access to regular content and future patron tutorial series.



If you do not feel like supporting me, there are these amazing examples; [OhGodMusicBot](#) by AoDude [Moosik](#) by Gawdl3y [Commando's Music](#) by Crawl

You think you're so clever do you?

I've had this request since I started my Idiot's Guide, in fact it was one of the very first requests I had, but I had a feeling it would be a disappointing short episode, maybe a 5 minute long episode. But for a written guide it'd be perfect!

So to get started, let's grab the example from [getting started](#) and shove it in a file.

```
const Discord = require("discord.js");
const client = new Discord.Client();

client.on("ready", () => {
  console.log("I am ready!");
});

client.on("message", (message) => {
  if (message.content.startsWith("ping")) {
    message.channel.send("pong!");
  }
});

client.login("superSecretBotTokenHere");
```

Once you've got that, we should go check out `cleverbot-node` on npmjs.com and grab their example code

```
var Cleverbot = require("cleverbot-node");
cleverbot = new Cleverbot;
cleverbot.configure({botapi: "IAMKEY"});
cleverbot.write(cleverMessage, function (response) {
  console.log(response.output);
});
```

NOTE: As you can see in the above example taken from the npm page, you now need an API key, which you can get [here](#), they do offer a free tier, but that's 5,000 api calls per month (at the time of writing), and please note it says it's a free **trial**.

Alright, we've got both parts we need, now before we continue we should get the module installed, just run `npm i cleverbot-node` with the `--save` flag if you have a `package.json` file (and you should!).

Installed? Good! Now, let's get to the final step... the code.

We have both our example codes, now we need to combine them for a working bot.

NOTE: A lot of the naive developers would just shove the cleverbot example straight in their message event and wonder why it wasn't working. It would create a new instance of Cleverbot and would eventually cause a memory leak.

Right, we need to take the first two lines of the cleverbot example...

```
var Cleverbot = require("cleverbot-node");
cleverbot = new Cleverbot;
```

...and put them with our discord definitions.

```
const Discord = require("discord.js");
const Cleverbot = require("cleverbot-node");
const client = new Discord.Client();
const clbot = new Cleverbot;
clbot.configure({botapi: "IAMKEY"});
```

I renamed `cleverbot` to `clbot` to reduce any possible confusion between the variable names as JavaScript is case sensitive.

Then we take the rest of the code and place that inside our message event handler, but for this example I only want the bot to talk to me in DM's, so we'll check the channel `type` with the following code, you can make it respond on mentions or even in channels (I would honestly advise against that.)

```
if (message.channel.type === "dm") {
  // Cleverbot code goes here.
}
```

Your code should look something like this...

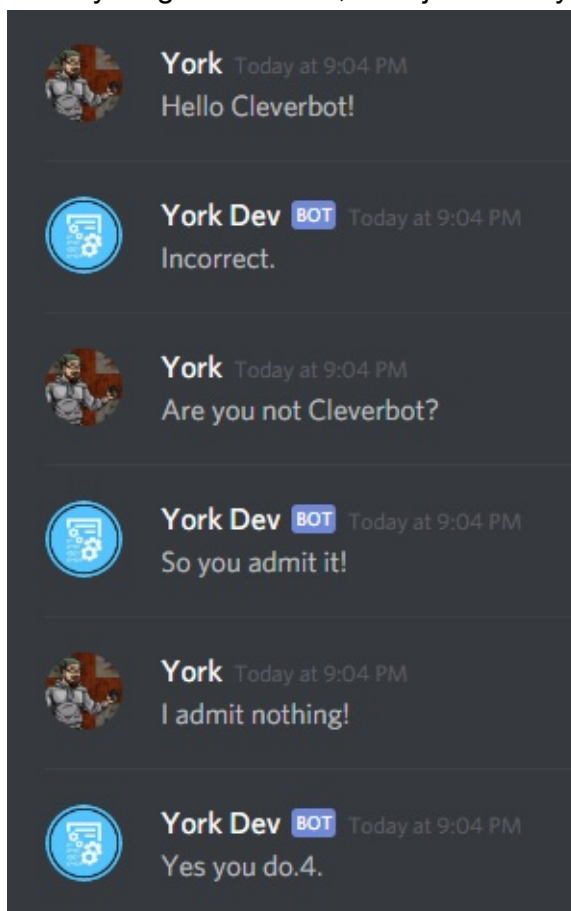

```
const Discord = require("discord.js");
const Cleverbot = require("cleverbot-node");
const client = new Discord.Client();
const clbot = new Cleverbot;

client.on("message", message => {
  if (message.channel.type === "dm") {
    clbot.write(message.content, (response) => {
      message.channel.startTyping();
      setTimeout(() => {
        message.channel.send(response.output).catch(console.error);
        message.channel.stopTyping();
      }, Math.random() * (1 - 3) + 1 * 1000);
    });
  }
});

client.on("ready", () => {
  console.log("I am ready!");
});

client.login("superSecretBotTokenHere");
```

If everything is as above, then just send your bot a DM and watch the magic unfold!



Creating Discord Webhooks

NOTE At the time of writing this guide (07/02/2017) there is a bug with how webhooks are created via code, you must supply a webhook name and avatar. **However it does not work as intended.** You must *edit* the webhook with the same details for the avatar to be applied.

This has been a rather demanded topic recently, everyone wants to know how to use the webhooks, so here I am with this guide to explain the basic coverage of the webhooks.

As per usual let's grab the example source code.

```
const Discord = require("discord.js");
const client = new Discord.Client();

client.on("ready", () => {
  console.log("I am ready!");
});

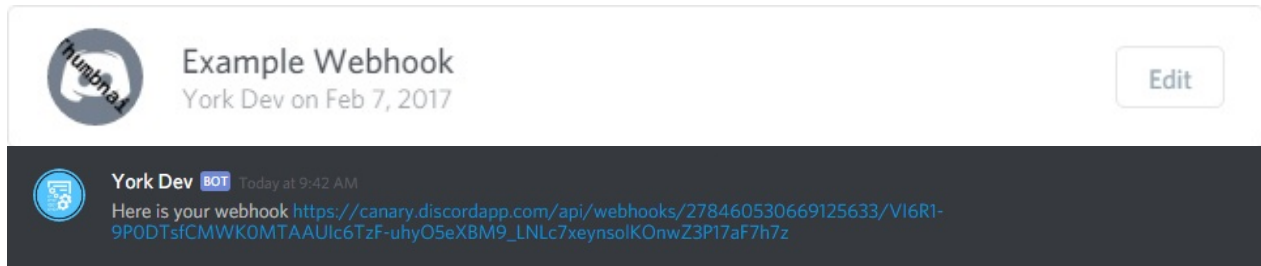
client.on("message", (message) => {
  if (message.content.startsWith("ping")) {
    message.channel.send("pong!");
  }
});

client.login("SuperSecretBotTokenHere");
```

Right, we'll start off slow, we need to create a webhook first, if we look at the [documentation](#) it comes with an example, that is basically all we need to create a webhook, but we'll add some polish to it and throw it into a basic command.

```
// This will create the webhook with the name "Example Webhook" and an example avatar.
message.channel.createWebhook("Example Webhook", "https://i.imgur.com/p2qNFag.png")
// This will actually set the webhooks avatar, as mentioned at the start of the guide.
.then(webhook => webhook.edit("Example Webhook", "https://i.imgur.com/p2qNFag.png"))
// This will get the bot to DM you the webhook, if you use this in a selfbot,
// change it to a console.log as you cannot DM yourself
.then(wb => message.author.send(`Here is your webhook https://canary.discordapp.com/ap
i/webhooks/${wb.id}/${wb.token}`)).catch(console.error))
```

This is what it should look like if you test the code.



NOTE: This webhook link has long since been deleted.

Now, that's all well and good, we can create the webhooks and get our bot to DM us, but the values are *hardcoded*, which means if we run that command, we'd get webhooks by the same name / avatar all the time, let's fix that shall we? we'll be looking at the [command arguments](#) page.

You should have a message handler that looks something like this.

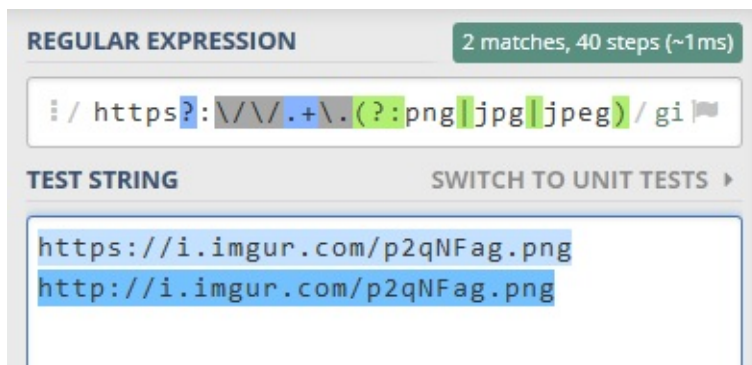
```
let prefix = "~";
client.on("message", message => {
  let args = message.content.split(" ").slice(1);
  if (message.content.startsWith(prefix + "createHook")) {
    message.channel.createWebhook("Example Webhook", "https://i.imgur.com/p2qNFag.png")
      .then(webhook => webhook.edit("Example Webhook", "https://i.imgur.com/p2qNFag.png"))
      .then(wb => message.author.send(`Here is your webhook https://canary.discordapp.com/api/webhooks/${wb.id}/${wb.token}`))
      .catch(console.error)
      .catch(console.error);
  }
});
```

So far so good, but we're going to run into a problem, what if you want to give your webhook a name that contains spaces? Right now you'd end up with the avatar url in the name, so we're going to have to use some *regex*, [Regular Expressions](#) is very powerful, and very daunting to start out with, but don't worry, the regex I'm going to supply for this example works, just drop it in your code and you're good.

Here's the regex on it's own

```
/https?:\/\/\.[^\s]*(?:png|jpg|jpeg)/gi
```

Using the above regex with `match`, `replace` and `test` will allow you to isolate the image url in the string and leave the remaining string to be used as the webhook's name, there's an amazing online tool called [regex101.com](#), with that tool I was able to create the above regex, here's an image of it in action.

