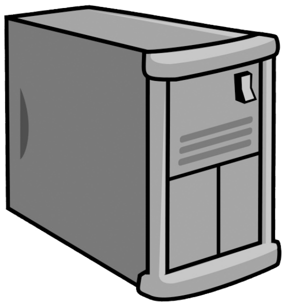
Node.js SERVER SETUP



Building your own production server.

1. Host custom application
   from localhost on your computer.
   (local environment)

2. Host custom application
   on a static domain name.
   (production environment)

Title:	Node.js – Server Setup
Edition:	I – XXXXX XX, 2019
Genre:	Software
Publisher:	Learning Curve Books
Imprint:	Independently published
ISBN:	Not Yet Available
Author:	Greg Sidelnikov (greg.sidelnikov@gmail.com)

Editors, volunteers, contributors: Grace Neufeld.

Primary purpose of **Learning Curve Books** publishing company is
to provide *effective education* for web designers, software engineers and all readers who are interested in being edified in the area of web development.

This edition of **Node.js – Server Setup** was created to speed up the learning process of setting up the JavaScript-based Node server for building your own applications.

For questions and comments about the book you may contact the author or send an email directly to our office at the email address mentioned below.

Learning Curve Books™

Node.js

What Is A Server?

Just like a server at a restaurant serving a meal, a computer server is something that serves files over the network via a dedicated IP address.

For many people server setup and configuration are some of the least exciting parts about being a software developer.

To those who have propensity toward back-end development, it might be the holy grail of how the application actually works.

To front-end developers, it's a black box that generates console errors.

If you want to work for a start-up company as a back-end engineer, having deep knowledge of how servers work is required experience. But even if you are a front-end developer, knowing how to set up a server will expand your skill set and might increase chances of getting that software job you want. It can only be a plus.

In this day and age it is not uncommon for front-end developers to know how to work with servers. You're only gaining if you learn how to set up your own server.

I also think that servers can be exciting for those with entrepreneurial mindset – it gives you the power to build your own custom applications. Nothing is more thrilling than having ability to launch the child of your imagination into the world.

But all deep knowledge starts with getting familiar with the fundamentals.

Node Server

Setting up a Node server can be broken into three parts: 1] **installation**, 2] adding modules and other dependencies using a **manifest file**, and 3] writing the minimum required code in main **index.js** file to launch and run the server.

In this book we will go through each step explaining everything you need to know to get started with 0 knowledge from absolute scratch.

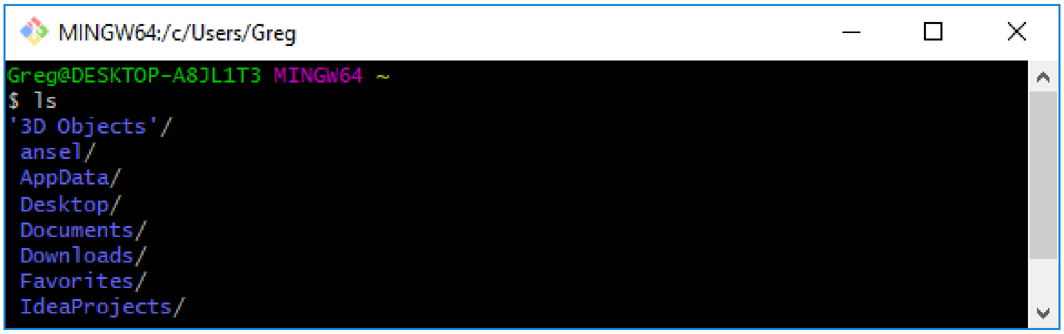
This book contains examples you can simply *copy and paste* into **Terminal** on Mac or **cmd** on Windows, but I encourage you to type them in manually. It's the only way to get better at the **command line** – the skill that makes you a much more productive software developer.

You might need to get a web host to deploy your server online but the process is the same on localhost as it is elsewhere. It's just the hosted solution will additionally route your server's localhost environment to your browser via IP address.

Chapter 1

bash.exe

Did you know that starting with Windows 10 you can work in Linux bash directly from your Windows computer? There is a **bash.exe** you can run:



```
Greg@DESKTOP-A8JL1T3 MINGW64 ~
$ ls
'3D Objects'/
 ansel/
 AppData/
 Desktop/
 Documents/
 Downloads/
 Favorites/
 IdeaProjects/
```

Figure 1.1: Running Linux **ls** (list directory contents) command on Windows.

You can run any Linux command on Windows this way.

In this book we will explore installing Node on Mac, Windows and Linux.

The rest of commands like **npm install package_name** (to install a node package with Node Package Manager) or **node index.js** (to run **index.js** on Node server) are the same on all operating systems.

1.1 Migrating From Apache to Node

Many people are still working with PHP servers. I love Apache and PHP but hate to say that this setup is becoming obsolete. There are good reasons for this. Node servers are used by companies to run libraries like React, which speeds development of the UI components by a large %, but Node is better even for solo developers.

In the past you dispatched **jQuery.ajax** or **fetch** calls to scripts written in **PHP**. This means that just to perform a simple action, you had to know two languages!

Having JavaScript on both front and back-end is magical. Also, Node is generally much faster than an Apache/PHP server. Once you set up a Node server and build your first application, you will never want to go back.

1.2 Node Libraries and Frameworks

It's tempting to just install a library or a framework with one command in Terminal or cmd (on Windows) and watch your application magically appear in your browser at localhost or a dedicated IP address.

But that is neither learning nor gaining experience. When errors creep up, you may not understand what hit you.

In this book, we will walk through the process step by step, without the help of *unnecessary* modules, libraries of frameworks.

If we can only narrow down on what the key issues are and avoid common pitfalls, I think the process of setting up a server can be fun and exciting!

1.3 Running Node Server

There is really one way of running a node server, regardless of whether you are doing it on **localhost** or on a hosted cloud server. If you learn how to set it up on localhost, you will know how to set it up elsewhere. The process is identical.

We'll go through both cases, including the minimum web hosting server setup.

Chapter 2

Installation

There are 3 things you should worry about when installing a Node server:

1. Choosing to install Node **globally** on your Operating System or making it **local** to a project directory on your hard drive. We'll take a look at which way is better.

2. Configuring **package.json** manifest file – it is automatically created by Node installation, but it's nice to know how it actually works, so we'll break it down.

3. Understanding **node_modules** – the default directory where all Node modules are downloaded when you install them from command line.

If you can understand these 3 key things about Node server, you're making a huge step toward getting better at back-end development with JavaScript. So we'll take a look at each one in the remainder of this book.

2.1 Mac

You might want to start learning how to install programs from the command line.

Mac & Linux may have different program installation managers. On OSX you generally use **brew**, **yum** or **port** (See macports.org) command. On Linux it's either **apt-get** or more commonly just **apt**. Your should probably use **apt**.

The reason for so many ways to install packages on Linux-based OS'es is due to many open-source distributions (Debian, Ubuntu, Fedora, Arch, CentOS...etc.,) each in own unique flavor and version history.

Technically, you can also use **apt** on Mac OSX, if it's available. (It should be.)

If you already have **apt** but you want **brew** – which some find to be better – on your OSX, run **apt install linuxbrew-wrapper**

Now you can use **brew** to install packages such as **nodejs**, for example:

Just run **brew install nodejs** to install **Node** in Terminal on OSX.

To do the same on Linux use **apt-get install nodejs** in bash.

Or simply **apt install nodejs** in bash. There are minor differences between **apt** and **apt-get**, you can use either one. But **apt** is considered better.

You can also try **apt install nodejs** in bash. And this is my preferred way.

2.1.1 apt install nodejs

On Linux-based systems these commands install a package or program locally on your computer, much like running a GUI-based installer on Windows.

All that happens is locating, downloading and copying program files to a directory on your hard drive.

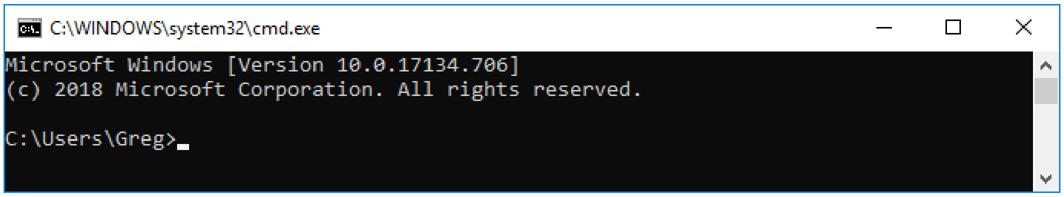
After installation of packages such as **nodejs** and **mysql** you usually have to configure and run them. We'll go over the process in this book!

2.2 Windows

To get started on Windows, hold Window-key and press R.

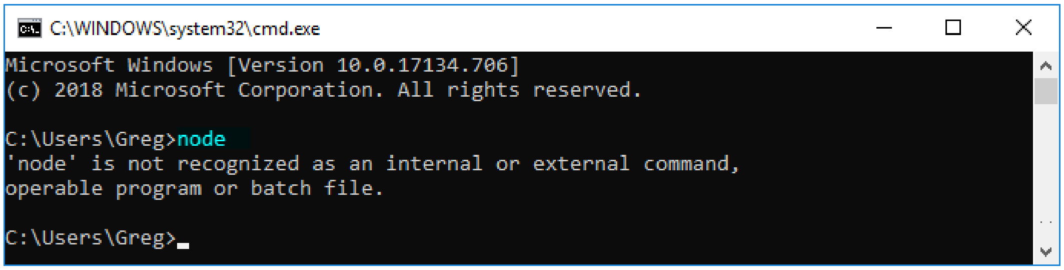
Type **cmd** and hit Enter in the box that appears.

The command line window will open, and it looks similar to this:



```
Microsoft Windows [Version 10.0.17134.706]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Greg>
```

To see if node is already installed, type **node** and hit Enter:

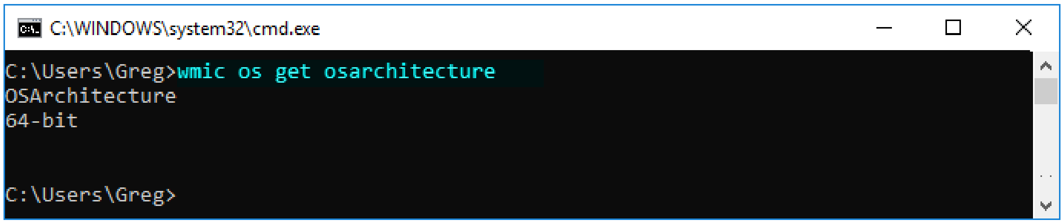


```
Microsoft Windows [Version 10.0.17134.706]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Greg>node
'node' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Greg>
```

In this case I don't have node installed, so let's get started!

2.3 Installing Node on Windows

Some older laptops or PCs might still be 32 bit. So, if you're not sure, run the following command in the **cmd** window: **wmic os get osarchitecture**

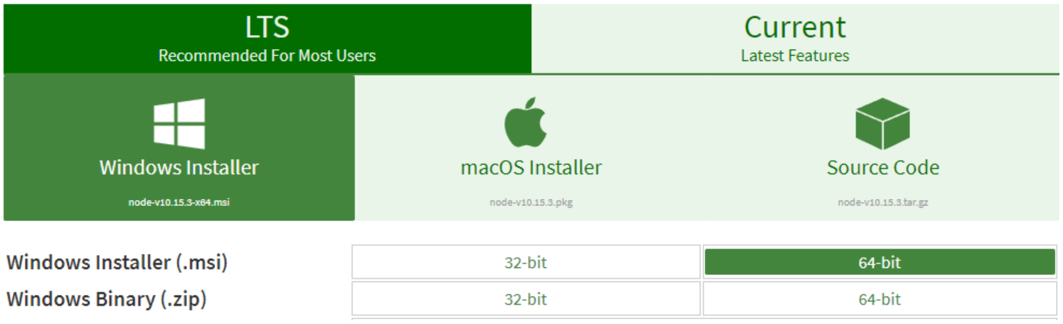


```
C:\WINDOWS\system32\cmd.exe
C:\Users\Greg>wmic os get osarchitecture
OSArchitecture
64-bit

C:\Users\Greg>
```

Figure 2.1: **Running:** wmic os get osarchitecture

Looks like my computer is 64-bit, just as I thought!

To install Node on Windows, first head over to the official Node website:



Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.zip)	32-bit	64-bit

Figure 2.2: **URL:** https://nodejs.org/en/download/

Just choose Window Installer and 32-bit or 64-bit version. I chose 64-bit one, because my computer is 64-bit. Yours is too, most likely.

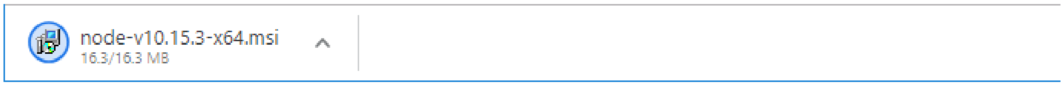


Figure 2.3: Click on the downloaded file (I am using Chrome browser here.)

This will begin the Windows installation process. Just click **Next** everywhere.

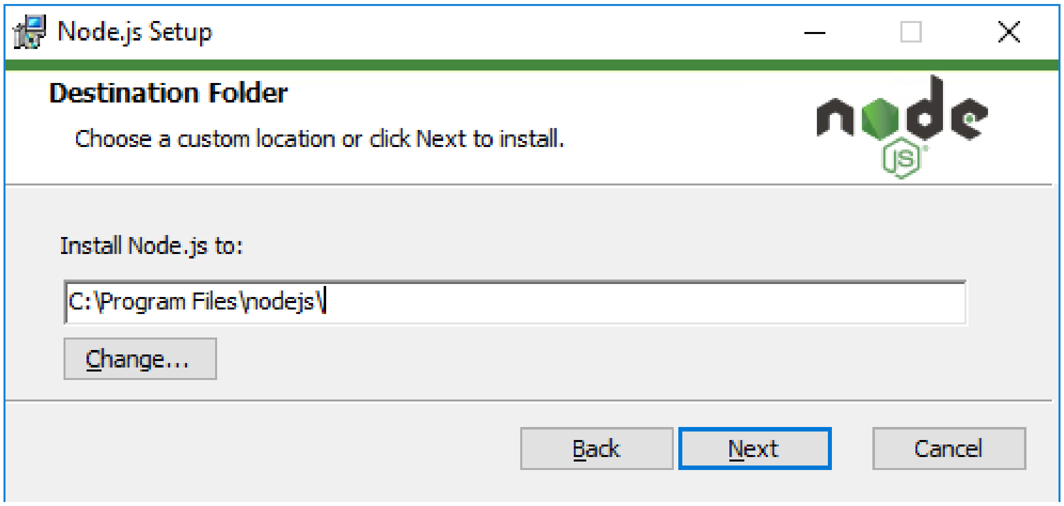
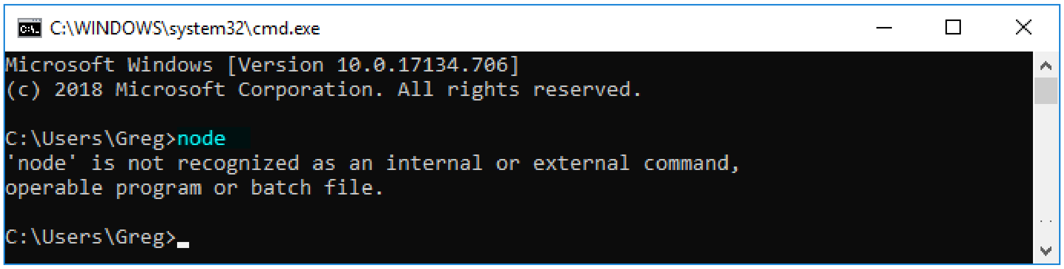


Figure 2.4: I chose default path **C:\Program Files\nodejs**

2.4 Configuring Node Server Globally

Sometimes you want to be able to call **node** command from any location on your hard drive. Many developers do this because they want to run **node** command from anywhere on their hard drive. Just like I did in the previous example.

So now that we installed Node with the Windows installer, we should be able to run **node** command from command line, right?



```
Microsoft Windows [Version 10.0.17134.706]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Greg>node
'node' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Greg>_
```

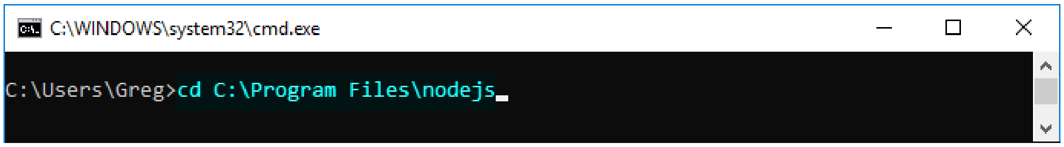
Wrong.

We still can't launch node globally.

In order to run the **node** command, you have to be in the directory in which Node was installed, because that's where **node.exe** executable file is.

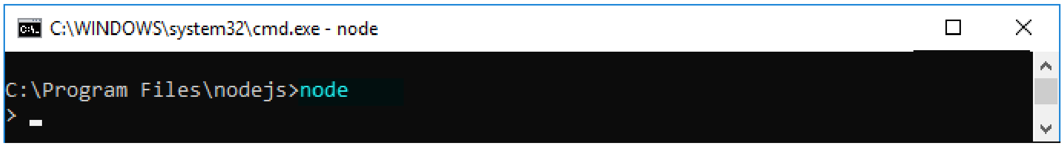
Earlier we installed node into **C:\Program Files\nodejs**

Navigate to that directory by typing: **cd C:\Program Files\nodejs**



```
C:\Users\Greg>cd C:\Program Files\nodejs_
```

Now that we're there, execute **node** command:

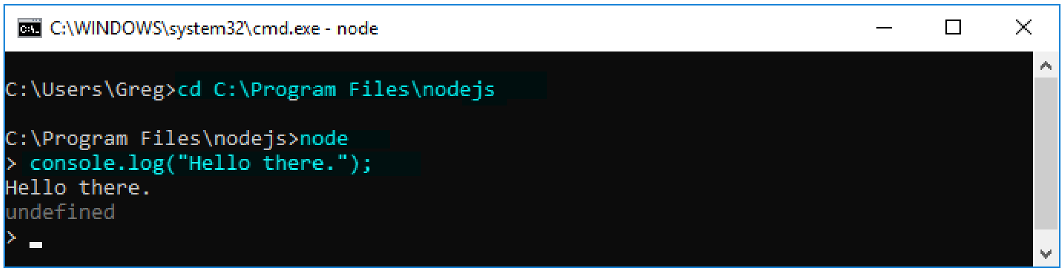


```
C:\Program Files\nodejs>node
> _
```

Your cursor will turn to right bracket >_

This means node server is running and awaiting your JavaScript code!

Type console.log("Hello there."), to see that Node server works (almost) just like your regular JavaScript console in Chrome. That's because Node server runs on the same JavaScript engine (V8) used in Chrome (with just a few minor distinctions.)



```
C:\Users\Greg>cd C:\Program Files\nodejs

C:\Program Files\nodejs>node
> console.log("Hello there.");
Hello there.
undefined
> _
```

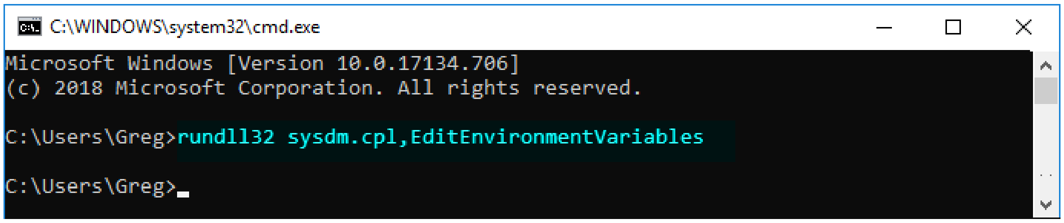
This means that we can launch node and execute JavaScript code. You can feed node an **index.js** file to execute, instead of typing the code live in the **cmd** program. We'll take a look at that in just a moment.

2.5 Enable Node To Run From Any Directory

In previous section we ran some JavaScript code on an instance of a node server.

But if you create new projects, it's likely they will be located in different folders like **C:\develop\my_app** and as we had just seen we cannot run node outside of its installation folder by default.

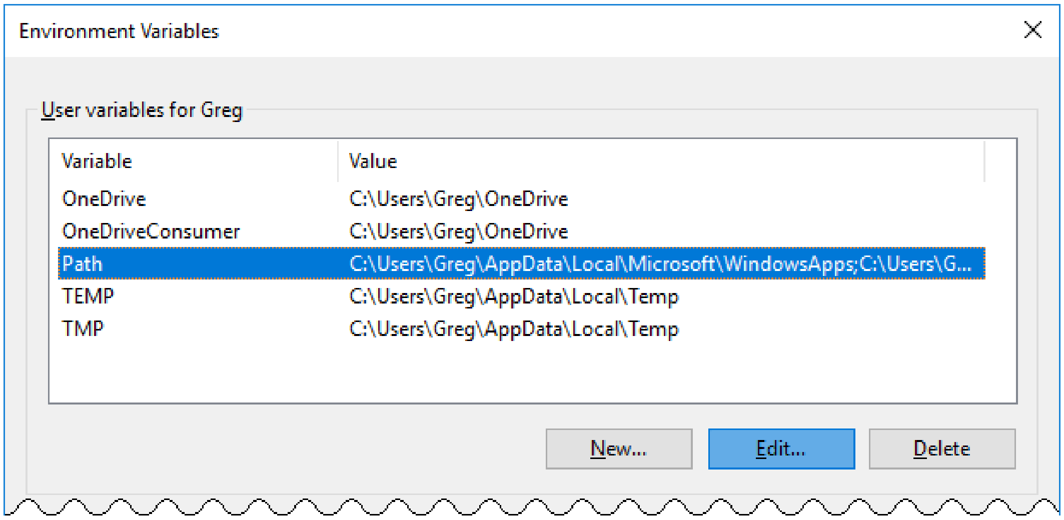
The solution is to include path to **node.exe** to your **Environment Variables**.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.17134.706]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Greg>rundll32 sysdm.cpl,EditEnvironmentVariables

C:\Users\Greg>_
```

Figure 2.5: **Running:** rundll32 sysdm.cpl,EditEnvironmentVariables

Running command **rundll32 sysdm.cpl,EditEnvironmentVariables** will crack open Windows system variable editor:



Environment Variables

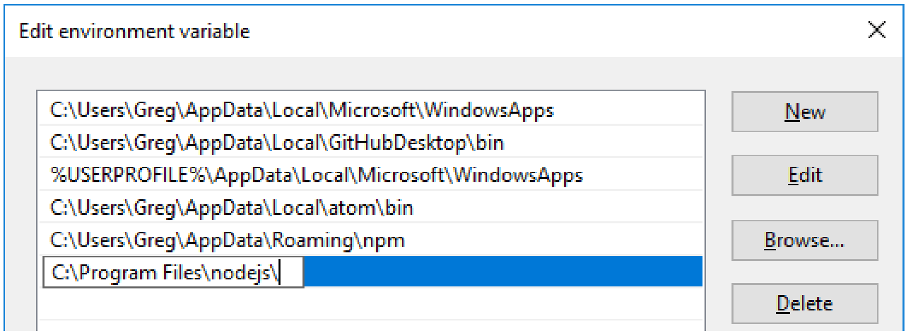
User variables for Greg

Variable	Value
OneDrive	C:\Users\Greg\OneDrive
OneDriveConsumer	C:\Users\Greg\OneDrive
Path	C:\Users\Greg\AppData\Local\Microsoft\WindowsApps;C:\Users\G...
TEMP	C:\Users\Greg\AppData\Local\Temp
TMP	C:\Users\Greg\AppData\Local\Temp

Figure 2.6: Upper half of **Environment Variables** window.

In the *upper* box of **Environment Variables** window (where it says user variables for **YourUserName**) click on **Path** and click **Edit...** button.

In next window, click **New** and type **C:\Program Files\nodejs** on a new line:



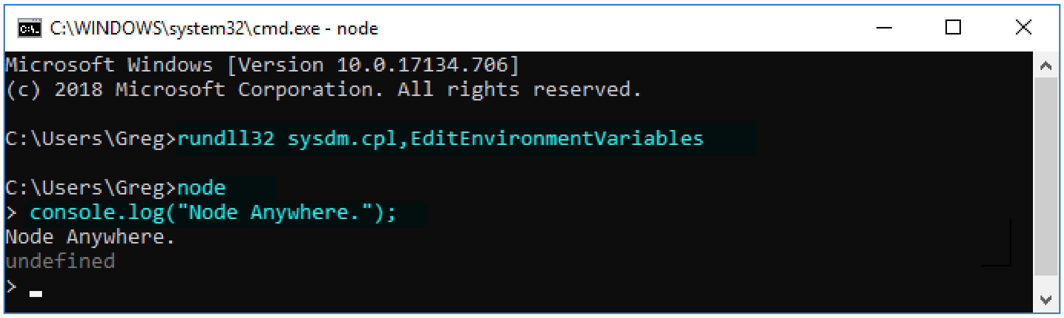
Edit environment variable

```
C:\Users\Greg\AppData\Local\Microsoft\WindowsApps
C:\Users\Greg\AppData\Local\GitHubDesktop\bin
%USERPROFILE%\AppData\Local\Microsoft\WindowsApps
C:\Users\Greg\AppData\Local\atom\bin
C:\Users\Greg\AppData\Roaming\npm
C:\Program Files\nodejs\
```

Figure 2.7: Upper half of **Environment Variables** window.

Click **OK**, then click **OK** again on underlying window.

That's it! Now we can run **node** from anywhere on the hard drive:



C:\WINDOWS\system32\cmd.exe - node

```
Microsoft Windows [Version 10.0.17134.706]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Greg>rundll32 sysdm.cpl,EditEnvironmentVariables

C:\Users\Greg>node
> console.log("Node Anywhere.");
Node Anywhere.
undefined
> _
```

Figure 2.8: Now you can run node from anywhere.

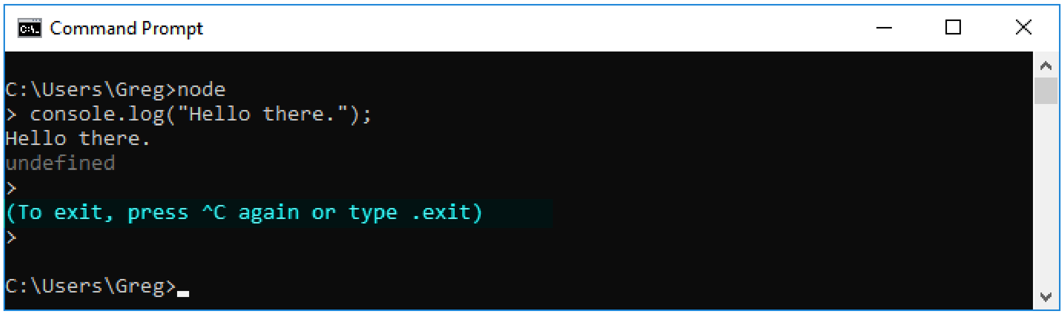
Type **node** then type **console.log("Node Anywhere.")** and watch output.

The function **console.log** returns **undefined** because it is not an expression. If you want to learn more about JavaScript, check out JavaScript Grammar (**www.javascriptgrammar.com**) for a free copy.

2.6 Exit from Node

After entering some JavaScript, we're still stuck in Node.

To exit, hold **Ctrl** and press **C** key two times in a row.



```
C:\Users\Greg>node
> console.log("Hello there.");
Hello there.
undefined
>
(To exit, press ^C again or type .exit)
>

C:\Users\Greg>_
```

2.7 Congratulations!

You have just installed Node on your computer.

Chapter 3

Running Applications

So far we only installed a dormant **node.exe** on the hard drive.

But how do we actually run our own application that serves **index.html** or any other file on the system when the browser makes a request to it?

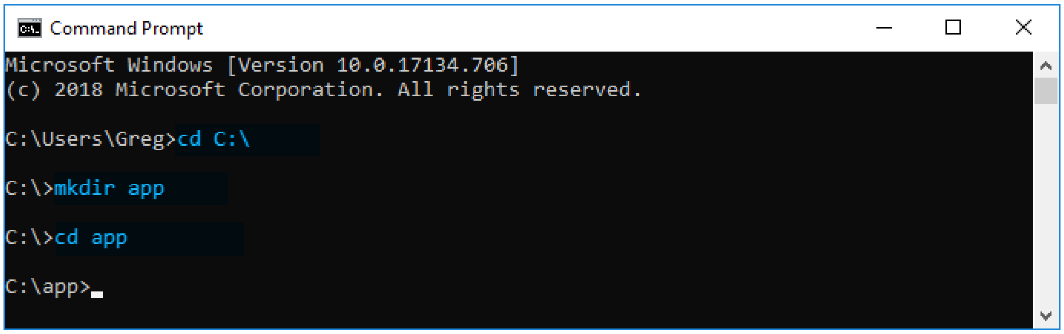
We'll get to that in just a moment. But first, let's set up our **project directory**. It can be located anywhere on your computer – choose wisely.

3.1 Project Home Directory

This is my favorite part. We're going to choose where our Node application is going to live on the hard drive.

I know we installed Node globally on the Operating System, but the project will usually be located in a local folder, like **C:\app**

Let's navigate to **C:\** drive using **cd C:\** command:



```
Microsoft Windows [Version 10.0.17134.706]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Greg>cd C:\

C:\>mkdir app

C:\>cd app

C:\app>_
```

Then, use **mkdir app** to create new directory **C:\app** and enter it with **cd app**.

3.2 Running a file via Node

Instead of typing all the code on the command line, we can pass a JavaScript file for node to execute. Let's create our first file containing some basic JavaScript.

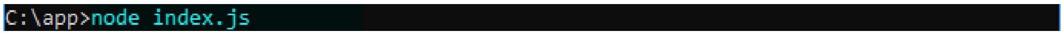
Create new file **index.js** as in **C:\app\index.js** and type following in it:

```
console.log(process); // Output native Node process object
```

You don't have to import or require any modules to use **process** object.

It is available in the module scope of your application by default.

Let's run our **index.js** file via node to see what happens. Make sure you are in **C:\app** and type the following command in **cmd**:



```
C:\app>node index.js
```

Figure 3.1: **Run:** node index.js

node index.js *must* be executed from same folder where you have **index.js** file.

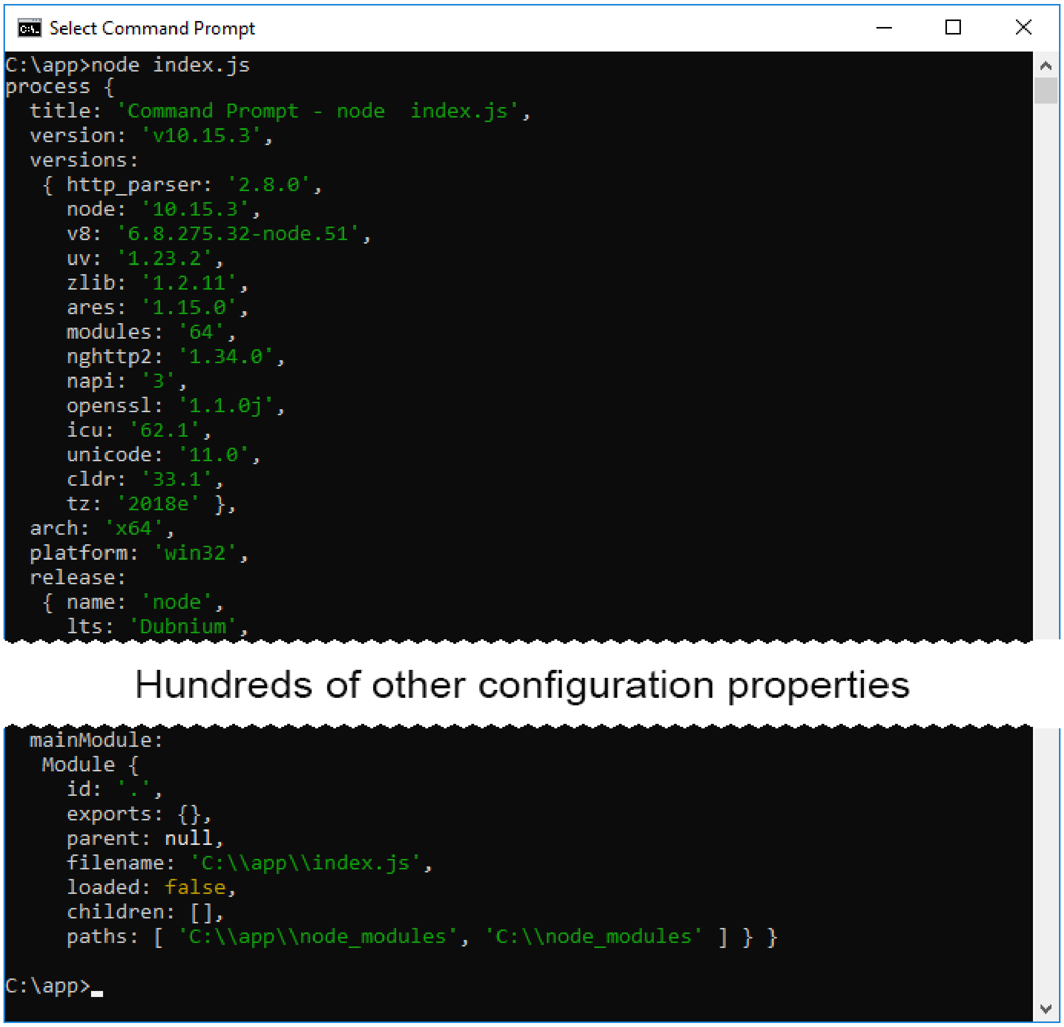
The only reason we can run **node** (actually **node.exe**) from this folder is because we added its installation path to our environment variables in an earlier step. So if you get an error, make sure to complete that step first!

This runs the single line of code **console.log(process)** from our **index.js** script.

See the output on the next page.

3.3 The process Object

In our **index.js console.log(process)** outputs the contents of **process** object:



```
C:\app>node index.js
process {
  title: 'Command Prompt - node  index.js',
  version: 'v10.15.3',
  versions:
   { http_parser: '2.8.0',
     node: '10.15.3',
     v8: '6.8.275.32-node.51',
     uv: '1.23.2',
     zlib: '1.2.11',
     ares: '1.15.0',
     modules: '64',
     nghttp2: '1.34.0',
     napi: '3',
     openssl: '1.1.0j',
     icu: '62.1',
     unicode: '11.0',
     cldr: '33.1',
     tz: '2018e' },
  arch: 'x64',
  platform: 'win32',
  release:
   { name: 'node',
     lts: 'Dubnium',
```

Hundreds of other configuration properties

```
  mainModule:
   Module {
     id: '.',
     exports: {},
     parent: null,
     filename: 'C:\\app\\index.js',
     loaded: false,
     children: [],
     paths: [ 'C:\\app\\node_modules', 'C:\\node_modules' ] } }

C:\app>_
```

Figure 3.2: Printing out contents of **process** object.

This Node **process** object contains all kinds of useful configuration properties.

For example **process.versions.v8** tells you V8 engine version it's running on.

3.4 Adding Packages To index.js

Node comes with about 349 packages by default. They are also called modules.

Modules are stored in your node installation folder (**C:\Program Files\nodejs** in our case) under **node_modules**.

You might also find them in **nodejs\npm** folder:
C:\Program Files\nodejs\node_modules\npm\node_modules\

In browser-side JavaScript code, you can use **import / export** keywords. But in addition to that, in node you can use **require** keyword to add useful modules to your app from the main NPM repository.

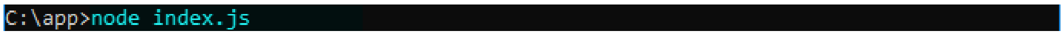
Node assumes **path** module exists in **node_modules** directory. If not, you can install it with **npm** (*node package manager*) by running: **npm install path**

The package called **path** gives us ability to work with URL paths in an easier way.

Once included **path.extname** method can parse extension part of a filename from a string. This is a frequent need in server-side programming:

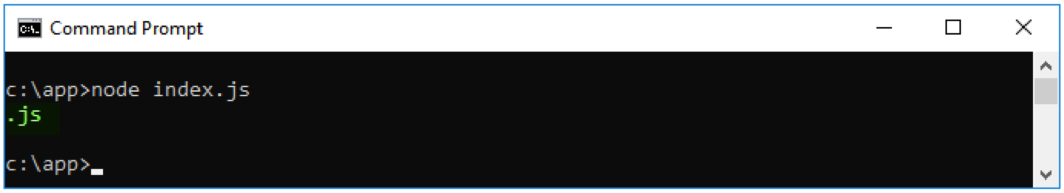
```
let path = require('path');

let ext = path.extname('index.js');

console.log(ext); // outputs '.js'
```

Update **index.js** file with above code and run **node index.js** command:



```
C:\app>node index.js
```

Figure 3.3: **Run:** node index.js



```
c:\app>node index.js
.js

c:\app>
```

Figure 3.4: **Output:** .js

3.5 Next Steps

We've just experimented with running JavaScript files with node, imported a package, and ran a method. You can run any valid JavaScript file this way.

But there is still a problem. After executing **index.js** the **node** process quits, and control is given back to **cmd**. This won't help us run our application server. We need to build something that **runs continuously** so we can serve a file every time someone will access our server from their browser.

3.6 Running Application Server Continuously

To continuously listen for connections we need to write server code in **index.js**. But don't worry, the bare bones version is not as difficult as it may sound.

Adding **let http = require("http")** to the very top of **index.js** is the first step.

When you assign a variable to a package name this way, we make **http** variable point to the default object defined in that package. Most packages export a single object this way, with all the properties and methods already attached to it, so we can start using them, without worrying how they were implemented!

Likewise, the **http** object can be used to start listening for requests.

Calling **http.createServer** makes our node program **enter listening state**. How this actually works internally is tied to how your Operating System implements socket connections. But to get started quick we don't need to think about that.

```
let http = require('http');

const ip = '127.0.0.1'; // localhost
const port = 3000;

http.createServer(function(request, response) {
  console.log('request ', request.url);
}).listen(port, ip);

console.log('Running at http://' + ip + ':' + port + '/');
```

Figure 3.5: The bare-bones Node server.

Copy code above into **index.js** and let's run it via **node index.js** command:

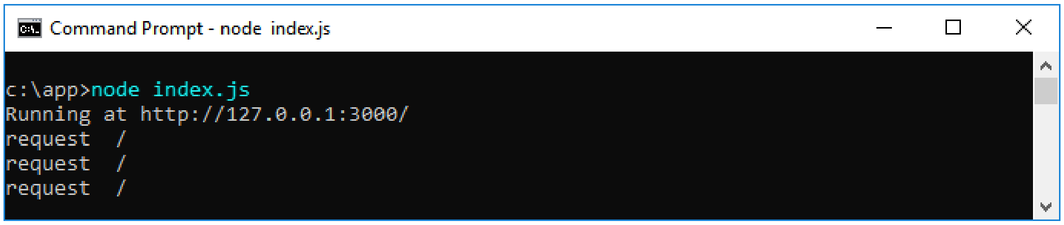
```
C:\app>node index.js
```

Note that we used **127.0.0.1** (It is the same as **localhost**) with port **3000**.

3.7 Congratulations!

There is now a back-end server running from your computer.

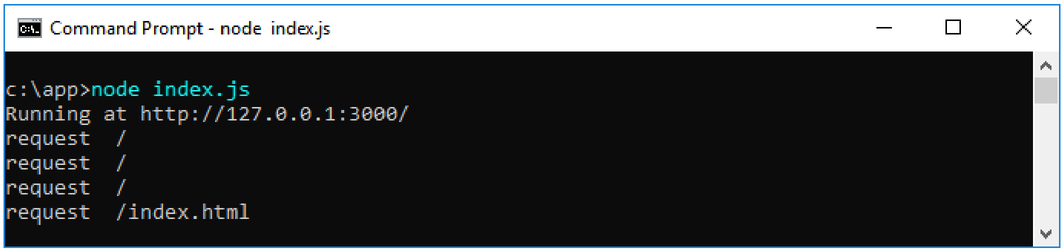
Still don't believe me? Keep **cmd** window open and paste **http://127.0.0.1:3000** into your browser's Address Bar. I just did that 3 times and here is the server receiving the requests:



```
Command Prompt - node index.js

c:\app>node index.js
Running at http://127.0.0.1:3000/
request  /
request  /
request  /
```

The request shows up as / because we're accessing just the IP address. It maps to the root directory of our application.

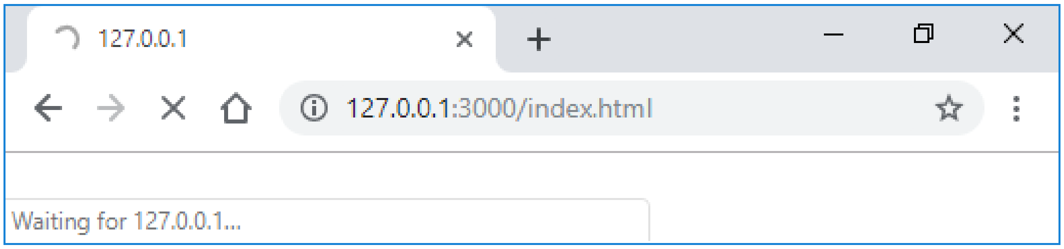
Now enter **http://127.0.0.1:3000/index.html** and watch **cmd** window:



```
Command Prompt - node index.js

c:\app>node index.js
Running at http://127.0.0.1:3000/
request  /
request  /
request  /
request  /index.html
```

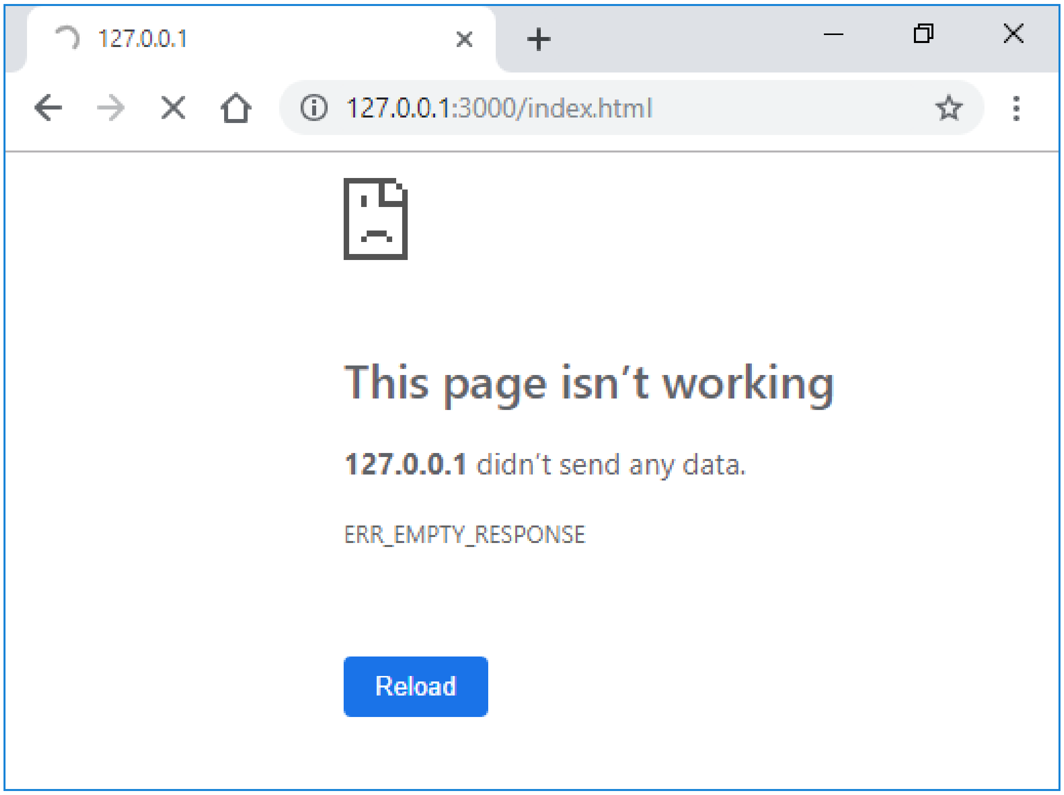
The file **index.html** doesn't even exist yet. All we do is intercept incoming requests and print them out.

Because we are not sending anything at all back to the browser, you will be stuck on this page **"waiting for 127.0.0.1..."**:



Now you know what happens when you try to go to a website and that little note in the lower left corner appears!

The bar is spinning and it's only a matter of time until you get the following:

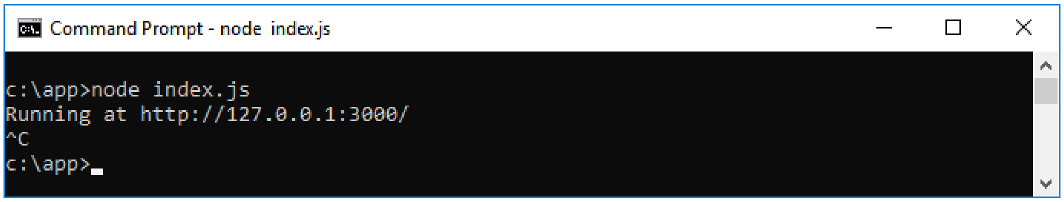


This page isn't working

127.0.0.1 didn't send any data.

ERR_EMPTY_RESPONSE

Request timed out. But you can terminate the server yourself with **Ctrl + C**:



```
c:\app>node index.js
Running at http://127.0.0.1:3000/
^C
c:\app>
```

Every time you want to terminate the server manually use this combo. This can become useful if your server is stuck in an infinite loop or processing a long script.

3.8 Serving Files Continuously

We've just set up our Node server to listen to URL requests.

To actually serve the requested file, we need to write additional code.