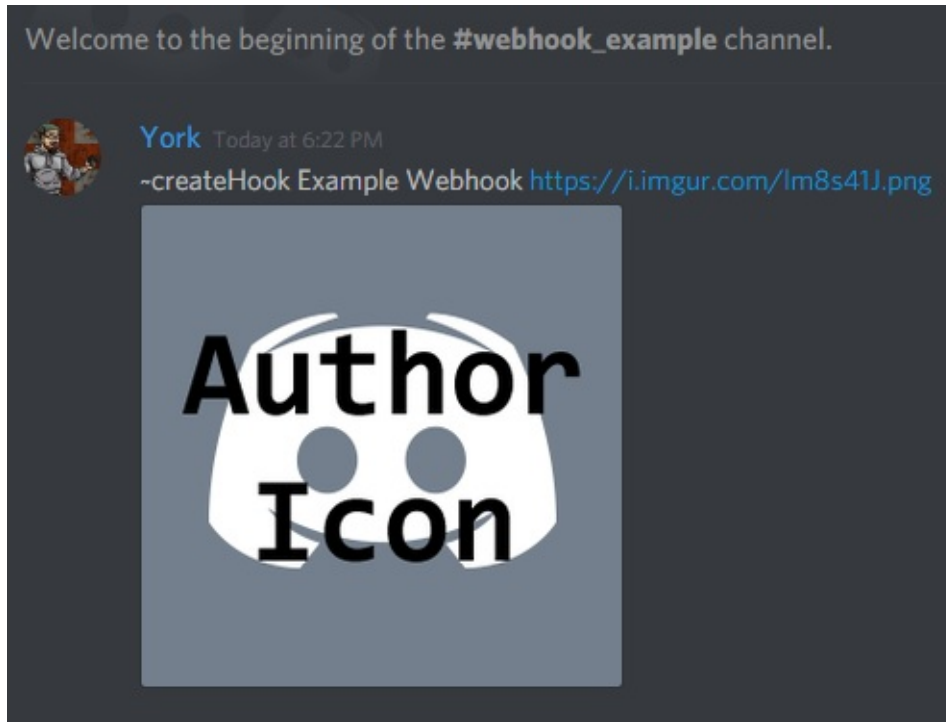


I'm not going to go into much detail, but the fact that both of the test strings are highlighted, and it's saying there's 2 matches is all we need to know, it works with links starting with `http` and `https`, and it looks for valid extensions, which are `jpg`, `jpeg` and `png`.

let's do the rest of the command shall we? we've got our regex, and we know we need to use `match`, `replace` and `test`, so we should `test` for a link first using our regex, if it returns false we need to notify the user, if it returns true we need to `match` and `replace` for the rest, the code can look like this.

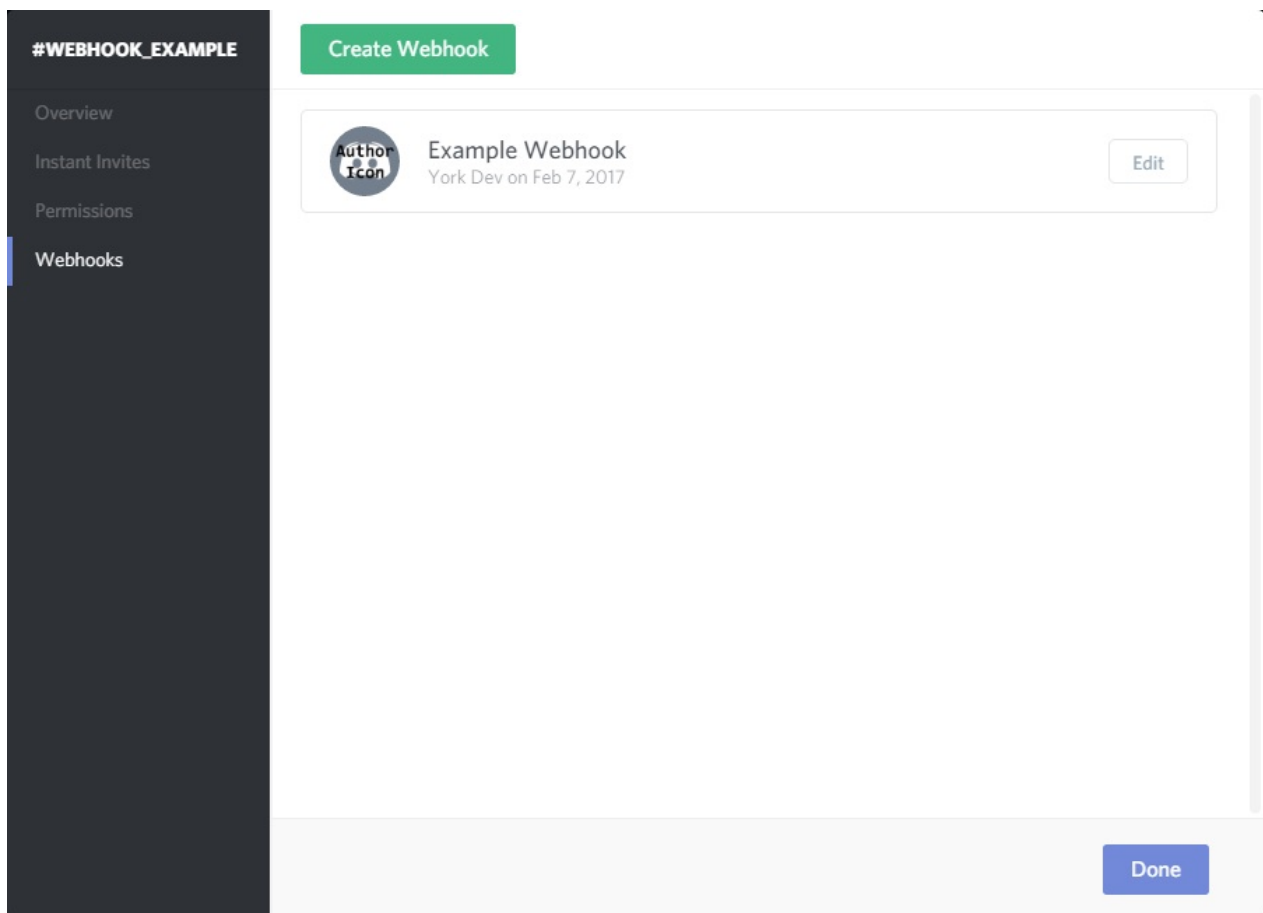
```
const nameAvatar = args.join(" ");
const linkCheck = /https?:\/\/\..+\.(?:png|jpg|jpeg)/gi;
if (!linkCheck.test(nameAvatar)) return message.reply("You must supply an image link.");
const avatar = nameAvatar.match(linkCheck)[0];
const name = nameAvatar.replace(linkCheck, "");
message.channel.createWebhook(name, avatar)
  .then(webhook => webhook.edit(name, avatar)
    .catch(error => console.log(error)))
  .then(wb => message.author.send(`Here is your webhook https://canary.discordapp.com/api/webhooks/${wb.id}/${wb.token}\n\nPlease keep this safe, as you could be exploited.`)
    .catch(error => console.log(error)))
  .catch(error => console.log(error));
```

Alright, now let's throw that together with our bot code and issue the command!



And let's check the

channel webhooks!



Wooo! we did it!

Now we can create webhooks on the fly via our bot code, but in the next [chapter](#) we'll see what we can do with them!

Using Discord Webhooks

In the [last chapter](#) we covered how to create the webhooks via code, which to be honest isn't very useful, in this chapter we will continue where we left off and we will actually use the webhooks we create in some bot code.

Now, one way we could use this, is to grab mentions... For some reason people think it's acceptable to mention me then shortly afterwards remove the mention if I don't respond within X seconds or minutes.

We have two choices, either create a stand-alone bot, or throw it in an existing bot... For the purposes of this guide I will throw the code in a stand alone bot, but it should be pretty self-explanatory how to add this to an existing bot.

NOTE USE AT YOUR OWN RISK. Even though selfbots are not allowed to react to anything other than the account owner, I believe this is perfectly acceptable if it doesn't react in a public fashion, having a **PRIVATE** channel would be acceptable in my opinion.

Let's grab some example code...

```
const Discord = require("discord.js");
const client = new Discord.Client();

client.on("ready", () => {
  console.log("I am ready!");
});

let prefix = "~";
client.on("message", (message) => {
  if (message.author.id === client.user.id || message.author.bot) return;
  let args = message.content.split(" ").slice(1);
  if (message.content.startsWith(prefix + "ping")) {
    message.channel.send("pong!");
  }
});

client.login("SuperSecretBotTokenHere");
```

Now, we've got the example code, we want to take our previously made webhook and grab the `id` and `token` from the URL, let's get it together!

You want to start off by defining your webhook at the top of your code, don't forget to replace `Webhook ID` and `Webhook Token` with their respective values.


```
const Discord = require("discord.js");
const client = new Discord.Client();
const mentionHook = new Discord.WebhookClient("Webhook ID", "Webhook Token");

client.on("ready", () => {
  console.log("I am ready!");
});

let prefix = "~";
client.on("message", (message) => {
  if (message.author.id === client.user.id || message.author.bot) return;
  let args = message.content.split(" ").slice(1);
  if (message.content.startsWith(prefix + "ping")) {
    message.channel.send("pong!");
  }
});

client.login("SuperSecretBotTokenHere");
```

Now, this bit will be a little long winded; but inside the message event you want to check for mentions, now the more mentions you can capture the better, for example there's the `@everyone` and `@here` mentions, role mentions and the direct mentions.

The official documentation has the wonderful `Message.mentioned(data)` boolean, that data can be a `GuildChannel`, `User` or `Role` Object, or a `string` representing the ID of any of the previously mentioned things, so inside the message event create a new `if` statement.

```
client.on("message", (message) => {
  if (message.author.id === client.user.id || message.author.bot) return;
  if (message.mentioned("YOUR USER ID")) {
    // Additional Code
  }
  let args = message.content.split(" ").slice(1);
  if (message.content.startsWith(prefix + "ping")) {
    message.channel.send("pong!");
  }
});
```

Okay, that covers direct mentions, but what about the mentioned `@everyone`, `@here` and role mentions?

Well the `message` object has `mentions` which has both `everyone` and `roles`, so this is what the code will look like so far...

```

client.on("message", (message) => {
  if (message.author.id === client.user.id || message.author.bot) return;
  if (message.mentions.everyone || (message.guild && message.mentions.roles.filter(r => message.guild.member("YOUR USER ID").roles.has(r.id)).size > 0)) {
    // Additional Code
  }
  let args = message.content.split(" ").slice(1);
  if (message.content.startsWith(prefix + "ping")) {
    message.channel.send("pong!");
  }
});

```

Alright, that's the mention detection stuff finished, but let me cover that last bit...

```

(message.guild && message.mentions.roles.filter(r => message.guild.member("YOUR USER ID").roles.has(r.id)).size > 0)

```

The `message.guild` check will make sure we're being mentioned inside a guild channel, now the next bit is a little more complex basically the code is checking for any roles that were mentioned and filtering them against our own roles if any of them match (making the size greater than 0) it'll return true.

So far so good, we're almost half way there... We've set up the conditions to check for mentions, now we just need to ignore a few things, namely ourselves, and bots.

```

client.on("message", (message) => {
  if (message.author.id === client.user.id || message.author.bot) return;
  if (message.mentions.everyone || (message.guild && message.mentions.roles.filter(r => message.guild.member("YOUR USER ID").roles.has(r.id)).size > 0)) {
    if (message.author.id === "YOUR USER ID") return;
    // Additional Code
  }
  let args = message.content.split(" ").slice(1);
  if (message.content.startsWith(prefix + "ping")) {
    message.channel.send("pong!");
  }
});

```

This code may look familiar, and you would be right. It's what we use to get bots to ignore themselves and other bots, but we've changed it slightly so if the `message.author` is the target user (you), then ignore it.

Alright, now we're done with the conditions for the webhook, let's actually use the webhook! Take your code so far (or copy the code from below)...

```
const Discord = require("discord.js");
const client = new Discord.Client();
const mentionHook = new Discord.WebhookClient('Webhook ID', 'Webhook Token');

client.on("ready", () => {
  console.log("I am ready!");
});

let prefix = "~";
client.on("message", (message) => {
  if (message.author.id === client.user.id || message.author.bot) return;
  if (message.isMentioned("YOUR USER ID") || message.mentions.everyone || (message.guild && message.mentions.roles.filter(r => message.guild.member("YOUR USER ID").roles.has(r.id)).size > 0)) {
    if (message.author.id === "YOUR USER ID") return;
    // Additional Code
  }
  let args = message.content.split(" ").slice(1);
  if (message.content.startsWith(prefix + "ping")) {
    message.channel.send("pong!");
  }
});

client.login("SuperSecretBotTokenHere");
```

... and add the following line below where it says `// Additional Code`

```
mentionHook.send("You were mentioned!");
```

Now, let's fill in all of the details we need to get this working (webhook `id` and `token` , and your user `id`)

NOTE This webhook has long since been deleted.

```

const Discord = require("discord.js");
const client = new Discord.Client();
const mentionHook = new Discord.WebhookClient("336099488869384192", "UT_jumpd9cEi3X7Dx
ls0pv9_dscvTSB5oDAVHEWhMh2Psz8n0ZwAVr7JjSszfu5z7BGH");

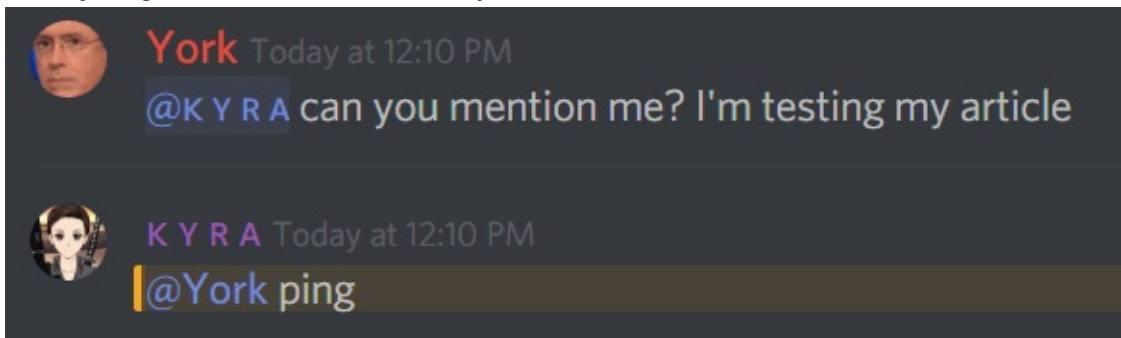
client.on("ready", () => {
  console.log("I am ready!");
});

let prefix = "~";
client.on("message", (message) => {
  if (message.author.id === client.user.id || message.author.bot) return;
  if (message.mentions("146048938242211840") || message.mentions.everyone || (message.guild && message.mentions.roles.filter(r => message.guild.member("146048938242211840").roles.has(r.id)).size > 0)) {
    if (message.author.id === "146048938242211840") return;
    // Additional Code
    mentionHook.send("You were mentioned!");
  }
  let args = message.content.split(" ").slice(1);
  if (message.content.startsWith(prefix + "ping")) {
    message.channel.send("pong!");
  }
});

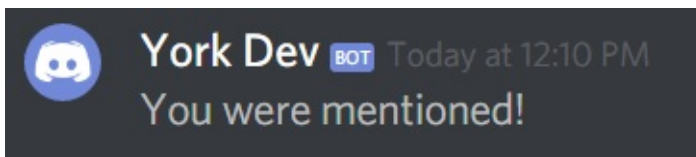
client.login("SuperSecretBotTokenHere");

```

Now, just get someone to mention you!



If all goes well, check the channel you set the webhook for and you should see something like this...



That's basically it for this guide, but you can spice up the webhook notification by grabbing who mentioned you, what channel/guild you were mentioned in and what the content was (in cases of deletion.)

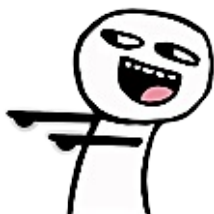
In *Chapter 3* I'll be covering third party websites such as [Zapier](#) and [IFTT](#), which allows you to expand your webhook reach to things like Facebook, Twitter, GMail, and more!

Using Emojis

Here's a fun fact you might not know about bots on Discord: They have access to every single "custom emoji" of every single guild they're in - for free. That's right, you have 1/4 of the features of Nitro, free in your bot, right now! In this page we'll be taking a look at how to take advantage of these emojis, how to access them and how to display them.

What's an Emoji?

Let's start by tearing apart exactly what an Emoji is, how they're configured and how they're accessed. So here, we have an emoji:



When I want to write this emoji in my chat, I simply type `:ayy:` and it turns into the above (smaller, of course, but still). But behind the scenes, 2 things happen for this emoji to show:

- Discord looks up the emoji in my list, finds the one with the name `ayy` and looks up its ID.
- It then sends the *actual* emoji code to the server, which looks like this:
`<:ayy:305818615712579584>`. This is the code that makes up the emoji.
- When a client receives the above, it looks up the URL for the Emoji from its ID, to get the image location. In this case, it's:
`https://cdn.discordapp.com/emojis/305818615712579584.png`.
- As you can see the ID is the only thing that really matters in the URL. This ID is unique to each emoji.

How does Discord.js store emojis?

There are two places where you can grab emojis using discord.js: in the client, and in the guilds. `client.emojis` is a collection of every emoji the client has access to, and `guild.emojis` is a collection of the emojis of a specific guild.

If you've learned anything from [Understanding Collections](#), you might already know how to get something by ID from a collection:

```
const ayy = client.emojis.get("305818615712579584");
```

You might also know how to use `find` to get something with another property - so here, I can get `ayy` through its name:

```
const ayy = client.emojis.find("name", "ayy");
```

Outputting Emoji in chat

But how does one output that emoji to the chat? Well, just like users and roles, emojis have a special `.toString()` method that converts them to the appropriate format. So, `ayy.toString()` will actually output the `<:ayy:305818615712579584>` we saw above, which the client turns into a proper emoji.

You can also take advantage of concatenation and template literals to simplify the task, since they will automatically do the conversion for you:

```
if(message.content === "ayy") {  
    const ayy = client.emojis.find("name", "ayy");  
    message.reply(`${ayy} LMAO`);  
}
```

If you wanted to list all the emojis in a guild, a simple map operation on the collection should give you proper results:



```
if (message.content === "listemojis") {  
    const emojiList = message.guild.emojis.map(e=>e.toString()).join(" ");  
    message.channel.send(emojiList);  
}
```

Reacting with Emojis

You can also use custom emojis as reactions to messages, using `message.react(emoji)`. In the case of custom emojis, you must use the emoji's ID, so you could do something like `message.react(ayy.id)` OR `message.react("305818615712579584")` to add the `ayy` emoji as a reaction.

But what about Unicode Emoji?

Don't forget there is a very extensive collection of emojis that are built into Discord that you can have access to. Discord uses Twemoji, provided by Twitter. You can use those emojis to react to messages directly.

The way that Discord expects those emojis however is that they have to be the *unicode* character, not the "text". Meaning, you can't just do `message.send(":poop:")` and expect to see  appear. You actually need to get the unicode value. How do you do that? Just escape the emoji in chat: `\:poop:` will show as . You can copy/paste that inside your bot's code either in a message string, or as an emoji reaction such as `message.react("\:poop:")`.

Welcome Message every X users

This example will show how to keep an array/object of new users coming into a server. Then, when this array reaches a certain number of users, it shows a message welcoming those users as a group. When your server becomes popular and you get dozens of users every day, you will find this to be much less annoying than welcoming one user at a time!

The events we're going to use in this example:

- `guildMemberAdd` , which triggers when a new user joins the server.
- `guildMemberRemove` , which triggers when a user leaves the server.

Also, we're going to be using a `Discord.Collection()` object to save the users. Why? Because it's available, has great helper methods, and is *built* to support the Discord.js objects we're putting in it!

Initializing the discord collection is simple: `const newUsers = new Discord.Collection();` . This has to be done *outside* of the events we're going to use. I have it at the top of my file, *after* the `const Discord = require('discord.js')` line, obviously.

Adding new members that join to the collection is simple:

```
client.on("guildMemberAdd", (member) => {
  newUsers.set(member.id, member.user);
});
```

If a user leaves while he's on that list though, it would cause your bot to welcome @invalid-user. To fix this, we remove that user from the collection:

```
client.on("guildMemberRemove", (member) => {
  if(newUsers.has(member.id)) newUsers.delete(member.id);
});
```

But wait, where do we welcome users? That's done in `guildMemberAdd` , when the count reaches the number you want:

```
client.on("guildMemberAdd", (member) => {
  const guild = member.guild;
  newUsers.set(member.id, member.user);

  if (newUsers.size > 10) {
    const userlist = newUsers.map(u => u.toString()).join(" ");
    guild.defaultChannel.send("Welcome our new users!\n" + userlist);
    newUsers.clear();
  }
});
```

Two lines require a little more explanation:

- `newUsers.map(u => u.toString()).join(" ");` uses the fancy ES6 `map` function to get a mention for each user in the array, then joins them with a space between each.
- `newUsers.clear()` *resets* empties the cache completely, so it resets to 0.

Multiple servers?

The only issue with the above code is that it would only work if your bot is on a single server. Though this might be alright you, there's a chance you want to support multiple servers. How do we do that? We change `newUsers` to an `Array` instead, and each server gets its own cache. Here is a **complete** example that does nothing but welcome new users:

```
const Discord = require("discord.js");
const client = new Discord.Client();

const newUsers = [];

client.on("ready", () => {
  console.log("I am ready!");
});

client.on("message", (message) => {
  if (message.content.startsWith("ping")) {
    message.channel.send("pong!");
  }
});

client.on("guildMemberAdd", (member) => {
  const guild = member.guild;
  if (!newUsers[guild.id]) newUsers[guild.id] = new Discord.Collection();
  newUsers[guild.id].set(member.id, member.user);

  if (newUsers[guild.id].size > 10) {
    const userlist = newUsers[guild.id].map(u => u.toString()).join(" ");
    guild.channels.get(guild.id).send("Welcome our new users!\n" + userlist);
    newUsers[guild.id].clear();
  }
});

client.on("guildMemberRemove", (member) => {
  const guild = member.guild;
  if (newUsers[guild.id].has(member.id)) newUsers.delete(member.id);
});

client.login("SuperSecretBotTokenHere");
```

Message Reply Array

This sample shows the use of a simple string array to reply specific strings when triggered.

I have often seen the following type of code happen in new bots:

```
client.on("message", (message) => {
  if(message.content === "ayy") {
    message.channel.send("Ayy, lmao!");
  }
  if(message.content === "wat") {
    message.channel.send("Say what?");
  }
  if(message.content === "lol") {
    message.channel.send("roflmaotntmp");
  }
});
```

Ignore the fact that this code doesn't have a prefix and also does not ignore itself or other bots for now. The important fact here is that we can reduce this to a much simpler code, through the use of a array. Well, actually, to make things simpler, let's use an object instead.

First, we declare this object:

```
const responseObject = {
  "ayy": "Ayy, lmao!",
  "wat": "Say what?",
  "lol": "roflmaotntmp"
};
```

This simple object (which can easily be in a JSON file) can then be used in a single command checker, which would look like this:

```
client.on("message", (message) => {
  if(responseObject[message.content]) {
    message.channel.send(responseObject[message.content]);
  }
});
```

That code basically says: "If you find the message content to be a key of the responseObject, send a message containing that key's value".

Boom. Done.

Command with arguments

In [Your First Bot](#), we explored how to make more than one command. These commands all started with a prefix, but didn't have any *arguments* : extra parameters used to vary what the command actually does.