```
let http = require('http'); // gives us http
let fs = require('fs');      // gives us fs
let path = require('path'); // gives us "string".pathext()

const ip = '127.0.0.1';      // or domain IP
const port = 3000;           // or 80

// Create server using http module, and wait for response:
http.createServer(function(request, response) {

  // Print request object, just for fun
  console.log('request ', request);
  // Add . to URL to convert it to local file path
  let file = '.' + request.url;
  // Redirect / to serve index.html
  if (file == './') file = './index.html';
  // Extract requested file’s extension
  let extension = String(path.extname(file)).toLowerCase();
  // Define acceptable file extensions
  let mime = { '.html': 'text/html' }
  // If requested file type is not in mime, default
  // to octet-stream which means “arbitrary binary data.”
  let type = mime[extension] || 'application/octet-stream';

  // Read the file from the hard drive
  fs.readFile(file, function(error, content) {
    if (error) {
      if (error.code == 'ENOENT') {
        fs.readFile('./404.html', function(error, content) {
          response.writeHead(200, {'Content-Type': type});
          response.end(content, 'utf-8');
        });
      } else {
        response.writeHead(500);
        response.end('Error: ' + error.code + '\n');
        response.end();
      }
    } else {
      response.writeHead(200,{'Content-Type': type});
      response.end(content, 'utf-8');
    }
  });

}).listen(port, ip);

// Display server is running message
console.log('Running at ' + ip + ':' + port + '/');
```

Update **index.js** with code from above and request **127.0.0.1:3000/index.html** or **localhost:3000/index.html** from your browser again.

If **index.html** exists in the root directory of your project (**C:\app**) then it will be served to your browser and its contents will be displayed.

But if the requested **index.html** doesn't exist the browser will serve an empty page by inserting blank **<html></html>** tags on its own for a clean slate view.

Adding More MIME Types

We now have our Node serving files of **.html** type. If any other type is requested, the file will be actually downloaded into the browser, instead of shown in the display view.

This is because if the file extension is not on our mime list, we serve it as **application/octet-stream** which means a binary file download.

In order to add other files (I assume you want at least support for **.css**, **.png**, **.jpg**, **.txt**, **.js** and **.json** files to start doing anything meaningful with your server) all you have to do is add them to **mime** variable:

```
let mime = { '.html': 'text/html',
               '.js': 'text/javascript',
              '.css': 'text/css',
             '.json': 'application/json',
              '.png': 'image/png',
              '.jpg': 'image/jpg',
              '.gif': 'image/gif' };
```

Conclusion

From here on you will build out the rest of your server architecture. The remainder of this book will walk you through how to install MySQL and write code for implementing your own API. You can then use this API to build your application.

Chapter 4

Building The API

So far we learned how to serve requested files. But if you want to build real web applications, you might want to integrate support for API endpoints.

An endpoint is a URL that follows a special pattern. We can intercept this pattern and upon its presence we will execute an API command, instead of serving a file.

This API command is likely to establish a database connection and either modify or fetch some type of data. Our server can return a JSON object containing requested data.

We will take a look at how to create some API end points for our existing Node server. And then we will install MySQL and execute some queries. Using this pattern you can build out the rest of your API.

Here is an example of an API endpoint that **gets** user info:

```
http://localhost:3000/api/user/get
```

Here is an example of an API endpoint that **sets** user info:

```
http://localhost:3000/api/user/set
```

When this type of URL is requested, we will cancel serving the file, run some server-side commands and return a result.

4.1 Building Endpoint API

Let's write our API code.

Place the following code into a separate file **api.js**. It will be imported into our main server application from **index.js**.

```
class API {
    constructor() { }
    static exec() {
        let parts = API.parts;
        console.log("API.exec(), parts = ", API.parts);
        if (parts[0] == "api") {
            if (parts[1] == "user") {
                if (parts[2] == "get") {
                    // database.exec();
                }
            }
        }
    }
    static catchAPIrequest(v) {
        v[0] == "/" ? v = v.substring(1, v.length) : null;
        if (v.constructor === String)
            if (v.split("/")[0] == "api") {
                API.parts = v.split("/");
                return true;
            }
        return false;
    }
}

API.parts = null;

module.exports = API;
```

Static function **API.catchAPIrequest** will return **true** if the **request.url** matches our API pattern. We will cancel serving the file as usual in this case.

The static **API.parts** property stores the parts of the API call in an array. For example **localhost:3000/api/user/get** becomes: **["api", "user", "get"]**. Based on these values we can tell our server what to do (execute MySQL query, etc.)

Static function **API.exec** will execute the command.

4.1.1 Integrating API class into our server

Now that we have the **API** class ready, let's add API functionality to **index.js**. The changes are minor, they are highlighted below.

Add the following line to the top of **index.js** on line **004**:

```
001 let http = require('http');
002 let fs = require('fs');
003 let path = require('path');
004 let API = require('./api.js');
```

This makes our API class available for use in **index.js**.

Now go to line **026-027**'ish in our current **index.js** file and add lines in green:

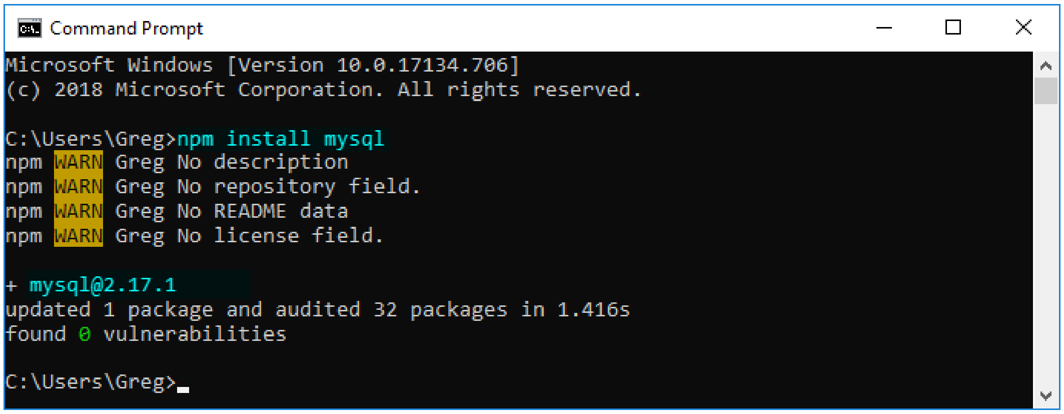
```
026 fs.readFile(filename, function(error, content) {
027   if (error) {
028     if (error.code == 'ENOENT') {
029       // Is this an API call, or should we serve a file?
030       if (API.catchAPIrequest(request.url))
031         response.end(API.exec(request.url), 'utf-8');
032       else
033         // Not an API call - file just doesn't exist
034         fs.readFile('./404.html', function(error, content) {
035         response.writeHead(200,{'Content-Type':contentType});
036         response.end(content, 'utf-8');
037         });
038     } else {
039       response.writeHead(500);
040       response.end('Error: ' + error.code + ' ..\n');
041       response.end();
042     }
043   } else {
044     console.log("API request detecting...");
045     response.writeHead(200, {'Content-Type':contentType});
046     response.end(content, 'utf-8');
047   }
048 });
```

We used boolean result of **API.catchAPIrequest** to branch out.

Our server can now draw a distinction between an API call and a file request.

4.2 Install NPM MySQL Module

Before we move on, make sure to install **npm** module **mysql** by executing the following command in your **cmd** or **Windows Terminal** or **bash.exe** on Windows (Yes, Windows 10+ has its own Linux-like bash.exe) or **Terminal** (OSX/Linux):


Command Prompt

```
Microsoft Windows [Version 10.0.17134.706]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Greg>npm install mysql
npm WARN Greg No description
npm WARN Greg No repository field.
npm WARN Greg No README data
npm WARN Greg No license field.

+ mysql@2.17.1
updated 1 package and audited 32 packages in 1.416s
found 0 vulnerabilities

C:\Users\Greg>_
```

Figure 4.1: Install MySQL by executing **npm install mysql** on **cmd**.

I already have **mysql** installed, so your output might be a bit different.

You might have to type **Y** and hit **Enter** a couple times during **mysql** installation.

```
let mysql = require('mysql');
```

This gives us ability to include **mysql** as a module in our server-side **.js** scripts:

4.2.1 Installing MySQL Server

In addition to this, you must have MySQL server running on your system or at some remote location (such as your web host's IP address: XXX.XXX.XX.XX). If you've already done that step, you can skip the next section.

If not, the next section will demonstrate how to set up your own database running either on your localhost or at some domain name.

Chapter 5

Setting Up MySQL Server

You can run a MySQL server directly from your computer on **localhost** – the home address of your computer as part of your local development environment. Or you can run it on a remote server (a paid web host, for example.)

5.0.1 Install MySQL Server Locally on Windows

Remember, you have to install MySQL – the actual program to run on your computer. But you also have to install the accompanying **mysql** NPM module so your Node server can include it using **require** keyword.

If you want to install MySQL on your computer as part of your local development environment, one of the fastest ways to do that is to download it from **https://dev.mysql.com/downloads/installer/**

Or you can install MySQL together with the **Ampps** server. You can download it from **https://www.ampps.com/** Just download it, go through the installer and in under a few minutes you will have MySQL running.

I won't go into great detail on pressing **Next** button on installation screen here. The process is pretty straightforward.

5.0.2 Install MySQL Server on Ubuntu

The **apt-get** (or simply **apt**) command is perhaps the most popular, because **apt** also identifies *dependencies*, downloads and installs them, whereas **dpkg** does not.

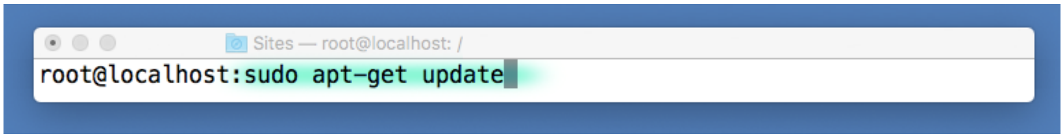
This section will walk you through the standard MySQL server configuration on a **Ubuntu** server, for no particular reason. It's just a personal favorite.

In the following images, I already used my Mac Terminal app to log into my hosted web server in the cloud. But if you are running Ubuntu on your computer, the steps are exactly the same, except you don't have to log in to your web host.

To log in to your web host open Terminal on your Mac and execute the command **ssh root@XXX.XX.XX.XX** and replace the X'es with your actual remote server address. Enter password, and you're in!

Once there, you can start executing the following commands.

Make sure all packages we will install are up to date:

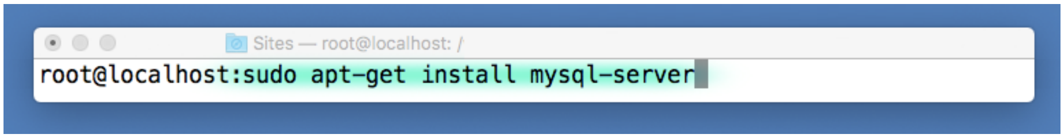


```
root@localhost:sudo apt-get update
```

Figure 5.1: **sudo apt-get update**

This will have your Linux system updated with fresh URLs to latest packages. So when you install **mysql** in the next step most recent update will be used.

Install the MySQL server on your system:



```
root@localhost:sudo apt-get install mysql-server
```

Figure 5.2: **sudo apt-get install mysql-server**

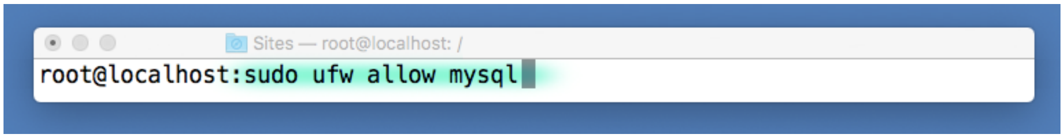
Installation will begin and files will start downloading to your hard drive.

At this point you will see a stream of commands flowing on your screen.

Type **Y** / **yes** and press **Enter** if you are ever asked any questions.

Allow incoming and outgoing traffic to/from mysql program:

Assuming **mysql** installation was successful, you can now use the **ufw** utility program (Which stands for **Uncomplicated Firewall**.)



```
root@localhost:sudo ufw allow mysql
```

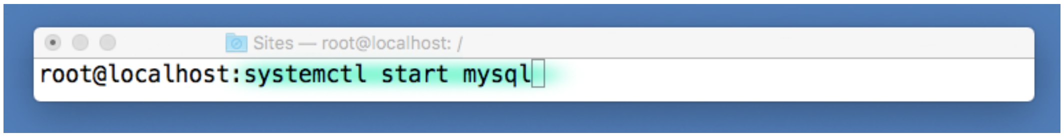
Figure 5.3: **sudo ufw allow mysql**

Note, our command starts with **sudo** which simply runs any program (**ufw** in this case) using security privileges of another user.

sudo originally meant **superuser do** because on older Linux versions it was designed to run only as *superuser* (usually **root**). Nowadays, you can use it to execute commands using any username/password pair on the system. The meaning of **sudo** acronym was gracefully re-formed **substitute user do** for said reason.

Uncomplicated Firewall

You can use **ufw allow** and **ufw disallow** commands to allow incoming and outgoing traffic from a program. Without any additional options it is assumed you are giving full access rights to this program. The **ufw** command has many other options, and you can even allow connections on a specific port. Look it up online if you need the complete documentation.



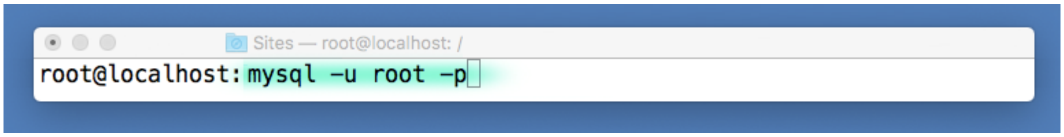
```
root@localhost:systemctl start mysql
```

Figure 5.4: **systemctl start mysql** (start mysql process)

If you run just the **mysql** command now, you will enter mysql prompt.

But... it's important to use **mysql -u root -p** instead.

This logs you in as root user, which gives you special privileges.

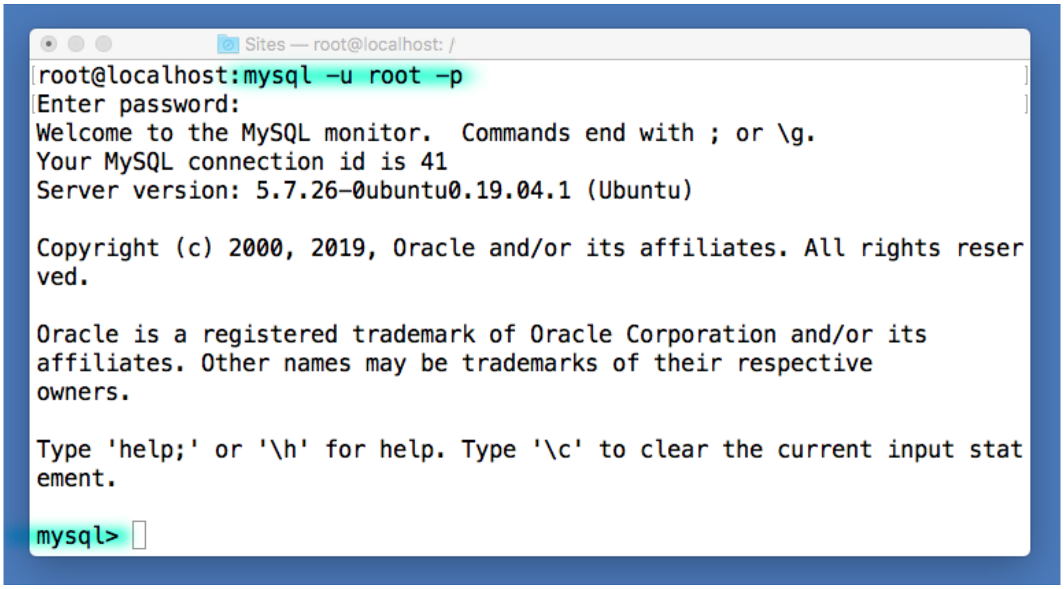


```
root@localhost:mysql -u root -p
```

Figure 5.5: **mysql -u root -p**

You will be prompted for your MySQL root user password. Enter it!

Then you should be greeted with **mysql** >:



```
root@localhost:mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 41
Server version: 5.7.26-0ubuntu0.19.04.1 (Ubuntu)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reser
ved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input stat
ement.