Creating an array of arguments

The first thing that we need to do to use arguments, is to actually separate them. A command with arguments would normally look something like this:

```
!mycommand arg1 arg2 arg3
```

In [Your First Bot](#) we actually simplify our task just a bit: our check for commands uses

```
startsWith() :
```

```
if (message.content.startsWith(config.prefix + "ping")) {  
  message.channel.send("pong!");  
}
```

This means that `!ping whaddup` OR `!ping I'm a little teapot` would both trigger the command, and the bot would respond "pong!". It would just ignore everything after the command because of course we're not doing anything with it.

Let's start with creating an *Array* containing each word after our command, using the

```
.split() function... with a regex :
```

```
if (message.content.startsWith(config.prefix + "ping")) {  
  const args = message.content.split(/\s+/g);  
}
```

You *could* just split using a space `split(" ")` ... however doing this would break if you add an extra space. Especially an issue with mentions on mobile which sometimes add a new space before. Regex is fast enough for the purpose, and this will split on "any number of spaces between each word" instead of "once for each space".

This generates an array that would look like `["!ping", "I'm", "a", "little", "teapot"]` , for instance. We can then access any part of that array using `args[0]` to `args[4]` , which return the string in that array position. And if you want to be more precise and don't want

`args` to contain the `command` , you can just remove it from the array: `const args = message.content.split(/\s+/g).slice(1);` .

For more information on arrays, see [Mozilla Developer Network](#)

If you want, you can then specify argument *names* by referring to the array positions:

```
if (message.content.startsWith(config.prefix + "asl")) {
  const args = message.content.split(/\s+/g).slice(1);
  let age = args[0]; // yes, start at 0, not 1.
  let sex = args[1];
  let location = args[2];
  message.reply(`Hello ${message.author.name}, I see you're a ${age} year old ${sex} from ${location}. Wanna date?`);
}
```

And if you want to be **really** fancy with ECMAScript 6, here's an awesome one (requires Node 6!):

```
if (message.content.startsWith(prefix + "asl")) {
  let [age, sex, location] = message.content.split(/\s+/g).slice(1);
  message.reply(`Hello ${message.author.username}, I see you're a ${age} year old ${sex} from ${location}. Wanna date?`);
}
```

This is called [Destructuring](#) and it's awesome!

Grabbing Mentions

Another way to use arguments, when the command should target a specific user (or users), is to use *Mentions*. For instance, to kick annoying shitposters with `!kick @Xx_SniperBitch_xx @UselessIdiot` can be done with ease, instead of attempting to grab their ID or their name.

In the context of the `message` event handler, all mentions in a message are part of the `msg.mentions` array. Each value in the array is a full `user` resolvable so you can get their ID, name, etc.

Let's build a quick and dirty `kick` command, then. No error handling or mod checks - just straight up! (*Cul Sec*, as the French would say):

```
// Kick a single user in the mention
if (message.content.startsWith(config.prefix + "kick")) {
  let member = message.mentions.members.first();
  member.kick();
}
```

This would be called with, for example, `!kick @AnnoyingUser23`

Variable Length arguments

Let's make the above kick command a little better. Because Discord now supports kick *reasons* in the Audit Logs, the Discord.js `kick()` command also supports an optional `reason` argument. But, because the reason can have multiple words in it, we need to *join* all these words together.

So let's do this now, with what we've already learned, and a little extra:

```
if(message.content.startsWith(config.prefix + "kick")) {  
  let member = message.mentions.members.first();  
  let reason = message.content.split(/\s+/g).slice(2).join(" ");  
  member.kick(reason);  
}
```

So, the reason is obtained by:

- Grabbing the message content
- Splitting it by spaces
- Removing the first 2 elements (the command, `!kick` and the mention, which looks like `<@1234567489213>`)
- Re-joining the rest of the array elements with a space.

To use this command, a user would do something like: `!kick @SuperGamerDude Obvious Troll, shitposting`.

If you're thinking, "What if I have more than one argument with spaces?", yes that's a tougher problem. Ideally, if you need more than one argument with spaces in it, do not use spaces to split the arguments. For example, `!newtag First Var Second Var Third Var` won't work. But `!newtag First Var;Second Var;Third Var;` could work by removing the command, splitting by `;` then splitting by space. Not for the faint of heart!

Let's be fancy with ES6 again!

Destructuring has a `...rest` feature that lets you take "the rest of the array" and put it in a single variable. To demonstrate this, let me show you part of a code I use in a "save message" command. Basically, I store a message to a database, with a name. I call this command using: `!quote <channelid> <messageID> quotename note`, where `quotename` is a single word and `note` may be multiple words. The *actual* command code is unimportant. However, the way I process these arguments, is useful:


```
if(message.content.startsWith(config.prefix + "quote")) {
  const [channelid, messageid, quotename, ...note] = message.content.split(/\s+/g).splice(1);
  // I also support "here" as a channelId using this:
  const channel = channelid == "here" ? message.channel : client.channels.get(channelid);
  // I do the same with message ID, which can be "last":
  const message = messageid === "last" ? msg.channel.messages.last(2)[0] : await channel.messages.get(messageid);
  // pretend for a second this is the rest of the function:
  insertInDB(quotename, channel.id, message.id, note.join(" "));
}
```

A few notes on this code, because I will admit there's some new concepts in it you might not know:

- `...note` is "the rest of the array arguments", so it would be `["this", "is", "a", "note"]`, that's why we `.join(" ")` when we use it.
- `const myVar = condition ? codeWhenConditionTrue : codeWhenFalse;` is called the "ternary operator" in javascript and makes some conditions much more simpler.
- `<Collection>.last(2)[0]` is only available on discord.js#master (the 12.0 beta version) which is not out at the time of writing. The alternative is complex, and beyond what this page tries to teach, so just know it gets "the second to last message".

Going one step further

Now, there's most definitely always room for some optimization, and better code. At this point, "parsing arguments" becomes something you might realize is necessary for *all* of your commands, and writing `.startsWith()` for every command is dull and boring. So, as your next step, consider looking at making [A Basic Command Handler](#). This **greatly** simplifies the creation of new commands.

Selfbots, the greatest thing in the universe

So, my friend, you are wondering what is a selfbot? Well let me tell you a little story.

The first time you see it, you wonder what happened: someone typed for 2 seconds, and a wall of text appears as if by magic. Or something appears and then all of that person's messages disappear one after the other, relegated to `/dev/null` or some other weird place.

That, my friend, is selfbots in action.

What is a selfbot?

First, let's define what a *userbot* is. A Userbot is **a bot that is running under a user account, instead of a true bot account**. Userbots do not have the blue [BOT] tag next to them... and they are **not** accepted by Discord as a valid way to make bots. At least, not since they introduced bot accounts under their [Discord Applications](#) page.

On the other hand, a **selfbot** runs under an **active** user account. In other words, **your** account. When it posts a message, it's like you did. It can edit and delete your messages, has the same permissions you have, is on the same servers. **It's you**.

I AM NOT RESPONSIBLE AND CANNOT BE HELD LIABLE IF YOU MESS UP WITH SELFBOOTS. THIS INCLUDES BUT IS NOT LIMITED TO LOSING PRIVILEGES, GETTING KICKED OR BANNED FROM SERVERS, OR BEING BANNED FROM DISCORD ITSELF

What are the rules for selfbots?

I mentioned that *userbots* are not tolerated by Discord. Selfbots, however, are tolerated under [a specific set of semi-offical rules](#) under which they turn a blind eye:

- A selfbot **must not**, under any circumstance, respond or react to other user's actions. This means it should not respond to commands, autoreply to certain keywords, welcome users to servers, automatically react, etc. **You** must be the only one that can control it.
- A selfbot **must not**, under any circumstance, do 'invite scraping'. This is the action of detecting server invites in chat, and automatically joining a server using that invite. That is akin to creating a virus, and it is not acceptable.
- As selfbots run under your account they are subject to the same Terms of Service. That

is to say, they should not *spam, insult or demean others, post NSFW material, spread viruses or illegal material*, etc. They have to follow the same rules that you follow.

IF, and **only if** you accept the above rules of selfbots, then you may proceed.

Further caveat (which was added recently): Certain user actions done by selfbots will *unverify* your account, as if you had not verified your email. This includes most "Friend" actions.

How do I make a selfbot?

The code that creates selfbots is essentially the same as regular bots, because... well it's the same library, right?

Respond only to you

The first difference is your `message` handler. It should start with a line that prevents the bot from interacting with anyone else's messages (see rule #1 above):

```
client.on("message", message => {
  if(message.author !== client.user) return;
  // rest of the code for commands go here
});
```

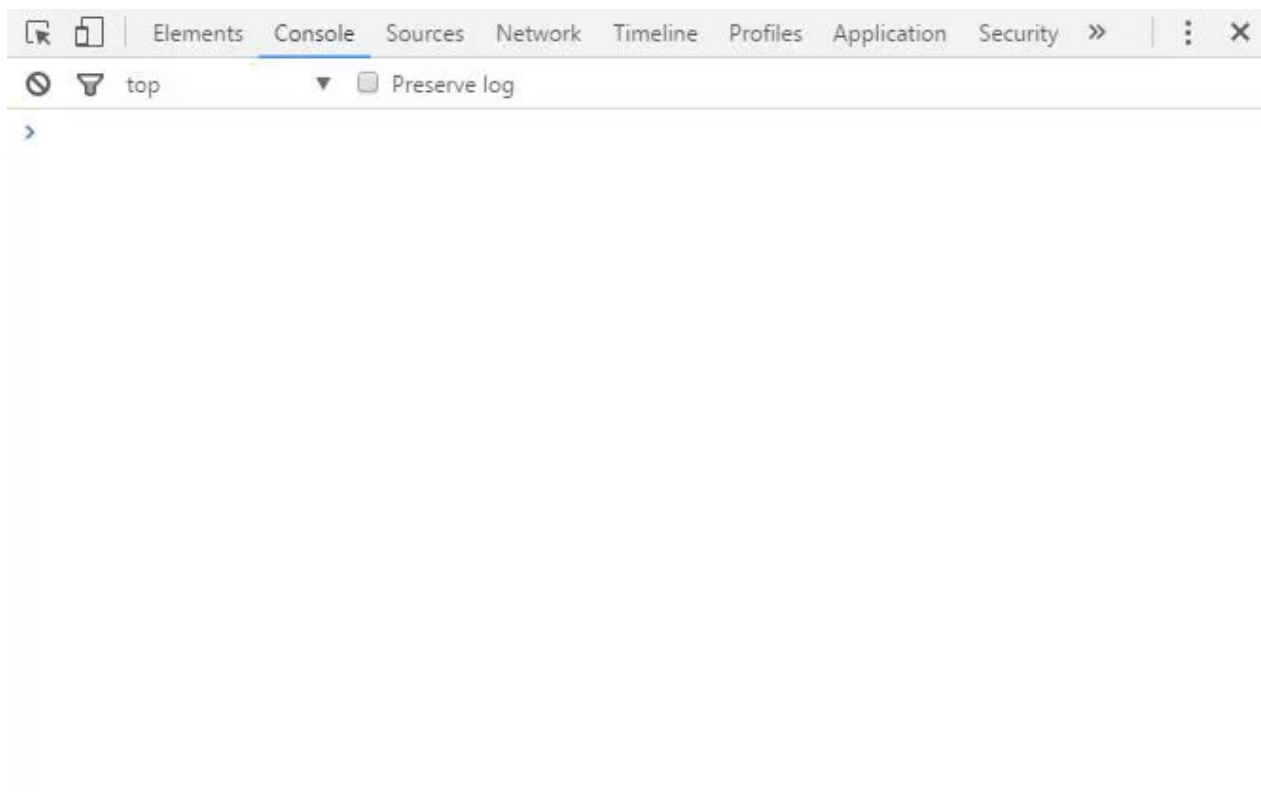
This condition says: "if the author of the message is **not** the bot user, stop processing". This is generic code that works for everyone, and it fulfills our first condition so **don't forget it**.

The Token

The second difference, is how the bot logs in. To log a regular bot, you grab the Bot Token from the Applications Page, then put it in the `client.login()` function.

In the case of a selfbot, this token can be obtained from the Discord application, from the Console:

1. From either the web application, or the installed Discord app, use the **CTRL+SHIFT+I** keyboard shortcut.
2. This brings up the **Developer Tools**. Go to the **Application** tab
3. On the left, expand **Local Storage**, then click on the discordapp.com entry (it should be the only one).
4. Locate the entry called `token`, and copy it.



KEEP YOUR TOKEN SECRET, AND NEVER SHARE IT WITH ANYONE

I've already covered why Bot Tokens should remain secret in the [Getting Started](#) guide. But this is **even more** critical. If someone gets a hold of your personal token, **they are you**. They can pretend to be you, send messages as you. **They can also massively fuck up any server where you have permissions**. Like transfer server control to themselves. Have I stated clearly enough that you should be careful with your token? Ok, good!

Example selfbot commands

Here are a few useful examples of commands that I like to use.

Prune Command

A *Prune* command is used to delete your own messages from the channel you're on. Since there is no way for any server owner to prevent you from deleting or editing your own messages, this will always work. The command is called as `/prune X` where 99 is the maximum number of messages to remove, and the prefix depends on your own prefix.

`fetchMessages()` is limited to 100 messages total, and gets *all* the messages and not just your own. This means you will probably never be able to delete 100 messages since they'll be mixed in with other people's. I generally don't use it to prune more than 10 messages anyway.

```

let prefix = "/"; // always use a prefix it's good practice.
client.on("message", message => {
  if (message.author !== client.user) return;
  if (!message.content.startsWith(prefix)) return; // ignore messages that... you know
  the drill.
  // We covered this already, yay!
  const params = message.content.split(" ").slice(1);
  if (message.content.startsWith(prefix + "prune")) {
    // get number of messages to prune
    let messagecount = parseInt(params[0]);
    // get the channel logs
    message.channel.fetchMessages({
      limit: 100
    })
    .then(messages => {
      let msg_array = messages.array();
      // filter the message to only your own
      msg_array = msg_array.filter(m => m.author.id === client.user.id);
      // limit to the requested number + 1 for the command message
      msg_array.length = messagecount + 1;
      // Has to delete messages individually. Cannot use `deleteMessages()` on selfb
      msg_array.map(m => m.delete().catch(console.error));
    });
  }
});

```

I've included the author check, prefix and params slice at the top for this first example - the other examples won't have that.

Custom /lenny and friends

I love lenny, he's really a great friend to have. And his buddy /justright is also pretty cool! This command is... not really a command in the traditional sense of the word, because it's not built with the regular `if(msg.content.startsWith("something"))` we've seen before.

What it does is, check if the message is *only* the prefix + any of the names in a map. The map itself can be placed outside of the `message` handler, or right before the commands are called:

```

let shortcuts = new Map([
  ["lenny", "( ͡° ͜ʖ ͡°)"],
  ["shrug", "\_(_ツ)_/"],
  ["justright", "ಠ_ಠ"],
  ["tableflip", "(ノಠ_ಠ)ノ彡┐─┐"],
  ["unflip", "┐─┐ノ(ಠ_ಠ)ノ"]
]);

```

There's technically only 2 "new" commands here. The rest are there because I like to do `/tableflip` on my mobile phone, and this does it for me! And you can add your own, of course. Like a `/doubleflip`, for example.

The next step is to add the check at the beginning of your message handler. Well not quite: this code should be *right after* your prefix check, before you split into parameters, for maximum code efficiency.

```
let prefix = "/";
client.on("message", message => {
  if (message.author !== client.user) return;
  if (!message.content.startsWith(prefix)) return;

  // custom shortcut check
  let command_name = message.content.slice(1); // removes the prefix, keeps the rest
  if (shortcuts.has(command_name)) {
    // setTimeout is used here because of a bug in message delays in Discord.
    // Otherwise the message would edit and then "seem" to un-edit itself... ㄟ(っ)_/_
    setTimeout(() => {message.edit(shortcuts.get(command_name))}, 50);
    return;
  }

  // The Rest of your code
  // ...
});
```

A complete implementation

Don't really care for coding this all yourself? Want a complete, functional, awesome selfbot all done for you? I got you covered, obviously! Check out [Evie's Selfbot](#) on Github!

The Power of Eval

So, you've seen a lot of people on the Discord.js Official server use some sort of `eval` command. When they do, magical things happen - arbitrary javascript is executed and output is produced. MAGIC!

What is eval exactly?

In JavaScript (and node), `eval()` is a function that evaluates any string *as javascript code* and actually executes it. Meaning, if I `eval(2+2)`, eval will return `4`. If I eval

`client.guilds.size`, it'll return however many guilds the bot is currently on. And if I eval

`client.guilds.map(g=>g.name).join('\n')` then it will return a list of guild names separated by a line return. Cool, right?

But eval is dangerous

I'll say this in a way that's probably dead simple to understand: *Giving someone access to*

`eval()` *is literally like sitting them **at your computer**, with **full admin access**, and then **stepping out of the room**.* `eval()` in browser javascript is trivial and not dangerous -

you're running in your own browser, anything you fuck up is going to be on your own PC, not the web servers.

But `eval()` in **node** is really, really dangerous and powerful. Because it can run anything **you** run as a bot, and it can also run code you're not **expecting** to run, if someone else has access to it. **Node.js** has access to your **hard drive**. The whole thing. Every bit of it. To

understand what this means, look at the following command: `rm -rf / --no-preserve-root`.

Do you know what this command does? **It deletes your entire server's hard drives**. I mean, it only works on Linux, but most VPS systems and most hosting providers are on UNIX-based systems.

Another thing that's on your hard drive is passwords. Have a config file with your database password? I can get that. `config.json` with your token and other API keys in it? I can grab that easy. Hell I can just grab your token straight from the `client` object if I know how to.

Securing your eval

So first, we need to understand the #1 rule when using `eval` commands:

NEVER EVER GIVE EVAL PERMS TO ANYONE ELSE

I don't care if it's a server owner, someone you've been talking to for months, you **cannot** trust anyone with eval. There's only one exception to this rule: Someone you know **in real life** that you can punch in the face when they actually destroy half your server or mistakenly ban everyone in every server your bot is in. Eval bypasses any command-based permission you might have, it bypasses all security checks. Eval is all powerful.

So how do you secure it? Simple: only allow use from your own user ID. So for example my user ID is `139412744439988224` so I check whether the message author's ID is mine, which we added into our config at the start:

```
{
  //the rest of the config
  "ownerID": "139412744439988224"
}
```

In the code for the bot:

```
if(message.author.id !== config.ownerID) return;
```

It's as simple as that to protect the command directly inside of your condition or file or whatever. Of course, if you have some sort of command handler there's most likely a way to restrict to an ID too. This isn't specific to discord.js : there's always a way to do this. If there isn't (if a command handler won't let you restrict by ID), then you're using the **wrong lib**.

Simple Eval Command

So now you've been thoroughly briefed on the dangers of Eval, let's take a look at how to implement a simple eval command.

First though I strongly suggest using the following function (plop it outside of any event handler/functions you have, so it's accessible anywhere). This function prevents the use of actual mentions within the return line by adding a zero-width character between the `@` and the first character of the mention - blocking the mention from happening.

NOTE: EITHER of the following clean snippets are **REQUIRED** to make the eval work.


```
function clean(text) {
  if (typeof(text) === "string")
    return text.replace(/`/g, "`" + String.fromCharCode(8203)).replace(/@/g, "@" + String.fromCharCode(8203));
  else
    return text;
}
```

It's ES6 variant:

```
const clean = text => {
  if (typeof(text) === "string")
    return text.replace(/`/g, "`" + String.fromCharCode(8203)).replace(/@/g, "@" + String.fromCharCode(8203));
  else
    return text;
}
```

Alright, So let's get down to the brass tax: The actual eval command. Here it is in all its glory, assuming you've followed this guide all along:

```
client.on("message", message => {
  const args = message.content.split(" ").slice(1);

  if (message.content.startsWith(config.prefix + "eval")) {
    if(message.author.id !== config.ownerID) return;
    try {
      const code = args.join(" ");
      let evaled = eval(code);

      if (typeof evaled !== "string")
        evaled = require("util").inspect(evaled);

      message.channel.send(clean(evaled), {code:"xl"});
    } catch (err) {
      message.channel.send(`\`ERROR\` \`${err}\`xl\n${clean(err)}\n\`\`\``);
    }
  }
});
```

That's it. That's the command. Note a couple of things though:

- Your IDE or editor might scream at the `eval(code)` for being `unsafe`. See the rest of this page for WHY it says that. Can't say you haven't been warned!
- If the response isn't a string, `util.inspect()` is used to 'stringify' the code in a safe way that won't error out on objects with circular references (like most Collections).
- If the response is more than 2000 characters this will return nothing.

I AM NOT RESPONSIBLE IF YOU FUCK UP, AND NEITHER ARE ANY OF THE DISCORD.JS USERS AND DEVELOPERS

Hopefully the warnings were clear enough to help you understand the dangers... but the idea of eval is still attractive enough that you'll use it for yourself anyway!

Embeds and Messages

You've seen them all around - these sexy dark grey boxes with a nice color on the left, images, and **tables** oh my god. So nice-looking, right? Well, let me show you how to make them!

Fair Warning: Embeds might look nice but they can be disabled through permissions and user preferences, and will not look the same on mobile - especially complex ones. It's strongly recommended *not* to use them unless you have a text-only fallback. Yes they're nice, but, don't use them if you don't *need* to!

Object-based embeds

Here are a few rules for embeds:

- Every field is optional
- At least one field must be present
- No field can be empty, null, or undefined.

Those aren't just guidelines, they are rules, and breaking those rules means your embed will not send - it will return `Bad Request`.

There are 2 ways to do embeds. The first, is by writing the embed yourself, as an object. Here's a very, very basic embed that writes on a single line:

```
message.channel.send({embed: {  
  color: 3447003,  
  description: "A very simple Embed!"  
}});
```

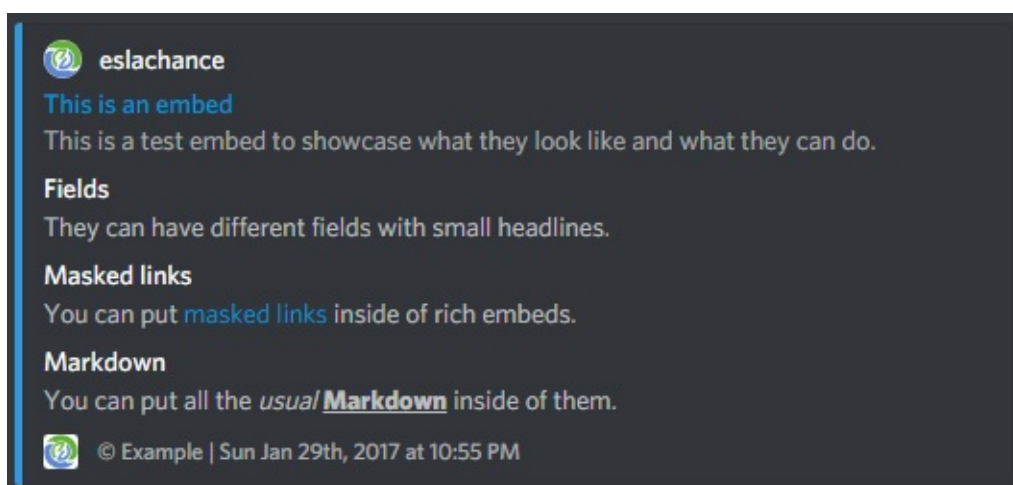
The `color` determines the bar on the left (here, a very nice blue), and the `description` is the main contents of the message.