mysql>
```

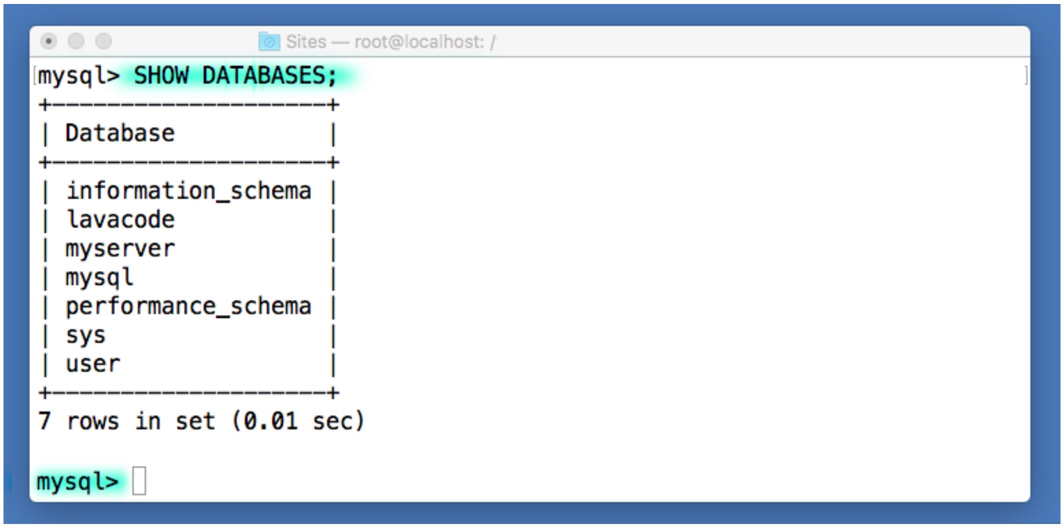
Figure 5.6: **mysql -u root -p**

Welcome to mysql prompt. This means your MySQL server is running and it's ready to receive MySQL commands. Every standard MySQL command you can execute via code will work in this prompt as well.

5.0.3 Show Existing Databases

When mysql was installed it automatically created **information_schema** – the first ever database added to your MySQL server. It contains additional information about all your databases. You can query it, but you cannot change anything on it.

You can type **SHOW DATABASES;** to see all databases that currently exist on your MySQL server. On my machine, I already created **lavacode** and **myserver**:



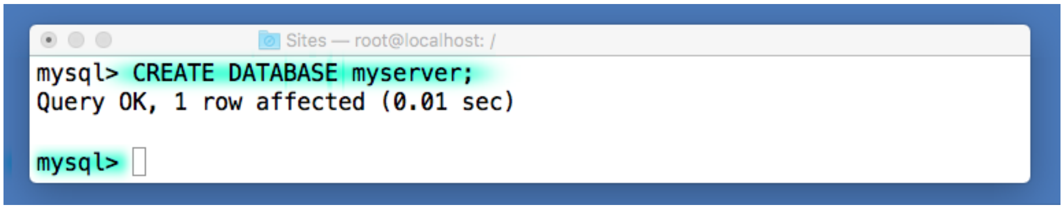
```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| lavacode           |
| myserver           |
| mysql              |
| performance_schema |
| sys                |
| user               |
+--------------------+
7 rows in set (0.01 sec)

mysql> 
```

Figure 5.7: **SHOW DATABASES;**

5.0.4 Creating A New Database

To create your own database, run **CREATE DATABASE database_name;**

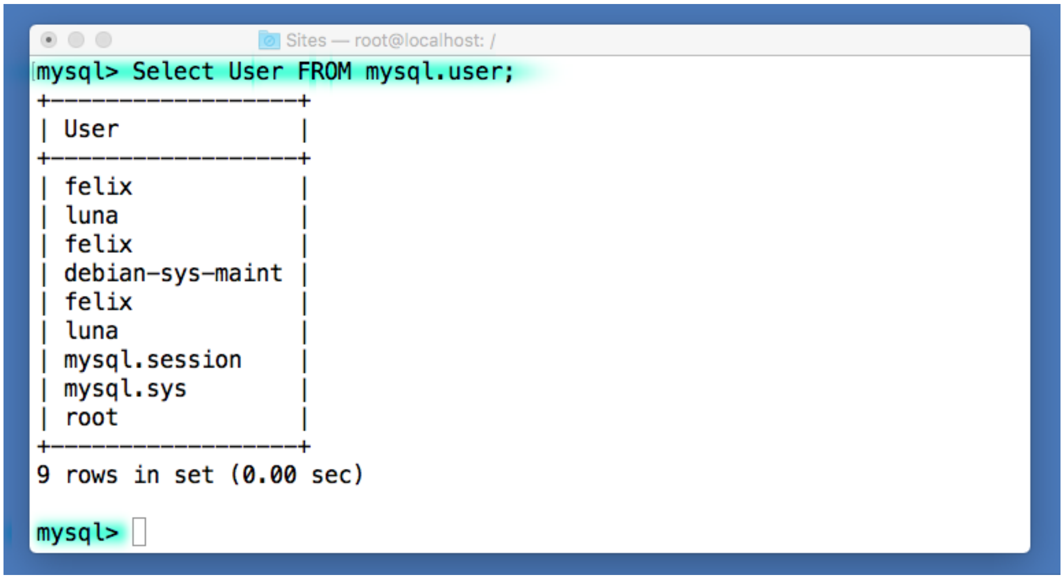
(Don't forget to end your statement with **;** or the mysql prompt will drop to the next line without executing the command.)



```
mysql> CREATE DATABASE myserver;
Query OK, 1 row affected (0.01 sec)

mysql> 
```

Figure 5.8: **CREATE DATABASE myserver;**

You can also check current users by running **Select User FROM mysql.user;**



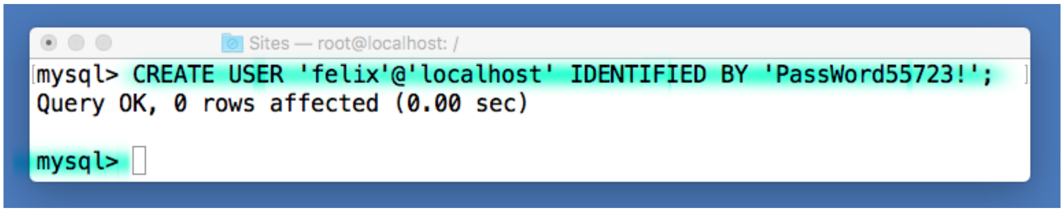
```
mysql> Select User FROM mysql.user;
+------------------+
| User             |
+------------------+
| felix            |
| luna             |
| felix            |
| debian-sys-maint |
| felix            |
| luna             |
| mysql.session    |
| mysql.sys        |
| root             |
+------------------+
9 rows in set (0.00 sec)

mysql> 
```

Figure 5.9: **SELECT user FROM mysql.user;**

Note that mysql command keywords and table names are case insensitive. You could have typed either **Select User** or **SELECT user** to the same effect!

5.0.5 Creating A New MySQL User

In addition to root user, we can create other users by issuing following command:

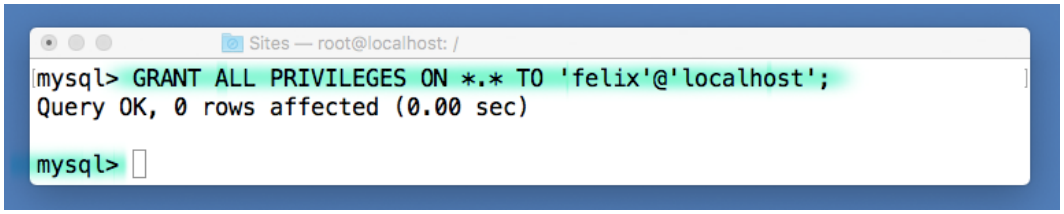


Sites — root@localhost: /

```
mysql> CREATE USER 'felix'@'localhost' IDENTIFIED BY 'PassWord55723!';
Query OK, 0 rows affected (0.00 sec)

mysql> 
```

CREATE USER 'felix'@'localhost' IDENTIFIED BY 'PassWord55723!';

But now we also need to give this user access privileges using **GRANT** command:



Sites — root@localhost: /

```
mysql> GRANT ALL PRIVILEGES ON *.* TO 'felix'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> 
```

GRANT ALL PRIVILEGES ON *.* TO 'felix'@'localhost';

This gives **felix** admin-level privileges on all tables in the database (***.***)

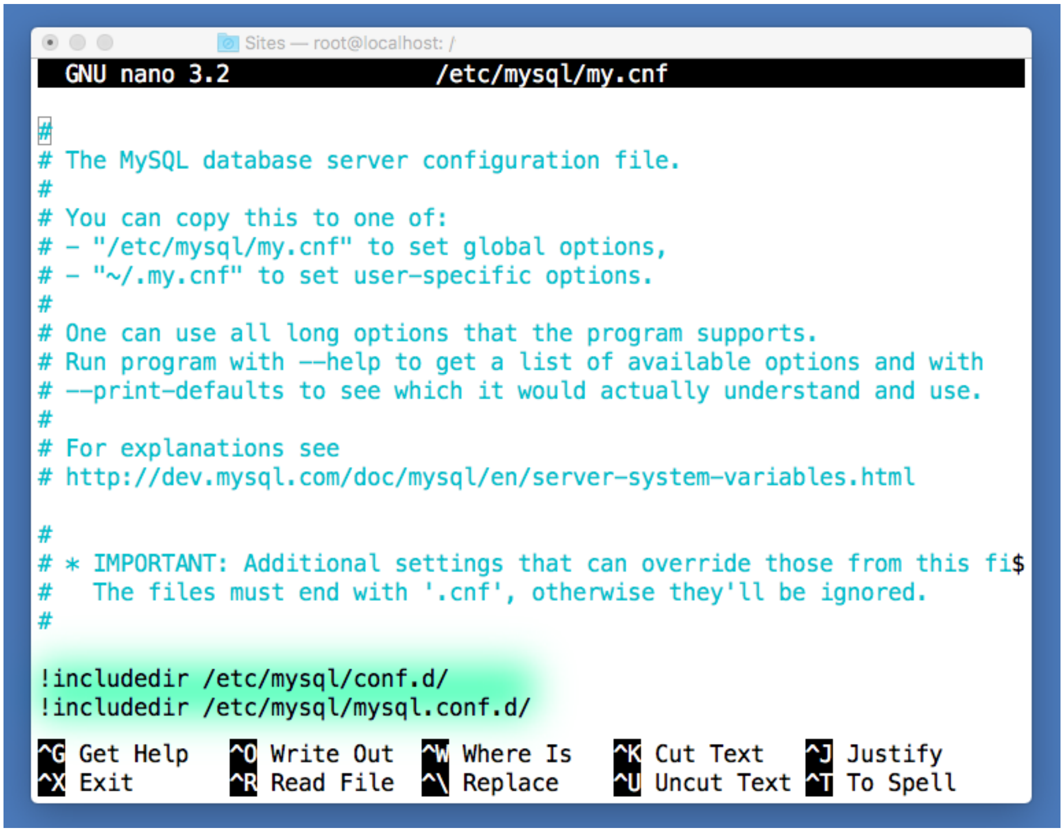
Now we have user **felix** ready to log in to the database via **localhost** address with provided password. This is the account you can use in your JavaScript code when we get to the **Chapter 6: Adding MySQL to api.js**

We just need to make sure our database is open for remote access. See next page!

5.0.6 Make MySQL Open For Remote Access

Enter **nano /etc/mysql/my.cnf** in your Linux bash to open your default MySQL configuration file with nano editor.

Usually **/etc/mysql/my.cnf** is the main configuration file. But it's *possible* that it might not be, depending on Linux and MySQL version, etc:

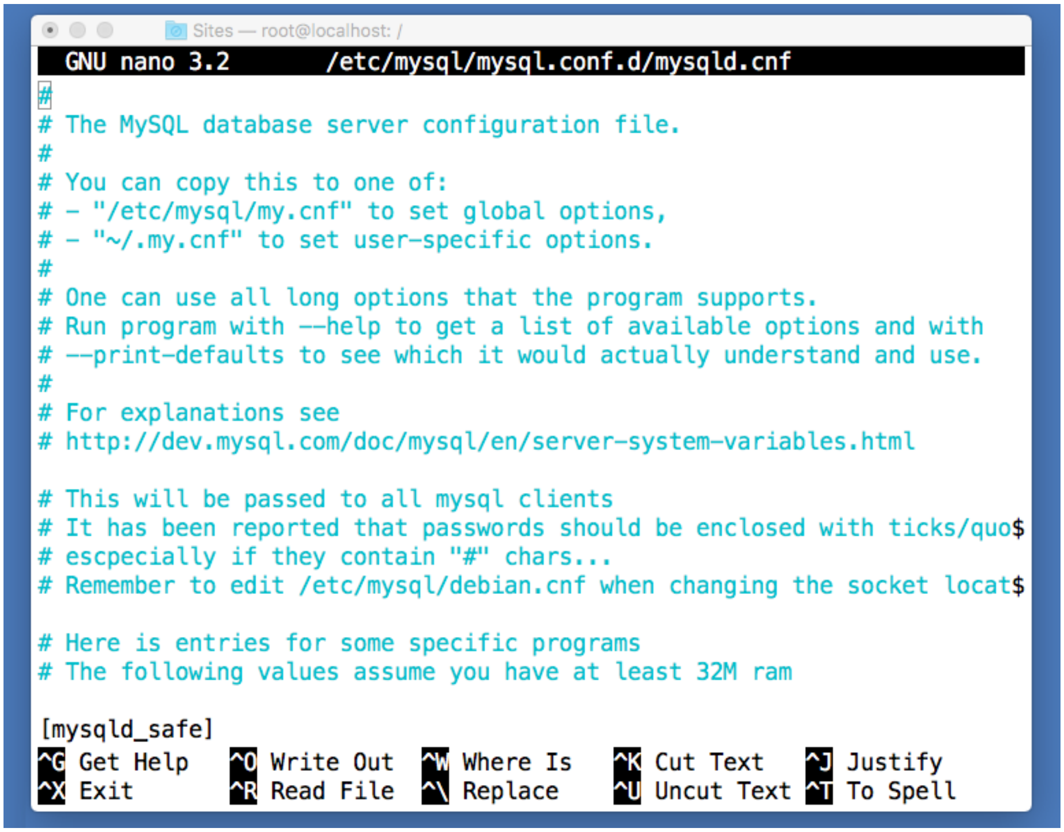


```
  GNU nano 3.2                     /etc/mysql/my.cnf

#
# The MySQL database server configuration file.
#
# You can copy this to one of:
# - "/etc/mysql/my.cnf" to set global options,
# - "~/.my.cnf" to set user-specific options.
#
# One can use all long options that the program supports.
# Run program with --help to get a list of available options and with
# --print-defaults to see which it would actually understand and use.
#
# For explanations see
# http://dev.mysql.com/doc/mysql/en/server-system-variables.html

#
# * IMPORTANT: Additional settings that can override those from this fi$
#   The files must end with '.cnf', otherwise they'll be ignored.
#

!includedir /etc/mysql/conf.d/
!includedir /etc/mysql/mysql.conf.d/
^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify
^X Exit        ^R Read File   ^\ Replace     ^U Uncut Text  ^T To Spell
```

If your **my.cnf** config looks like above, you need to find the actual mysql configuration file which should reside in one of the folders specified with **!includedir**.

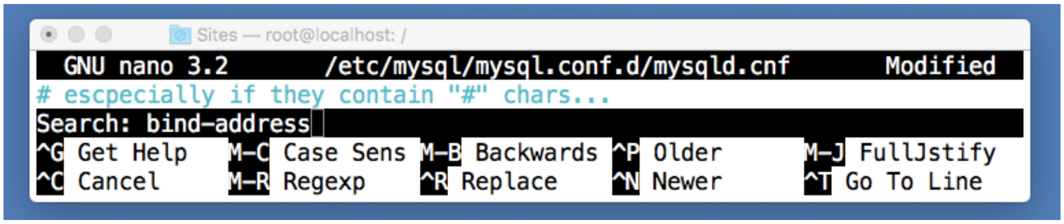
Snoop around these folders to find the file with a similar **.cnf** name. In my case it was **mysqld.cnf**, so I entered the following command to edit it:

nano /etc/mysql/mysql.conf.d/mysqld.cnf



```
 GNU nano 3.2          /etc/mysql/mysql.conf.d/mysqld.cnf
#
# The MySQL database server configuration file.
#
# You can copy this to one of:
# - "/etc/mysql/my.cnf" to set global options,
# - "~/.my.cnf" to set user-specific options.
#
# One can use all long options that the program supports.
# Run program with --help to get a list of available options and with
# --print-defaults to see which it would actually understand and use.
#
# For explanations see
# http://dev.mysql.com/doc/mysql/en/server-system-variables.html

# This will be passed to all mysql clients
# It has been reported that passwords should be enclosed with ticks/quo$
# escpecially if they contain "#" chars...
# Remember to edit /etc/mysql/debian.cnf when changing the socket locat$

# Here is entries for some specific programs
# The following values assume you have at least 32M ram

[mysqld_safe]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell
```

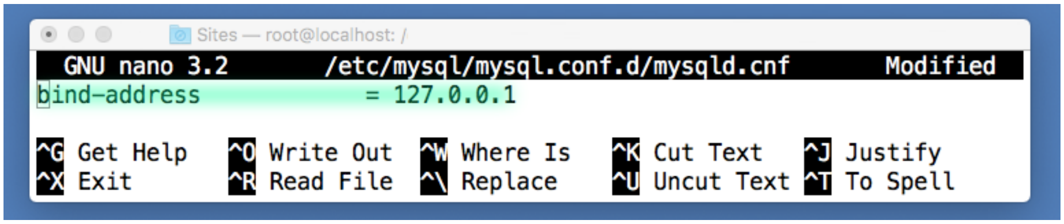
Note below comments the file starts with **[mysqld_safe]**. This is the recommended way to start MySQL server on Unix. On Windows you will probably have just **[mysqld]** in your config file, which may be called either **my.ini** or **my.cfg**. **[mysqld_safe]** tells MySQL server to restart in case of an error, among some other helpful features. If you see this command you're in the right MySQL config file.