Adding "Fields"

Fields are what can make embeds really nice - each field can have a title and value, and fields can also be stacked horizontally - as columns. Here's a more complex example of an embed that has many different fields, as well as icons and a footer:

```
message.channel.send({embed: {
  color: 3447003,
  author: {
    name: client.user.username,
    icon_url: client.user.avatarURL
  },
  title: "This is an embed",
  url: "http://google.com",
  description: "This is a test embed to showcase what they look like and what they c
an do.",
  fields: [{
    name: "Fields",
    value: "They can have different fields with small headlines."
  },
  {
    name: "Masked links",
    value: "You can put [masked links](http://google.com) inside of rich embeds."
  },
  {
    name: "Markdown",
    value: "You can put all the *usual* **__Markdown__** inside of them."
  }
],
  timestamp: new Date(),
  footer: {
    icon_url: client.user.avatarURL,
    text: "© Example"
  }
}
});
```

This results in the following:



RichEmbed Builder

There is an alternative to using straight-up objects, which might be simpler in some cases - it's certainly cleaner! It's using the `RichEmbed` builder.

The same rules apply for `RichEmbed` as does normal ones. In fact, the builder is just a shortcut to get the same object and offers no more, no less functionality. Here is a similar example to the one above, using the `RichEmbed`. It also has a nice fancy image, to boot!

```
const embed = new Discord.RichEmbed()
  .setTitle("This is your title, it can hold 256 characters")
  .setAuthor("Author Name", "https://i.imgur.com/lm8s41J.png")
  /*
   * Alternatively, use "#00AE86", [0, 174, 134] or an integer number.
   */
  .setColor(0x00AE86)
  .setDescription("This is the main body of text, it can hold 2048 characters.")
  .setFooter("This is the footer text, it can hold 2048 characters", "http://i.imgur.com/w1vhFSR.png")
  .setImage("http://i.imgur.com/yVpymuV.png")
  .setThumbnail("http://i.imgur.com/p2qNFag.png")
  /*
   * Takes a Date object, defaults to current date.
   */
  .setTimestamp()
  .setURL("https://discord.js.org/#/docs/main/indev/class/RichEmbed")
  .addField("This is a field title, it can hold 256 characters",
    "This is a field value, it can hold 2048 characters.")
  /*
   * Inline fields may not display as inline if the thumbnail and/or image is too big.
   */
  .addField("Inline Field", "They can also be inline.", true)
  /*
   * Blank field, useful to create some space.
   */
  .addBlankField(true)
  .addField("Inline Field 3", "You can have a maximum of 25 fields.", true);

message.channel.send({embed});
```

Which produces the following:



How ever, if you want a quick and easy way of designing an embed, try out this new tool by [leovoe1](#) , the [Embed Visualizer](#) shows you exactly how your embed will look, and will generate the code for you at the press of a button.

Installing and Using a proper editor

Let's take a moment to appreciate the fact that the best code, is not just code that *works* but also code that is *readable*. And *readable* code is already easier to troubleshoot. To simplify the life of any code, a good editor is key. A good editor will tell you where your mistakes are, validate your code, give you best practices, and some will even run your code for you.

In this tutorial we'll be looking at one such editor: Atom. Now, I'm not personally biased towards or away from Atom, but the editor, and some help from people who use it, were readily available, so this is the one I'm showing you.

Other alternatives would be VS Code and Sublime Text 3. You'll have to look up specific instructions for those editors on your own.

Getting Atom Installed

Far be it from me to actually show you how to install stuff on your computer - I will pretend for a moment you know what you're doing, and give you the outline:

1. Go to the [Atom.io website](https://atom.io).
2. Click the big red **Download Windows Installer** button (statistically, you're probably on Windows).
3. Once the .exe is downloaded, run it, and install it.

Atom starts off with a simple interface with a nice welcome screen and a guide, so if you feel like it, go ahead and read up a bit.

Opening your project

To the contrary of some more basic editors, Atom has the ability to open a project *folder* so you don't have to keep opening files individually.

- Go to **File, Open Folder** (CTRL+SHIFT+O)
- Browse to the location of your main bot file (mybot.js, app.js, or whatever)
- Click on **Select Folder** (*Note: your files will not appear here, only the folder structure. Don't worry they haven't gone anywhere*)

On the left of the editor you will have all the files. Just double-click any of them to open it. To clean things up, you can close the Welcome Screen and such.

Already, you can see that this code looks super clean, and colorful. Also the dark theme doesn't hurt the eyes so that's a plus.

Getting ESLint installed

A 'Linter' is a plugin or app that verifies your code to tell you where the errors are, and also helps in formatting the code with proper indentation and styles.

There's obviously a debate with what linter you should use. ESLint? JSHint? Some other obscure library? I'll use ESLint because... I know how to use it. Not endorsing it in any way!

ESLint works in 2 parts. The first is the plugin for Atom, which is installed through the command line:

- Hit Windows+R on your keyboard
- Type in `cmd` then press Enter
- Enter the command `apm install linter-eslint` then press Enter
- This may take a few seconds to a minute to complete, be patient.
- Keep the Command Prompt open.

Next, we need to install the eslint npm module, which is what actually verifies your code.

`linter-eslint` just integrates those results in Atom.

In the command line, type `npm i -g eslint` then press Enter. Once it completes (it will show you a list of eslint and installed dependencies).

You'll need to restart Atom to take the changes into effect.

Setting up ESLint options

ESLint, by default, will expect *very* strict code from you, and will complain for a number of things. If you use the wrong type of quotes (single vs double), or if you use 4 instead of 2 spaces, alarms will go blaring.

ESLint options can be global or local to your project. Let's do a local one as an example.

- Create a new file in Atom
- Save it as `.eslintrc.json` in your project folder (where your `app.js/mybot.js` is)
- Copy the code below inside the file and save it again:


```
{
  "env": {
    "es6": true,
    "node": true
  },
  "extends": "eslint:recommended",
  "parserOptions": {
    "sourceType": "module"
  },
  "rules": {
    "no-console": "off",
    "indent": [
      "error",
      2
    ],
    "linebreak-style": [
      "error",
      "windows"
    ],
    "quotes": [
      "warn",
      "double"
    ],
    "semi": [
      "warn",
      "always"
    ]
  }
}
```

Don't be daunted by this: it's just a little config in JSON format. I won't go into details of what each option does, they are all explained [in the docs](#) but this is the rules I'm using. Basically:

- Force linter to support node
- Force linter to support ES6 code `<3`
- Indent to 2 spaces (and not a 4-space-sized tab, ugh)
- Make linebreaks Windows styles
- Warn on single quotes, prefer double quotes
- Warn when the semicolon is absent

Using Git to share, update and convert your code

Have you ever come to a point where you're editing code, removing and adding and changing stuff and all of a sudden you realize, *Shit, I deleted this piece and I need to rewrite or re-use it now. Damn!*

Have you ever wished there was a simpler way to transfer your code to your hosting, rather than having to connect to an FTP, or zip your file, upload to your host and unzip... you know the drill, right? Ugh!

Have you ever wished you could easily share your code and have other people help you out in some bits, making your bot better?

If you answered **Yes** to any of these questions, boy do I have a product for you! It's called `git` and it's a pretty magical tool.

Pre-requisites and Software

To take full advantage of `git` you need to first have *at least* completed the [Adding a config.json file](#) walkthrough. This means that at the very least, your token is located in a separate file, and not within your javascript files. Having the prefix and the owner ID in that separate configuration file means that you get the advantage of easy testing: Change the prefix and token, and you have a secondary bot you can test your code with, wooh!

What else do you need? `git` itself, of course! For Windows get [Git SCM](#), on Linux run `sudo apt-get install git-all` OR `sudo yum install git-all` depending on your distro's install method. Mac users also have a [Git SCM](#) installer.

Anything else? Eeeeh, nope! That's all, really. Let's get on to usage!

Initialization and `.gitignore`

So the first baby step into the world of git, is to initialize your project folder as a git repository. To do that, you need to navigate to that folder in your command line. Or use an OS shortcut:

- Mac users can use [this trick from lifehacker.com](#).
- Windows users, remember the magic trick: SHIFT+Right-Click in your project folder, then choose **Open command window here**.

- Most Linux distros have an **Open in Terminal** option. But you use Linux, you can figure it out, right?

Once you have your prompt open in that folder, go ahead and run `git init`. It doesn't have any options or questions - it just inits the folder.

Ignoring Files

In git, one of the most important files is `.gitignore` which, as the name would imply, ignores certain files. This is pretty critical with node.js apps, and our bot: you definitely want and need to ignore both the `node_modules` folder and the `config.json` file. The former because you don't want thousands of files being uploaded (and they don't work on other systems anyway), the latter because you don't want your token to end up on a public `git` repository!

There are pre-built `.gitignore` files you can grab off the internet, but for the purpose of this exercise you only need 2 lines in that file:

```
node_modules
config.json
```

But how do you create it? Linux and Mac users, you can probably easily create the file directly. Windows users, you have to take a tiny detour - Windows doesn't like files with only an extension and no filename. Don't worry though it's simple. Start by creating a new file called, for example, `gitignore.txt` and put in the 2 lines above. Then, in your console, run `rename gitignore.txt .gitignore` and this will rename it as expected.

Your first `git` command

And now we're ready to start to `git gud` (no no don't type that it's a meme!)

The first command you can try is `git status`. This will show you a list of all your files and in all subfolders and indicate them as **untracked files**, meaning they're not being tracked (saved) by git.

Let's resolve that with our second command, which essentially tells git to track all these files and start doing its magic on them: `git add .` where `.` is just a shortcut to 'all the files including subfolders'.

And now it's time to *commit* to the task. *Committing* basically means you're creating a snapshot of your project, that is forever saved in your history. A commit can be as small as a tiny bugfix of one line or rewriting a complete function or module. Committing is always going

to be with a message that describes the change, so smaller commits can be easier to manage and track in the future. So here goes:

```
git commit -m 'Initial Project Commit'
```

This will output a new commit with some stats including the number of changed lines, and the list of changed files. Great! Now we've got a commit. Funny story: you can continue committing changes locally, and do all the great things `git` does without every pushing to a remote repository. But that's what we're here for, right? So let's move on!

Creating a GitHub/GitLab/BitBucket account

With `git` you have a lot of choices when it comes to hosting your projects online. Pushing to an online repository gives you the safety in backups - your whole commit history is available, and nothing is lost whatever happens. Also gives you the ability to share your projects either publicly or privately with other people, and have them contribute to your project if you want them to.

The 3 main `git` hosting services that are free are GitHub, GitLab and BitBucket. While all three are free, GitHub is the only one that doesn't offer private repository - but it's also the most *stable* and *open*. I'll be going with GitHub in this case, but most of the steps I'll take here are the same for all 3 services. You just need an account and you're good to go!

So let's go ahead and prepare GitHub. On [GitHub.com](https://github.com), create an account and then create a **New Repository**. Give it a name that identifies your project - it doesn't need to be identical to your folder name, and you can change it later if you want. Don't add a readme for now.

In the page that displays after creating the repository you actually get the instructions you need to do, under **push an existing repository from the command line**:

```
git remote add origin git@github.com:your-username/your-repo-name.git
git push -u origin master
```

Obviously, change `your-username` and `your-repo-name` to the appropriate strings. The first line *links* your local repository to the GitHub website. The second line actually takes all your local commit history, and it *pushes* it directly to github. Note that this action will ask you to enter your github username and password.

'Wait, really? That's it?' you ask. Yep, your code is now on github. Refresh the github page and you'll see all your code there - without the `node_modules` and `config.json` if you've followed correctly!

Pushing further updates

Linking your account doesn't save you from future commands. Actually, every time you make a change and want to push it to the repository you have to use that `push` command again. Note that you don't *need* to push every commit individually - you can work for a day and do multiple commits, and push them all at once at the end of the day.

A small shortcut that you can use is to `add` and `commit` at the same time. This only works if you have changed files but not created new ones (for that just do `git add .` separately). So if you're done just small changes, you can use this:

```
git commit -am 'Fixed permission issue in 8ball command'
```

And don't forget to `git push origin master` again!

Getting your code to another machine

So, let's say you have a VPS hosting somewhere. Or a Raspberry Pi. Or whatever other machine. You want to get that code of yours onto it. You first need to follow the `install` steps above to get `git` installed on the machine, and then you need to *clone* the repository from the git remote:

```
git clone git@github.com:your-username/your-repo-name.git
```

That's all! Well, you need to enter your username and password again, but once that's done, that code is available!

Don't forget that any file that's been added to your `.gitignore` file isn't present here, so you'll need to do 2 things:

1. Create your `config.json` again, either with the same token and prefix, or different ones.
2. Install your node modules again. If you've properly run `npm init` and installed all your modules with the `-s` or `--save` argument, you should only need to do `npm install .` Otherwise, you need to install all of them again, for example using `npm install discord.js`

Now, remember that these files aren't synced so they won't be overwritten or modified whenever you push or pull to and from the repo.

Speaking of which, any time you have `push` ed to GitHub, you can get an updated copy of the code by simply running `git pull origin master` .

You'll have to reset your bot for the changes to take effect, of course.

Some Last Tips

The process isn't one-way. If you modify files on *any* computer you can `git push origin master` and `git pull origin master` on the other one. Be careful about editing on both locations though, as this may cause conflicts that are annoying to resolve (and are beyond this guide's scope)

IF you ever have 'temporary' changes and you want to overwrite them - like a quick path on your VPS that you've also fixed locally - you can avoid the whole 'conflict' process by *Resetting* a repository to your last `pull` . Simply run `git reset --hard HEAD` to erase any local changes, then run `git pull origin master` to grab the latest changes.

You can connect to multiple remote repositories by running the `remote add` command above. You'll need a different name, for example instead of `origin` you can call a remote `gitlab` and then any command should reflect that, like `git pull gitlab master` .

There is a **lot** more to `git` than what was shown here (and I'm aware even that's already a shitton of information). You can create and merge `branches` , `revert` to previous commits, make and accept `PR` s... [There is a lot more](#) that you can you can learn!

But for now... I think this massive wall of text is plenty.

<https://www.youtube.com/watch?v=kpci6V8969g>

A lot of people have asked how to host their bots on a raspberry pi, and there's a lot of conflicting information on the internet, well my fellow idiots, I'm here to give you the information you'll need to get your discord.js bot running on node v6 on a raspberry pi.

I'm using a Raspberry Pi 2 Model B running in headless mode with a WiFi dongle, on Raspbian 8 (Jessie)

Commands used in the video;

`lsb_release -a` - This will show you what version of Raspbian you're running.

`node --version` - This will show you what version of Node you're running.

`npm --version` - This will show you what version of NPM you're running.

`sudo apt-get remove --purge node* npm*` - This will remove **ALL** traces of Nodered.

NOTE: There has been conflicting information regarding the purge command, a small number of people have claimed it has messed up their Raspbian installation, so please **USE AT YOUR OWN RISK**, I cannot be held responsible if it does go wrong, as I encountered no issues running these commands in this order.

`curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -` - This will download everything required for node to be installed.

NOTE: You can swap out `setup_6.x` for `setup_8.x` to update straight to Node 8.

`sudo apt-get install -y nodejs` - This will install node for you, the `-y` flag will auto-accept all terms and agreements for you.

`sudo npm install pm2 -g` - This will install pm2 (Process Manger 2), this *isn't* required, but it's a great thing to have to restart your bots if you lose power or crash.

`pm2 startup` - This will begin the process of creating a start up script.

Then follow the on screen instructions

<https://www.youtube.com/watch?v=02jt7l3eBK0>

EDIT: It's been brought to my attention that there's a typo, at 7:13 I say windows-built-tools, it should read;

```
npm i -g windows-build-tools OR npm i -g ffmpeg-binaries
```

In the last episode, I showed you how to get a bot up and running on a raspberry pi, in this episode I show you how to get a music bot up and running on the same raspberry pi, and I go over why you shouldn't host a music bot on the raspberry pi.

Just because you **CAN** do it, doesn't mean you **SHOULD**.

I'm using a Raspberry Pi 2 Model B running in headless mode with a WiFi dongle, on Raspbian 8 (Jessie)

Commands used in the video;

```
npm install
```

 - This will install the bot from the package.json file.

```
sudo apt-get install libav-tools -y
```

 - This will install an FFMPEG compatible transcoder.

```
pm2 start app.js --name Music
```

 - This will add the music bot to PM2's list, and starts it up.

```
pm2 list
```

 - This will list all currently stored items.

```
pm2 monit
```

 This will pull up a live monitor so you can watch the resources.

LINK DUMP

AoDude's OhGodMusicBot is available [here](#)

You will require a free discordapp.com account to use the discord service, sign up [here](#)

You can join the "*An Idiot's Guide Official Server*" [here](#)

If you'd like to support me on Patreon, you can do so [here](#)

Don't forget to join the "*Official Discord.js Server*" [here](#)

Check out the Discord Developers portal [here](#)

Discord.js Official Documentation is available [here](#)

Don't forget, if you want to invite your bots to server, you'll need to get an invite link, you can use the Discord Permissions Calculator [here](#) to set up your permissions, and generate an invite link.

List of Atom addons and themes I use;

Addons:

- atom-beautify - Glavin001,
- highlight-selected - richrace,
- linter-eslint - AtomLinter,
- seti-icons - wyze and
- tokamak-terminal - vertexclique

Themes:

- seti-syntax, seti-ui - jesseweed

<https://www.youtube.com/watch?v=rVfjZrqpQ7o>

Welcome to the first episode of my guide which will help you create a Discord Bot using the amazing Discord.js library, and I hope you follow along and create your very own bot!

If I got anything wrong in this episode, please keep it to yourself LOL, ignorance is bliss, nah I'm kidding, let me know what I got wrong and I'll correct it in the next episode.

Please let me know what you think to the series, and suggest things for me to cover in future episodes, I have a mental road map I will be following, but I may throw in the odd episode dedicated to requests.

LINK DUMP

All of the tutorial source code is available [here](#)

You will require a free discordapp.com account to use the discord service, sign up [here](#)

You can join the "*An Idiot's Guide Official Server*" [here](#)

If you'd like to support me on Patreon, you can do so [here](#)

Don't forget to join the "*Official Discord.js Server*" [here](#)

Check out the Discord Developers portal [here](#)

Discord.js Official Documentation is available [here](#)

Don't forget, if you want to invite your bots to server, you'll need to get an invite link, you can use the Discord Permissions Calculator [here](#) to set up your permissions, and generate an invite link.

List of Atom addons and themes I use;

Addons:

- atom-beautify - Glavin001,
- highlight-selected - richrace,
- linter-eslint - AtomLinter,
- seti-icons - wyze and
- tokamak-terminal - vertexclique

Themes:

- seti-syntax, seti-ui - jesseweed

<https://www.youtube.com/watch?v=8AiZBdcPKOM>

Welcome back to my guide, helping you create a Discord Bot using the amazing Discord.js library I hope you follow along and create your very own bot!

Please let me know what you think to the series, and suggest things for me to cover in future episodes, I have a mental road map I will be following, but I may throw in the odd episode dedicated to requests.

LINK DUMP

You can read up on `template literals` [here](#)

All of the tutorial source code is available [here](#)

You will require a free discordapp.com account to use the discord service, sign up [here](#)

You can join the *"An Idiot's Guide Official Server"* [here](#)

If you'd like to support me on Patreon, you can do so [here](#)

Don't forget to join the *"Official Discord.js Server"* [here](#)

Check out the Discord Developers portal [here](#)

Discord.js Official Documentation is available [here](#)

Don't forget, if you want to invite your bots to server, you'll need to get an invite link, you can use the Discord Permissions Calculator [here](#) to set up your permissions, and generate an invite link.

Music in the video:

Local Forecast - Slower by Kevin MacLeod incompetech.com

Licensed under Creative Commons: [By Attribution 3.0 License](#)

List of Atom addons and themes I use;

Addons:

- atom-beautify - Glavin001,
- highlight-selected - richrace,
- linter-eslint - AtomLinter,
- seti-icons - wyze and
- tokamak-terminal - vertexclique

Themes:

- seti-syntax, seti-ui - jesseweed

<https://www.youtube.com/watch?v=oDJrtA1YORw>

Welcome back to my guide, helping you create a Discord Bot using the amazing Discord.js library I hope you follow along and create your very own bot!

In this episode we cover the main guildEvents, such events for members joining, leaving, been banned or unbanned for example!

Please let me know what you think to the series, and suggest things for me to cover in future episodes, I have a mental road map I will be following, but I may throw in the odd episode dedicated to requests.

LINK DUMP

All of the tutorial source code is available [here](#)

You will require a free discordapp.com account to use the discord service, sign up [here](#)

You can join the "*An Idiot's Guide Official Server*" [here](#)

If you'd like to support me on Patreon, you can do so [here](#)

Don't forget to join the "*Official Discord.js Server*" [here](#)

Check out the Discord Developers portal [here](#)

Discord.js Official Documentation is available [here](#)

Don't forget, if you want to invite your bots to server, you'll need to get an invite link, you can use the Discord Permissions Calculator [here](#) to set up your permissions, and generate an invite link.

Music in the video:

Local Forecast - Slower by Kevin MacLeod incompetech.com

Licensed under Creative Commons: [By Attribution 3.0 License](#)

List of Atom addons and themes I use;

Addons:

- atom-beautify - Glavin001,
- highlight-selected - richrace,
- linter-eslint - AtomLinter,
- seti-icons - wyze and
- tokamak-terminal - vertexclique

Themes:

- seti-syntax, seti-ui - jesseweed

<https://www.youtube.com/watch?v=KKmyTfGbY54>

Welcome back to my guide, helping you create a Discord Bot using the amazing Discord.js library I hope you follow along and create your very own bot!

We're back with more events, the next episode we'll be tackling role events, that should be a juicy one!

Please let me know what you think to the series, and suggest things for me to cover in future episodes, I have a mental road map I will be following, but I may throw in the odd episode dedicated to requests.