```
 GNU nano 3.2          /etc/mysql/mysql.conf.d/mysqld.cnf          Modified
# escpecially if they contain "#" chars...
Search: bind-address
^G Get Help   M-C Case Sens M-B Backwards ^P Older       M-J FullJstify
^C Cancel     M-R Regexp    ^R Replace    ^N Newer       ^T Go To Line
```

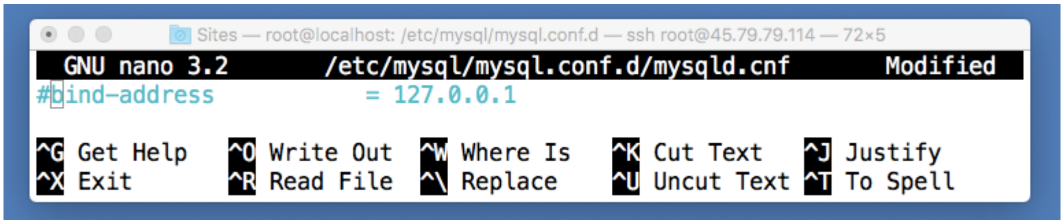
Hold **control**, press **W**, type **bind-address** and press **Enter**.

If you are in the right MySQL config file, your cursor will jump to this line:



```
  GNU nano 3.2        /etc/mysql/mysql.conf.d/mysqld.cnf        Modified
bind-address            = 127.0.0.1

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell
```

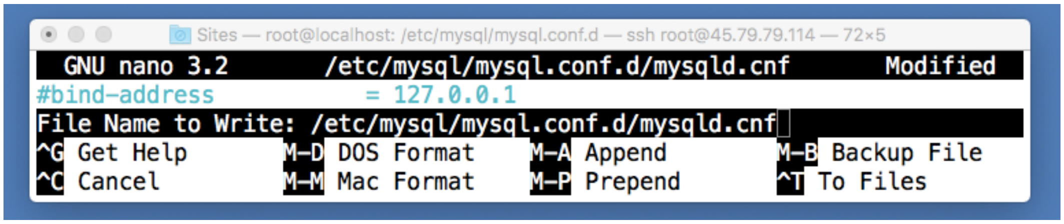
Comment the line **bind-address = 127.0.0.1** by adding # to the front:


Sites — root@localhost: /etc/mysql/mysql.conf.d — ssh root@45.79.79.114 — 72×5

```
  GNU nano 3.2        /etc/mysql/mysql.conf.d/mysqld.cnf        Modified
#bind-address           = 127.0.0.1

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell
```

Now that the line is commented...

Hold **control** and press **O**, then press **Enter** to save **.cnf** file:


Sites — root@localhost: /etc/mysql/mysql.conf.d — ssh root@45.79.79.114 — 72×5

```
  GNU nano 3.2        /etc/mysql/mysql.conf.d/mysqld.cnf        Modified
#bind-address           = 127.0.0.1
File Name to Write: /etc/mysql/mysql.conf.d/mysqld.cnf
^G Get Help        M-D DOS Format     M-A Append         M-B Backup File
^C Cancel          M-M Mac Format     M-P Prepend        ^T To Files
```

Type **/etc/init.d/mysql start** or **/etc/init.d/mysql restart** in Linux bash.

This will restart the MySQL server using the new configuration file.

Congrats, your server will now accept connections remotely.

This means from now you can log into your MySQL server using a MySQL client such as **Sequel Pro** on Mac or **MySQL Workbench** on Windows, using the login token pair **felix**/**PassWord557123!** we created earlier.

Final Words

Whenever you start a new project, if a hosted database is the only option, you will probably want to open your MySQL database for remote access. This way you can access it from your localhost environment without having to install MySQL server on your machine.

However, this is usually considered bad security practice. At one point, you will want to limit your MySQL database access to localhost only. This means only your Node server can connect to the database directly.

This still doesn't prevent database injection attacks.

Injection attacks use your API endpoints to pass actual MySQL query code into one of the *values* sent together with the POST or GET requests.

By clever use of quote characters the query can be re-constructed to execute a different MySQL command which can be virtually anything.

Keep this in mind when writing your endpoint code and generally thinking about how to secure your database!

Writing MySQL Code

Now that we have our mysql server running and waiting for incoming user connections, we need to write some code on our Node back-end (**index.js**) as a starting point for our MySQL-based API.

In the next chapter of this book we will add new mysql code by starting a connection to the server and executing a query. From that point on you can extend the API by implementing the entire CRUD pattern.

CRUD

We already started **api.js** file earlier in this book, which – at this point – can only interpret certain URL patterns as API endpoints and cancel serving the file.

But we are still not making any MySQL connections to the server. Nor are we executing MySQL queries to actually **Create**, **Read**, **Update** or **Delete** rows – collectively known as the **CRUD** API.

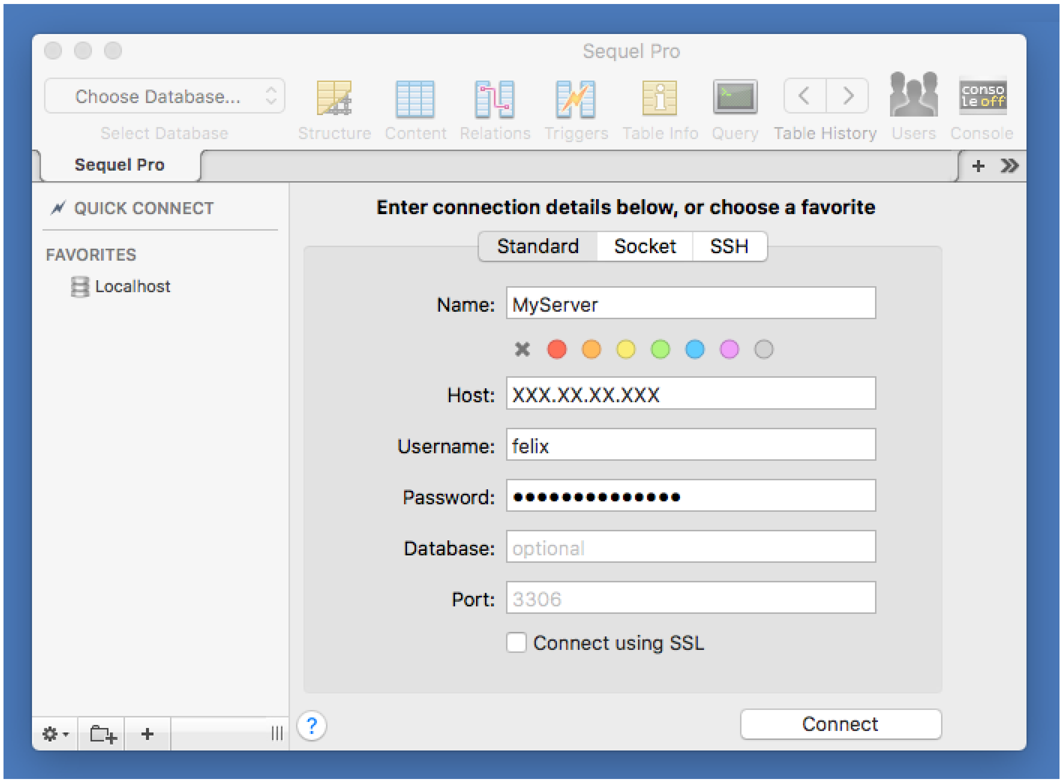
Once you implement the **CRUD** on your server, you will be ready to build out the rest of your application. Usually the CRUD gives you ability to code pretty much any feature on the front-end.

5.1 Table Specimen

In order to execute queries on a MySQL database, we need to create a test table.

For examples that follow we will assume that **user** table already exists.

You can create it via command line, but it is recommended to get good UI-based database manager software such as **Sequel Pro** on Mac or **MySQL Workbench** on Windows. As your application-data tables begin to evolve editing them from command line might turn into Hell.



Enter connection details below, or choose a favorite

Name: MyServer

Host: XXX.XX.XX.XXX

Username: felix

Password: ••••••••••••••

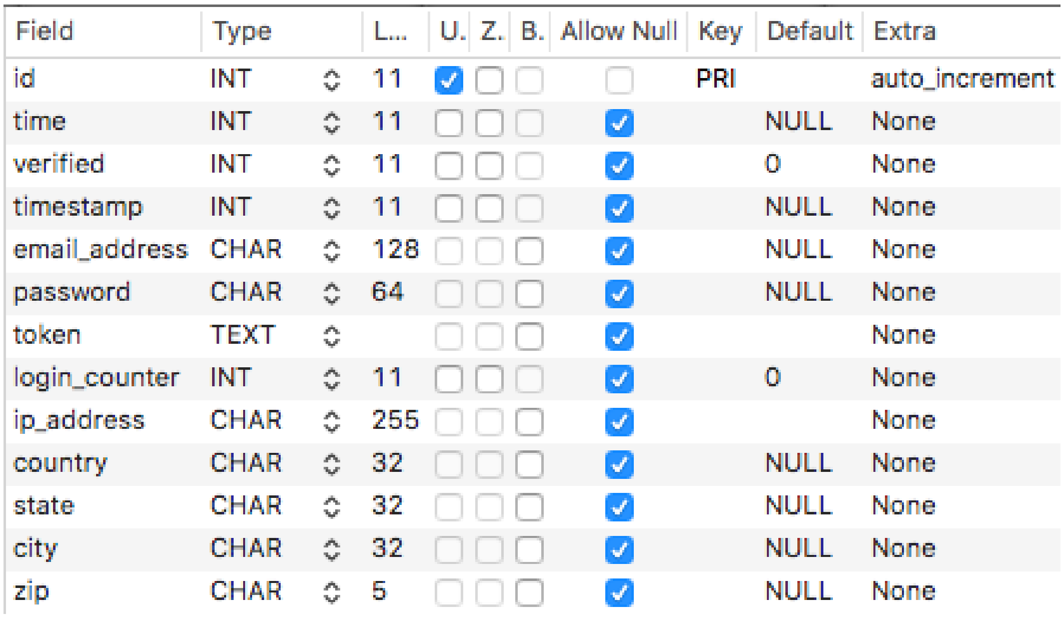
Database: optional

Port: 3306

Connect using SSL

Because our MySQL server is open for remote connection, it's possible to log in to it from **Sequel Pro** using **felix** account created earlier. If you set up MySQL server on **localhost** use that as the Host. Otherwise use your web host's database IP address (which very often is the same IP address for your web hosting server. It makes senes, because technically MySQL is installed on "localhost" there too.)

Once logged into your favorite MySQL editor, create the **user** table that looks similar to the following design:



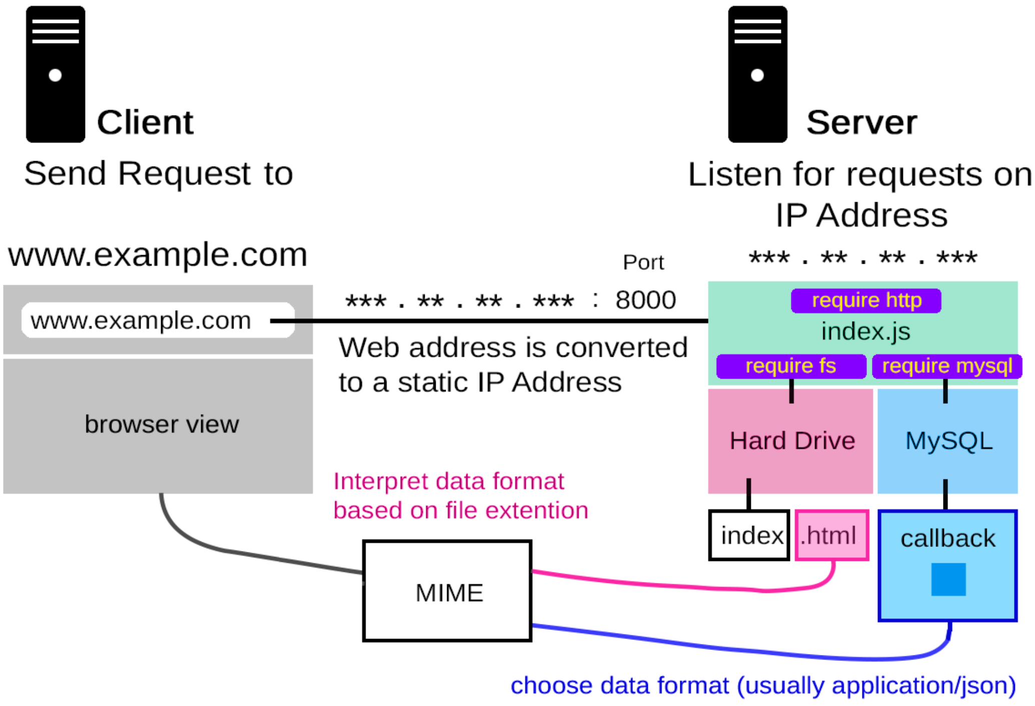
Field	Type	L...	U.	Z.	B.	Allow Null	Key	Default	Extra
id	INT	11	✓				PRI		auto_increment
time	INT	11				✓		NULL	None
verified	INT	11				✓		0	None
timestamp	INT	11				✓		NULL	None
email_address	CHAR	128				✓		NULL	None
password	CHAR	64				✓		NULL	None
token	TEXT					✓			None
login_counter	INT	11				✓		0	None
ip_address	CHAR	255				✓			None
country	CHAR	32				✓		NULL	None
state	CHAR	32				✓		NULL	None
city	CHAR	32				✓		NULL	None
zip	CHAR	5				✓		NULL	None

Perhaps your table can be completely different. But for users at least have a **username** and **email_address** columns!

5.1.1 Bird's Eye View

Before we move on to the next chapter, I want you to take a look at the bird's eye view architecture we've got going on so far.

This is the mental model you will eventually arrive at after breaking down a simple Node server into compartments.



Our barebones server requires only **http**, **path**, **fs** and **mysql** modules.

Note that the **mysql** object from MySQL package will return result in a separate process in a callback function. So it's a bit different from how you would expect MySQL to work on a PHP server.

If we can find a way to handle the result and write back to the requesting client from the callback, we can make API calls just as you would expect from Apache / PHP setup. Think of it as an extra step. This makes Node MySQL API architecture a bit more challenging, but also more flexible.

Chapter 6

Adding MySQL to api.js

Now that we have our database running either on **localhost** or at some remote location, we're ready to integrate our MySQL code into our server, so that we can convert our API endpoint calls to MySQL queries.

Go to our **app.js** file and add lines highlighted in green to the very top of the file (so that now we will have **database** and **API** classes together in the same file).

If you think class **database** should be **Database**, go ahead and change it to your heart's desires! The only reason it's in lowercase is because we're only using static methods here, and in my opinion **database.create** looks more elegant.