LINK DUMP

All of the tutorial source code is available [here](#)

You will require a free discordapp.com account to use the discord service, sign up [here](#)

You can join the "*An Idiot's Guide Official Server*" [here](#)

If you'd like to support me on Patreon, you can do so [here](#)

Don't forget to join the "*Official Discord.js Server*" [here](#)

Check out the Discord Developers portal [here](#)

Discord.js Official Documentation is available [here](#)

Don't forget, if you want to invite your bots to server, you'll need to get an invite link, you can use the Discord Permissions Calculator [here](#) to set up your permissions, and generate an invite link.

Music in the video:

Local Forecast - Slower by Kevin MacLeod incompetech.com

Licensed under Creative Commons: [By Attribution 3.0 License](#)

List of Atom addons and themes I use;

Addons:

- atom-beautify - Glavin001,
- highlight-selected - richrace,
- linter-eslint - AtomLinter,
- seti-icons - wyze and
- tokamak-terminal - vertexclique

Themes:

- seti-syntax, seti-ui - jesseweed

<https://www.youtube.com/watch?v=S5DVdjLQA44>

Welcome back to my guide, helping you create a Discord Bot using the amazing Discord.js library I hope you follow along and create your very own bot!

And we're back, and this time we're taking on roles! They're not as bad as people make out once you know how to use them, we cover them a fair bit in this episode!

Please let me know what you think to the series, and suggest things for me to cover in future episodes, I have a mental road map I will be following, but I may throw in the odd episode dedicated to requests.

LINK DUMP

All of the tutorial source code is available [here](#)

You will require a free discordapp.com account to use the discord service, sign up [here](#)

You can join the "*An Idiot's Guide Official Server*" [here](#)

If you'd like to support me on Patreon, you can do so [here](#)

Don't forget to join the "*Official Discord.js Server*" [here](#)

Check out the Discord Developers portal [here](#)

Discord.js Official Documentation is available [here](#)

Don't forget, if you want to invite your bots to server, you'll need to get an invite link, you can use the Discord Permissions Calculator [here](#) to set up your permissions, and generate an invite link.

Music in the video:

Local Forecast - Slower by Kevin MacLeod incompetech.com

Licensed under Creative Commons: [By Attribution 3.0 License](#)

List of Atom addons and themes I use;

Addons:

- atom-beautify - Glavin001,
- highlight-selected - richrace,
- linter-eslint - AtomLinter,
- seti-icons - wyze and
- tokamak-terminal - vertexclique

Themes:

- seti-syntax, seti-ui - jesseweed

<https://www.youtube.com/watch?v=zdQpIH3fwbU>

Welcome back to my guide, helping you create a Discord Bot using the amazing Discord.js library I hope you follow along and create your very own bot!

This is part 1 of a 2 parter, wewlad, I had a lot to get recorded in this episode, turned out alright didn't it, in this episode we're breaking down our bot into separate files, we're doing event and command handlers, this will clean up our code for what's coming in the next few videos

Please let me know what you think to the series, and suggest things for me to cover in future episodes, I have a mental road map I will be following, but I may throw in the odd episode dedicated to requests.

LINK DUMP

All of the tutorial source code is available [here](#)

You will require a free discordapp.com account to use the discord service, sign up [here](#)

You can join the *"An Idiot's Guide Official Server"* [here](#)

If you'd like to support me on Patreon, you can do so [here](#)

Don't forget to join the *"Official Discord.js Server"* [here](#)

Check out the Discord Developers portal [here](#)

Discord.js Official Documentation is available [here](#)

Don't forget, if you want to invite your bots to server, you'll need to get an invite link, you can use the Discord Permissions Calculator [here](#) to set up your permissions, and generate an invite link.

Music in the video:

Local Forecast - Slower by Kevin MacLeod incompetech.com

Licensed under Creative Commons: [By Attribution 3.0 License](#)

List of Atom addons and themes I use;

Addons:

- atom-beautify - Glavin001,
- highlight-selected - richrace,
- linter-eslint - AtomLinter,
- seti-icons - wyze and
- tokamak-terminal - vertexclique

Themes:

- seti-syntax, seti-ui - jesseweed

<https://www.youtube.com/watch?v=qEDhVKFWoVg>

Welcome back to my guide, helping you create a Discord Bot using the amazing Discord.js library I hope you follow along and create your very own bot!

This is the second part (final) of the handlers episode, I want to apologise for how long it's taken me to get this video edited, rendered and uploaded!

Please let me know what you think to the series, and suggest things for me to cover in future episodes, I have a mental road map I will be following, but I may throw in the odd episode dedicated to requests.

LINK DUMP

You can get the Komada Handler framework [here](#)

All of the tutorial source code is available [here](#)

You will require a free discordapp.com account to use the discord service, sign up [here](#)

You can join the *"An Idiot's Guide Official Server"* [here](#)

If you'd like to support me on Patreon, you can do so [here](#)

Don't forget to join the *"Official Discord.js Server"* [here](#)

Check out the Discord Developers portal [here](#)

Discord.js Official Documentation is available [here](#)

Don't forget, if you want to invite your bots to server, you'll need to get an invite link, you can use the Discord Permissions Calculator [here](#) to set up your permissions, and generate an invite link.

Music in the video:

Local Forecast - Slower by Kevin MacLeod incompetech.com

Licensed under Creative Commons: [By Attribution 3.0 License](#)

List of Atom addons and themes I use;

Addons:

- atom-beautify - Glavin001,
- highlight-selected - richrace,
- linter-eslint - AtomLinter,
- seti-icons - wyze and
- tokamak-terminal - vertexclique

Themes:

- seti-syntax, seti-ui - jesseweed

<https://www.youtube.com/watch?v=1AjBVocSQhM>

Welcome back to my guide, helping you create a Discord Bot using the amazing Discord.js library I hope you follow along and create your very own bot!

In this episode we start going over the moderation actions, we start with warning, mute and channel lockdown!.

Please let me know what you think to the series, and suggest things for me to cover in future episodes, I have a mental road map I will be following, but I may throw in the odd episode dedicated to requests.

LINK DUMP

All of the tutorial source code is available [here](#)

You will require a free discordapp.com account to use the discord service, sign up [here](#)

You can join the "*An Idiot's Guide Official Server*" [here](#)

If you'd like to support me on Patreon, you can do so [here](#)

Don't forget to join the "*Official Discord.js Server*" [here](#)

Check out the Discord Developers portal [here](#)

Discord.js Official Documentation is available [here](#)

Don't forget, if you want to invite your bots to server, you'll need to get an invite link, you can use the Discord Permissions Calculator [here](#) to set up your permissions, and generate an invite link.

Music in the video:

Local Forecast - Slower by Kevin MacLeod incompetech.com

Licensed under Creative Commons: [By Attribution 3.0 License](#)

List of Atom addons and themes I use;

Addons:

- atom-beautify - Glavin001,
- highlight-selected - richrace,
- linter-eslint - AtomLinter,
- seti-icons - wyze and
- tokamak-terminal - vertexclique

Themes:

- seti-syntax, seti-ui - jesseweed

<https://www.youtube.com/watch?v=mmvSIXCQIKo>

Welcome back to my guide, helping you create a Discord Bot using the amazing Discord.js library I hope you follow along and create your very own bot!

Continuing off from the last episode, we go straight into giving people the boot, and the hammer!

Please let me know what you think to the series, and suggest things for me to cover in future episodes, I have a mental road map I will be following, but I may throw in the odd episode dedicated to requests.

LINK DUMP

All of the tutorial source code is available [here](#)

You will require a free discordapp.com account to use the discord service, sign up [here](#)

You can join the "*An Idiot's Guide Official Server*" [here](#)

If you'd like to support me on Patreon, you can do so [here](#)

Don't forget to join the "*Official Discord.js Server*" [here](#)

Check out the Discord Developers portal [here](#)

Discord.js Official Documentation is available [here](#)

Don't forget, if you want to invite your bots to server, you'll need to get an invite link, you can use the Discord Permissions Calculator [here](#) to set up your permissions, and generate an invite link.

Music in the video:

Concentration by Kevin MacLeod incompetech.com

Licensed under Creative Commons: [By Attribution 3.0 License](#)

List of Atom addons and themes I use;

Addons:

- atom-beautify - Glavin001,
- highlight-selected - richrace,
- linter-eslint - AtomLinter,
- seti-icons - wyze and
- tokamak-terminal - vertexclique

Themes:

- seti-syntax, seti-ui - jesseweed

<https://www.youtube.com/watch?v=wb8hBCGApss>

Welcome back to my guide, helping you create a Discord Bot using the amazing Discord.js library I hope you follow along and create your very own bot!

We made some nice embeds for the moderation action logs in the previous episode, but I'm not happy with them, so in this episode after we update the code to v11.1.0 we get to work making those embeds even sexier.

Please let me know what you think to the series, and suggest things for me to cover in future episodes, I have a mental road map I will be following, but I may throw in the odd episode dedicated to requests.

LINK DUMP

All of the tutorial source code is available [here](#)

You will require a free discordapp.com account to use the discord service, sign up [here](#)

You can join the *"An Idiot's Guide Official Server"* [here](#)

If you'd like to support me on Patreon, you can do so [here](#)

Don't forget to join the *"Official Discord.js Server"* [here](#)

Check out the Discord Developers portal [here](#)

Discord.js Official Documentation is available [here](#)

Don't forget, if you want to invite your bots to server, you'll need to get an invite link, you can use the Discord Permissions Calculator [here](#) to set up your permissions, and generate an invite link.

Music in the video:

Concentration by Kevin MacLeod incompetech.com

Licensed under Creative Commons: [By Attribution 3.0 License](#)

List of Atom addons and themes I use;

Addons:

- atom-beautify - Glavin001,
- highlight-selected - richrace,
- linter-eslint - AtomLinter,
- seti-icons - wyze and
- tokamak-terminal - vertexclique

Themes:

- seti-syntax, seti-ui - jesseweed

<https://www.youtube.com/watch?v=W8tgc8I-dQs>

Welcome back to my guide, helping you create a Discord Bot using the amazing Discord.js library I hope you follow along and create your very own bot!

We made some nice embeds for the moderation action logs in the previous episode, but we never did manage to get the case numbers (or mod log numbers) actually working, in this episode we do! as well as fix some major issues I completely glossed over!

Please let me know what you think to the series, and suggest things for me to cover in future episodes, I have a mental road map I will be following, but I may throw in the odd episode dedicated to requests.

LINK DUMP

All of the tutorial source code is available [here](#)

You will require a free discordapp.com account to use the discord service, sign up [here](#)

You can join the *"An Idiot's Guide Official Server"* [here](#)

If you'd like to support me on Patreon, you can do so [here](#)

Don't forget to join the *"Official Discord.js Server"* [here](#)

Check out the Discord Developers portal [here](#)

Discord.js Official Documentation is available [here](#)

Don't forget, if you want to invite your bots to server, you'll need to get an invite link, you can use the Discord Permissions Calculator [here](#) to set up your permissions, and generate an invite link.

Music in the video:

Garden Music by Kevin MacLeod incompetech.com

Licensed under Creative Commons: [By Attribution 3.0 License](#)

List of Atom addons and themes I use;

Addons:

- atom-beautify - Glavin001,
- highlight-selected - richrace,
- linter-eslint - AtomLinter,
- seti-icons - wyze and
- tokamak-terminal - vertexclique

Themes:

- seti-syntax, seti-ui - jesseweed