```
let mysql = require('mysql');

class database {
    constructor() { }
    // Create MySQL connection
    static create() {
        let message = "Creating MySQL connection...";
        this.connection = mysql.createConnection({
            host     : 'XX.XX.XX.XXX',
            user     : 'XXXXX',
            password : 'XXXXX',
            database : 'XXXXX'
        });
        this.connection.connect();
        console.log(message + "Ok.");
    }
    // Execute a MySQL query
    static exec(command) {
        let Q = 'SELECT email_address FROM user';
        this.connection.query(Q, (error, results, fields) =>
          if (error)
              throw error;
          let result = results[0];
          console.log("email = " + result.email_address);
        });
    }
}

class API {
    constructor() { }
    static exec() {
        let parts = API.parts;
        ....
```

Make sure to replace **XX.XX.XX.XXX** with your actual web server IP, or if you're running Node server locally **localhost** (or **127.0.0.1**).

Replace other XXXXX entries with your actual database **username** (felix), **password** (PassWord55723!) and **database** name – all of which were set up in the previous section – or just use your own values.

The static function **create()** can now be used to create a database connection.

You can call **database.create()** once from the very top of **index.js** after **require** statements. Your server will connect to the MySQL database and you will be ready to start executing MySQL queries.

Inside static function **exec** all we do is call **connection.query** method on our database connection object.

6.1 Asynchronous Back-End Problem

If you are coming to Node from Apache PHP server, it is inevitable that you will experience a paradigm shift, when trying to write your own mysql queries.

In PHP, in order to execute a mysql query, all you had to do was write a script such as **get_user.php** on the back-end.

Then you would use **fetch API** or a **jQuery.ajax** to call that file. The result would be generated as part of the page content and returned inside the *callback*. This way you could safely update the front end UI. We used to call this asynchronous HTTP requests, or "ajax" calls. It works naturally for updating the front end!

A Node callback doesn't run in the same process with the function serving the file to browser. So the file will be served before the mysql query returns! And this is the fundamental problem we need to solve in order to build a decent API server.

Executing a MySQL query works exactly the same on Node. The result is returned via a callback function in a separate process. And that's part of the problem, because we're not on front-end anymore.

A callback will often depend on completion of another MySQL query. So we will also need a way to execute a chain of multiple queries. And this can introduce a whole new set of problems.

Solution. If we can somehow forge the process of serving the file with the result we get from mysql query, we're taking first step toward solving this problem.

You can either use **nesting queries** or (more elegantly) **promises**.

6.2 Nesting Queries

First, let's take a look at what problem we're trying to solve with nested queries.

Associative Tables

Let's say you want to store **followers** of all existing **users**. You can't store **users** and **followers** in the same table because they are two completely different types of data sets.

However, you can store **followers** in a separate table and "associate" it with **users** by adding **userID** column to it.

This way when you want to access an entry in **followers** table you can use the **userID** column to look up which user it belongs to.

Nesting Queries

For the reason stated above, you will often need to write *chains* of nested queries.

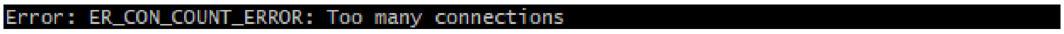
To get **followers** from a user account, first you will retrieve a **user** from **users** table. Then you will retrieve followers associated with that user's primary key ID.

Because **followers** table has a link to the user via the **user_id** column, you should be able to gather a list of followers separately for that individual user.

6.3 Handling Many Connections

Default server configurations start out with some arbitrary connection limits. For latest versions of MySQL this limit is **151** maximum connections.

If your app connects many users at the same time, this limit will be reached and your program will cause the famous **"too many connections"** error.



```
Error: ER_CON_COUNT_ERROR: Too many connections
```

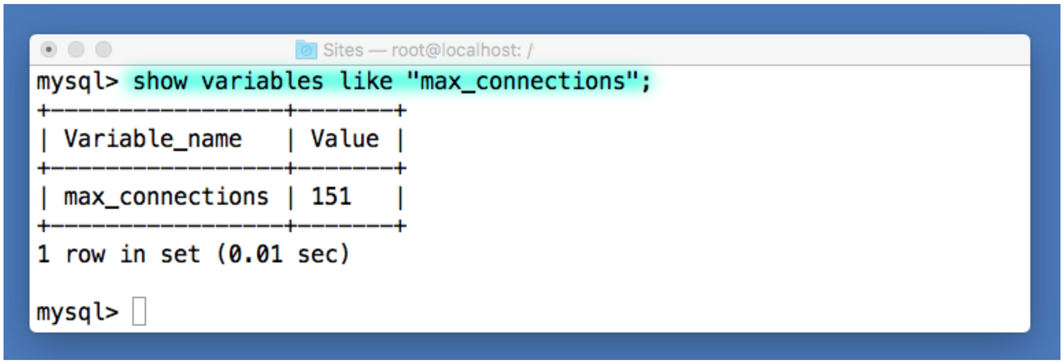
Figure 6.1: **ER_CON_COUNT_ERROR**

This is just a MySQL error caused by configuration limits. It doesn't actually mean your server can't handle that many. In general, these limits are set by default because every time you initiate a connection, a new thread is created.

Creating a new thread process just to serve 1 user will inefficiently use up your server's memory resources. Of course, once we no longer need it, this thread is eventually terminated by our code. But with thousands of users connecting every other second you are creating anywhere from hundreds to thousands of threads *at the same time* hogging the CPU and possibly the hard drive.

When developing your server you might run into configuration limits on your MySQL database.

To find out the maximum number of allowed connections, enter the **mysql** prompt (by typing **mysql** either on mac or PC – it's the same command) and enter the mysql command: **show variables like "max_connections";**

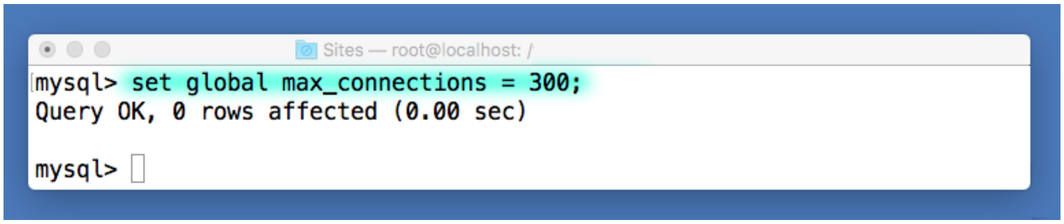


```
mysql> show variables like "max_connections";
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 151   |
+-----------------+-------+
1 row in set (0.01 sec)

mysql> 
```

Figure 6.2: **show variables like "max_connections";**

Don't forget the **semicolon** at the end of each mysql command or it will not run.

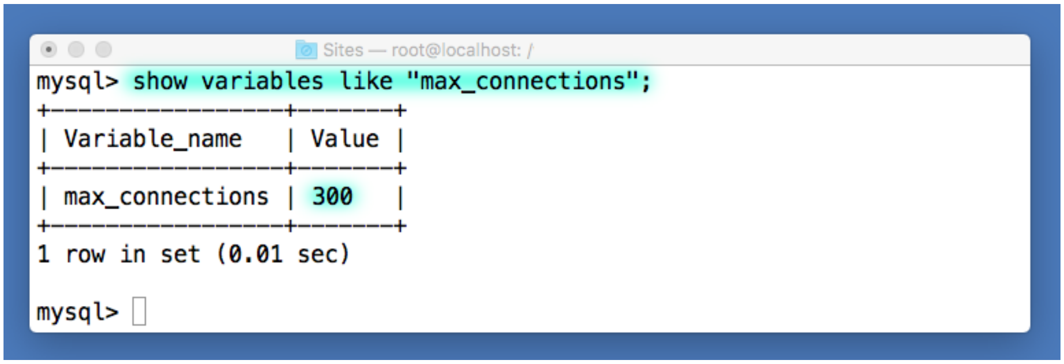
I'm running **set global max_connections = 300;** to double connection limit:



```
mysql> set global max_connections = 300;
Query OK, 0 rows affected (0.00 sec)

mysql> 
```

Figure 6.3: **set global max_connections = 300;**

And if everything went well, running show variables like command should yield the following result:



```
mysql> show variables like "max_connections";
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 300   |
+-----------------+-------+
1 row in set (0.01 sec)

mysql> 
```

Figure 6.4: **show variables like "max_connections";**

We've just expanded the theoretical number of simultaneous MySQL connections.

Creating MySQL connections is computationally expensive. Outsourcing the server workload by increasing the number of connections might work but it is still a naive strategy. And often, just one connection is enough to handle thousands of queries.

Having said this, before increasing the number of connections, make sure your system has enough resources to handle them.

However, this doesn't prevent programmers from using **connection pools** described in next section. Connection pools might offer a fair deal of additional optimization. Just keep in mind, each architectural decision will depend on the actual purpose of your API server.

6.4 Executing A MySQL Query

So far we explored the idea of executing a MySQL query with **exec(command)** method residing on our main **database** class as a static function.

But architecturally, this won't do any good as far as our **API** goes. Remember that on Node a MySQL query returns in a *callback function* which produces the query result value away from our main execution thread.

That's fine if all you want to do is execute a query or two without caring about sending a response back to the browser.

But if you want to build an API which deals with a series of unique endpoint calls, you might seriously consider using **promises**. They can help you to properly respond to each incoming request with data returned from a MySQL query.

6.4.1 Promises

Promises add one extra layer to the standard callback pattern. When the Promise returns, it returns a Promise object, instead of the callback function. This Promise object is not guaranteed to carry a value yet, until it is resolved at a later time, which is exactly why it's called a promise. You can use the built-in **.then** method on the Promise object to retrieve the results from the MySQL query.

The Promise's **.then** method allows us to wait for the result and resolve it soon as it becomes available even if it happens at a later time.

Using **.then** method allows you to avoid the nested callback pattern (or at least reduce its occurence) where you keep nesting a callback within a callback, because one API call depends on the data returned from the previous one.

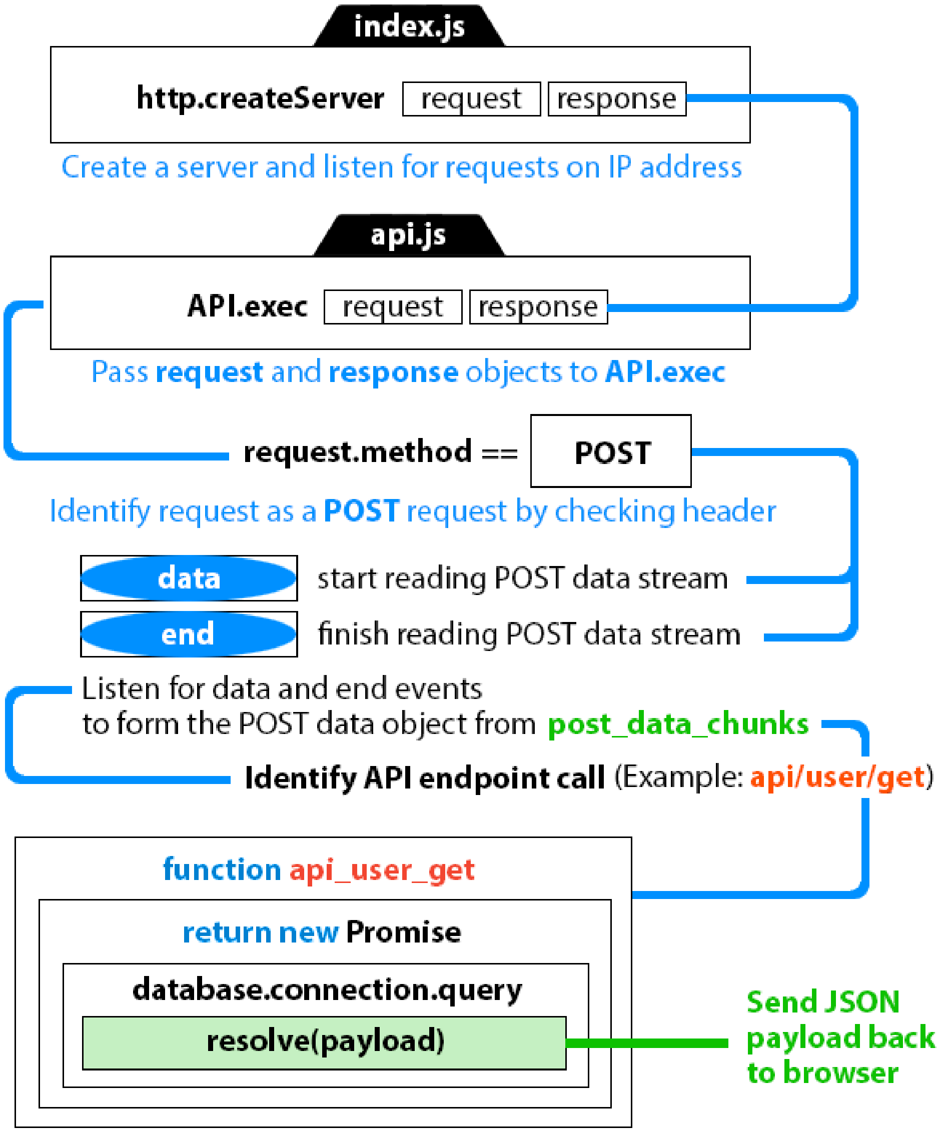
Promise-based MySQL query

So what does a promise-based MySQL query call look like in Node?

Let's take a look at that in the context of entire **API endpoint round trip** (Which includes requesting a resource using an API endpoint and responding to it.)

6.4.2 The Round-trip Pattern

Before we dive into source code, let's look at how it all fits together, starting from server listener and finally returning a value from MySQL query.


index.js
http.createServer request response
Create a server and listen for requests on IP address
api.js
API.exec request response
Pass request and response objects to API.exec
request.method == POST
Identify request as a POST request by checking header
data start reading POST data stream
end finish reading POST data stream
Listen for data and end events
to form the POST data object from post_data_chunks
Identify API endpoint call (Example: api/user/get)
function api_user_get
return new Promise
database.connection.query
resolve(payload)
Send JSON payload back to browser

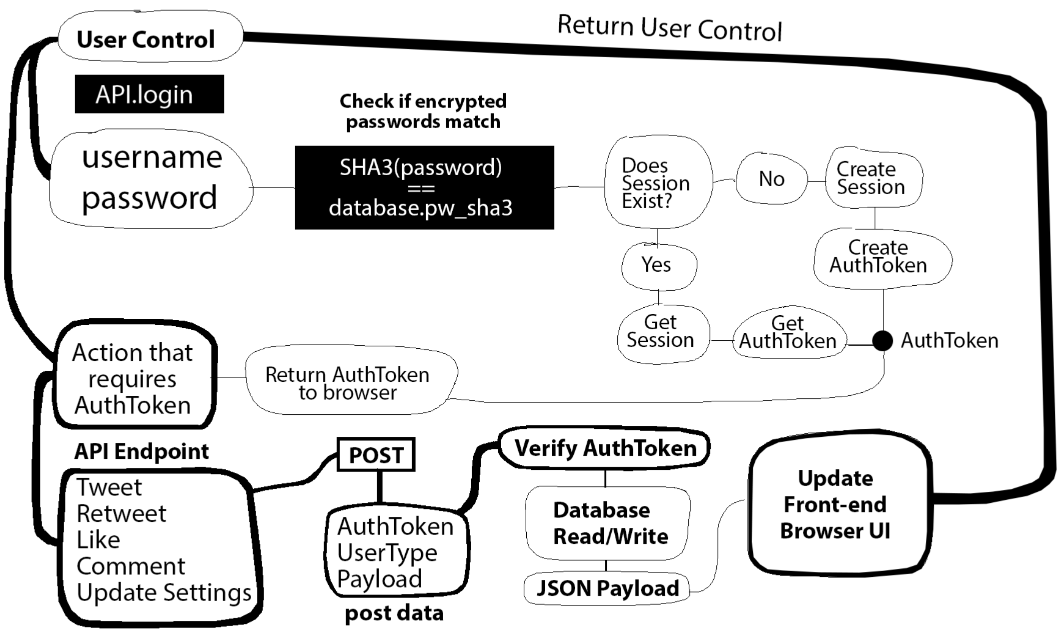
6.4.3 MySQL Promise Function

One of the ways to build the function responsible for executing the MySQL query related to an endpoint call is by wrapping it in a promise and returning the result using **resolve** method - just pass payload to it in JSON format. Here is an example of just the **action_register_user** function, but all other actions (login, tweet, comment, etc.) follow the same pattern:

```
// Requires payload.email = <email_address>
         payload.password = <password>
function action_register_user ( request, payload ) {
    return new Promise((resolve, reject) => {
        if (!request || !request.headers || !payload)
            reject("Error: missing request or payload");
        let q = `SELECT email FROM user`;
            q += `WHERE email = '${payload.email}' LIMIT 1`;
        // Check if user already exists in database
        database.connection.query(q, (error, results) => {
            if (error)
                throw(error);
            let result = results[0];
            if (results &&
                results.length != 0 &&
                result.email == payload.email)
                resolve(`{"success": false}`);
            else {
                // Encrypt password with sha3 algorithm
                let password_sha3 = sha3(payload.password);
                let fields = "(`email`, `password_sha3`)";
                let values = `VALUES('${payload.email}',`;
                    values += `'${password_sha3}')`;
                let q = "INSERT INTO user " +
                         fields + " " + values,
                // Create new user in database
                database.connection.query((error, results)=>{
                    if (error)
                        throw(error);
                    resolve(`{"success": true}`);
                });
            }
        });
    }).catch((error) => { console.log(error) });
}
```

6.5 Endpoint API Architecture

Below is a diagram displaying the minimum structure of a functioning Node API server. It's the same diagram as you've just seen above, except in the context of a concrete API call to execute **user login** action.



Logging in should be treated as a special API action. It's the only endpoint that performs the initial authentication step.

This is where you will take user's password, encrypt it with an algorithm such as SHA3, and compare it with the SHA3 password already stored in the database under **password_sha3** in SHA3-encrypted format.

The primary purpose of logging in is to obtain an **AuthToken**. This becomes your key to perform other authenticated endpoint actions – **follow**, **tweet**, **retweet**, **like**, **comment**, **update** user settings, etc.

6.6 Helper Functions

To avoid rewriting redundant code, I wrote these functions that accomplish various actions you'll commonly come across when writing your Node API server. Using them in various places will help reduce clutter and keep your code clean.

identify(target, action)

target argument is database table name (such as **user**, **tweet**, etc.) The **action** argument is the "what" you want to do to target (**get**, **update**, **delete**, etc.)

```
// Check if API.parts match a URL pattern
// example: "api/user/get"
function identify(target, action) {
     return API.parts[0] == "api" &&
            API.parts[1] == target &&
            API.parts[2] == action;
}
```

Figure 6.5: Identify an API endpoint request from **API.parts**. The **API.parts** array is available globally. It represents the API endpoint call broken down into 3 parts. If there is a match, the return value is **true**.

respond(response, content)

The **response** argument is the response object from the server and **content** is a string representing an object in JSON format (to send back to the browser) which is usually formed by **resolve** method inside a promise from MySQL function.

```
// General use respond function --
// Send JSON object back to the browser
// in response to an API request
function respond( response, content ) {
    console.log("responding = ", [ content ]);
    const jsontype = "{ 'Content-Type':'application/json' }";
    response.writeHead(200, jsontype);
    response.end(content, 'utf-8');
}
```

json(chunks)

In Node POST data is streamed instead of received as a single object. So we need this json function that will convert all received chunks into actual JSON object.

```
// Convert buffer to JSON object
function json( chunks ) {
    return JSON.parse( Buffer.concat( chunks ).toString() );
}
```

Figure 6.6: Form a JSON object from the streamed POST data chunks.

I intentionally keep my function names short and use common names. This way when putting it all together the code turns out to be so much cleaner, readable and easy to maintain down the road.

6.7 Sending POST Data To API Endpoint

Usually an API endpoint expects additional data sent together with the request.

For example, if we want to **get user data** for user with **user id** of 1, we need to include that **user id** in the POST request as additional data specified in JSON format, for example: {`"user_id": 1`};

When you receive the **request** object, you only get the header data instantly. The POST data is submitted separately and you need to listen for it using two different events: **"data"** and **"end"** because it's sent as a stream / buffer.

The **request.data** event tells you that there is POST data incoming. But instead of sending entire data object, Node streams it in chunks. So you need to put it together using chunks first. One way of doing that is to create an array and keep pushing each chunk received in the **"data"** event's callback onto that array.

The **request.end** event tells you that all chunks of the POST data have been finished downloading. At this point you would take the **chunks** array and convert it to a JSON object.

This is done using Node's built-in **Buffer.concat(chunks).toString()** method. The **.toString()** method is required to convert binary chunk data to string format.

After this you simply convert the resulting string to a JavaScript object using the native **JSON.parse** function. We'll see the source code for that in just a moment!

If the POST data is short, it is possible to receive only one chunk. But you still need to intercept it as a stream.

6.8 Streaming POST Data

Let's look at streaming POST data as an isolated example first:

```
class API {
  constructor() { }
  static exec( request, response ) {
    if (request.method == 'POST') {

      // Chunk collector array
      request.chunks = [];

      // Step 1.) "data" event - start reading POST chunks
      request.on('data', segment => {
        // Push this chunk onto array
        request.chunks.push(segment);
      });

      // Step 2.) "end" event - POST data fully received
      request.on('end', () => {

        request;                 // request object
        response;                // response object
        request.chunks;          // all POST data chunks
        json( request.chunks );  // convert chunks to JSON

        // Step 3.) Identify API endpoint
        //        a) execute endpoint action function
        //        b) return result back to browser
      });
    }
  }
}
```

Going from here we can finish building out the entire round-trip pattern.

(See next page.)

Knowing everything we do now, let's upgrade our API class by adding support for one endpoint to enable User Registration. This endpoint triggers the function we wrote earlier: **action_user_register**

```
class API {
  constructor() { }
  // Identify and execute an API endpoint request
  static exec( request, response ) {
    console.log("API.exec(), parts = ", API.parts);
    if (request.method == 'POST') {
      request.url[0] == "/" ? request.url =
      request.url.substring(1, request.url.length) : null;
      request.parts = request.url.split("/");
      request.chunks = [];
      // Start reading POST data chunks
      request.on('data', segment => {
        // 413 = "Request Entity Too Large"
        if (segment.length > 1e6)
          let type = {'Content-Type': 'text/plain'};
          response.writeHead(413, type).end();
        else
          request.chunks.push(segment);
      });
      // POST data fully received
      request.on('end', () => {
        API.parts = request.parts;
        // Register (create) user
        if (identify("user", "register"))
          action_register_user(request, json(request.chunks))
          // Return result back to the browser
          .then(content => respond(response, content));
      });
    }
  }
  static catchAPIrequest(request) {
    request[0] == "/" ? request =
    request.substring(1, request.length) : null;
    if (request.constructor === String)
      if (request.split("/")[0] == "api") {
        API.parts = request.split("/");
        return true;
      }
    return false;
  }
```

POST data is not available until request produces "end" event. This is where we put the chunks together into actual JSON object using the **json** helper function.

To add support for more endpoints just follow the same pattern. Place them inside **request.on('end')**'s callback which ensures POST data finished streaming:

```
// Finish reading POST data chunks
request.on('end', () => {

  API.parts = request.parts;

  if (identify("user", "register")) // Register (create) user
    action_register_user( request, json( request.chunks ) )
    .then( content => respond(response, content) );

  if (identify("user", "login")) // Log in
    action_login( request, json( request.chunks ) )
    .then( content => respond(response, content) );

  if (identify("user", "logout")) // Log out
    action_logout( request, json( request.chunks ) )
    .then( content => respond(response, content) );

  if (identify("user", "delete")) // Delete user
    action_delete_user( request, json( request.chunks ) )
    .then( content => respond(response, content) );

  if (identify("user", "get")) // Get user data
    action_get_user( request, json( request.chunks ) )
    .then( content => respond(response, content) );

  if (identify("user", "update")) // Update user
    action_update_user( request, json( request.chunks ) )
    .then( content => respond(response, content) );

  if (identify("session", "create")) // Create session
    action_create_session( request, json( request.chunks ) )
    .then( content => respond(response, content) );

  if (identify("user", "authenticate")) // Authenticate user
    action_authenticate_user( request, json( request.chunks )
    .then( content => respond(response, content) );
});
```

The **request** and **response** objects come all the way from initial call to **http.createServer(function(request, response)...** in **index.js**.

6.9 Complete Source Code

To see the complete source code of the Node server we've built so far, you can fork it from the public GitHub repository for this book:

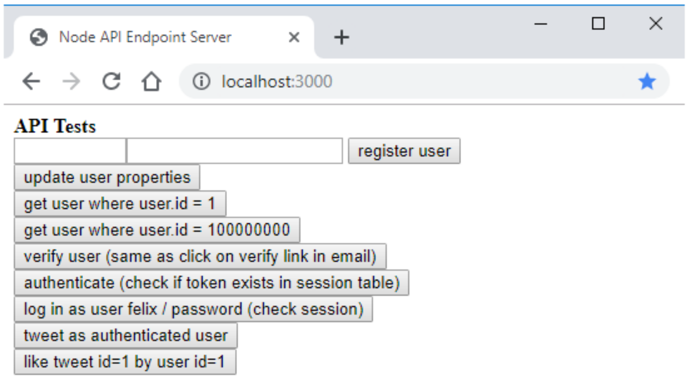
https://github.com/javascriptteacher/node

Note: Before running this app on your localhost (**cmd** > **node index.js**) make sure you already have a running MySQL server with user table in it and you included MySQL credentials (IP, username, password and database name) in **api.js**

Note: This is not a complete application, but it has enough scaffold code to get you started fast. User API is essential to most web applications. The rest of endpoints will depend on your app.

Note: You should replace **md5** with **sha3** if you need to ramp up security for your user passwords. Use **npm install sha3** to install the **SHA3** module which also includes **Keccak**.

After running **node index.js** you should see **index.html** at **http://localhost:3000**


Node API Endpoint Server
localhost:3000
API Tests
register user
update user properties
get user where user.id = 1
get user where user.id = 100000000
verify user (same as click on verify link in email)
authenticate (check if token exists in session table)
log in as user felix / password (check session)
tweet as authenticated user
like tweet id=1 by user id=1

Press on the API endpoint test buttons and watch console.

You can integrate these API calls into your front-end UI.

How To Run The Server

Import GitHub files into your project or copy them into one folder.

Navigate to that folder from command line.

Enter: **node index.js** and your **index.html** should be accessible via **http://localhost:3000** in your browser.

From here on, it's up to you how you want to build the rest of the front end!

Chapter 7

Building Scallable Applications

So far we learned how to build an API Node server that uses MySQL to perform storage and the classic CRUD data operations.

MySQL, MariaDB, MongoDB and other similar software might work well in the beginning stages of your application, running on a single server.

You will be able to host under 1K - 10K users (or maybe even more) based on the amount of RAM available, the IO rate of your SSD drive, and how well your server code is optimized. Fine tuning your server and writing code optimized for performance can be magical for **increasing the number of users** that can be served by a single instance of your server.

But once your server's maximum bandwidth is exhausted, it will start coming to a crawl if any more users join and start using your application.

In the olden days, Battle.net running Diablo I and StarCraft I used to work off a **single server** with hundreds of thousands of users connected simultaneously. This is an example of how limited resources encourage efficient code.

It's not uncommon for web applications of today to be real-time. Additional thought and engineering must be applied if you want to build large-scale applications that serve hundreds of thousands or even millions of simultaneous users.

7.1 Moving To RAM

No modern real-time application (Twitter, for example) uses SQL databases to serve primary features to their users. The reason is quite simple. When using SQL for everything it means there is a hard drive operation every time someone makes a request! This can be an overkill, because you are outsourcing the slowest part of your system to serve key features of your application.

One solution is to switch to an SSD server. But what if we store all of our data in RAM instead? This way we can avoid touching the hard drive completely.

Of course, the original data will still be stored on the hard drive. The difference is that when you launch your Node server, it will be copied into a JavaScript object.

When client requests a certain resource we can simply serve it directly from that object. It will contain a copy of the data your MySQL database would contain. But your application will become significantly more efficient.

You will still write this object to the hard drive, so it can permanently store the most recent data, but not as frequently as you would with a 100% MySQL setup.

Some databases such as MongoDB also implement similar RAM caching features. But they still may not be optimized for the primary purpose of your application which only you know. Thus missing the chance of taking advantage of the full potential of your server's horsepower.

And just like with anything else, you can either "install a package," or install and configure existing software, or you can learn to write your own code. Without the latter, you may never be able to understand or develop the skill of building an extremely efficient system for serving millions of users.

[This content is currently WIP]

7.2 Handle Server Clean Up

The server can exit for many reasons. For example, it can time out based on default timeout value provided in server settings.

There are a few cases in which your server might exit, either spontaneously as a response to an exception, or when **Ctrl + C** combo is detected.

You might want to do some code clean up when that happens:

```
// Server process exit
process.on('exit', function () {
    database.connection.end();
    console.log('process.exit');
});

// SIGINT happens when user hits Ctrl + C
process.on('SIGINT', function () {
    console.log('Ctrl-C...');
    database.connection.end();
    process.exit(2)
});

// Handle uncaught exception
process.on('uncaughtException', function(e) {
    console.log(e.stack);
    database.connection.end();
    process.exit(99);
});
```

Let's kill database connection every time server exits for any reason.

7.3 Master and Stage Servers

After a production cycle or coding sprint marathon the new code must be deployed to the main **production server** so that your customers can take advantage of the new features. This is your main public application server.

You should probably set up a secondary **stage server** where you would deploy the same code. This is important for two reasons: 1] you can show your application to your client, teammates, investors, friends or others who can test the new version of the application *before the public release*, and 2] you can make sure deployment process doesn't break the main production server with new unexpected bugs.

The **stage server** can be used to test integrity of the newly deployed code, without risking the chance of deploying erroneous code directly to master, losing customers and having a panic attack.

Now you are thinking like a real production engineer!

Both **master** and **stage** are nothing more than two versions of your application running separately in two different locations. So I won't talk about how to set up a stage server – the process is exactly the same. The difference is in where you push your code to. We'll cover deploying from command line and GitHub.

Index