

Git and Getting Started With Node.js

Fundamentals of using GIT and GitHub, installing and building our first Node app.

Previous Week Review — 00:15

Author: Shaun McKinnon

QUIZ: Lesson 01 in Review — 00:10

Author: Shaun McKinnon

Quick setup in Git & GitHub — 00:20

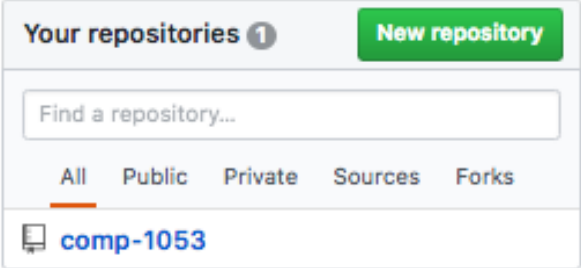
Working with GIT version control system and the online GitHub repository

Git: What is it?

- Git is a version control system
- Tracks changes in files
 - Commits only the changes that have occurred
 - This is what makes Git so awesome
 - Fast
 - Can easily see what has changed between two commits
- Allows for collaborative work while maintaining consistency, quality, and efficiency
 - Collaborators can work on various branches of a project
 - Branches are often used to manage features within a project
 - That branch then can be merged into the main branch, usually the master
- Merging can be controlled by the repository owner
 - The owner can prohibit merging for collaborators and force them to make a merge request instead
 - These are more commonly called a pull request
 - Pull requests allow collaborators to review each other's changes
 - Once a pull request has been okayed, the repository owner, or someone with the ability to merge, can merge the branch into the main branch

So, what do I do?

1. Ensure you have Git installed on your computer
<https://git-scm.com/downloads>
2. Once you have GIT installed, I recommend downloading the desktop application
<https://desktop.github.com/>
This will link with your GitHub account, making working with the remote and local repositories straight forward
3. Before launching the executable for **GitHub Desktop**, we should setup a GitHub account
4. Go through the steps of creating a new account
5.



Click **New Repository** to create a new repository

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner

gcsmkinnon

 /

comp-2068-lab-01

Great repository names are short and memorable. Need inspiration? How about **psychic-invention**.

Description (optional)

My first lab in Advanced Web Programming

Public

Anyone can see this repository. You choose who can commit.

Private

You choose who can see and commit to this repository.

☒ Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None**

Add a license: **None**

Create repository

6. Fill out the details for the repository name

and description. Choose **Public** (unless you are paying for a private repository) and choose to initialize with a **README**

gcsmkinnon / comp-2068-lab-01

Unwatch

1

Star

0

Fork

0

<> Code

Issues 0

Pull requests 0

Projects 0

Wiki

Insights

Settings

My first lab in Advanced Web Programming

Edit

Add topics

1 commit

1 branch

0 releases

1 contributor

Branch: master

New pull request

Create new file

Upload files

Find file

Clone or download

gcsmkinnon

Initial commit

Latest commit f6d1c95 just now

README.md

Initial commit

just now

README.md

comp-2068-lab-01

My first lab in Advanced Web Programming

7. Congratulations! Your first

repository is created!

8. Open **GitHub Desktop**
9. Sign In with your new GitHub account credentials



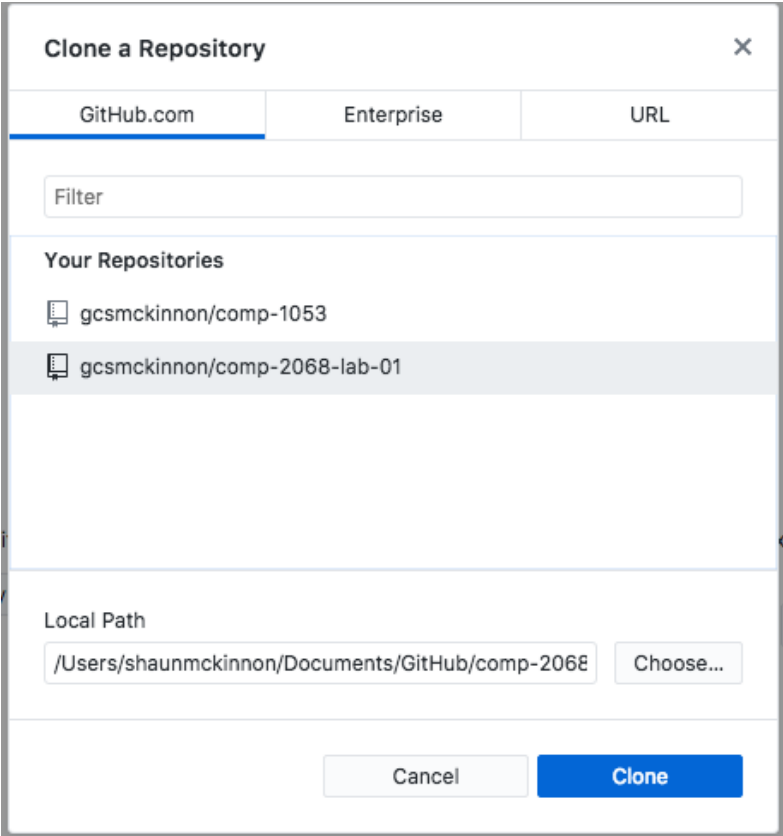
Clone an existing project from GitHub to your computer

Click **Clone a Repository**

10.

Clone a Repository

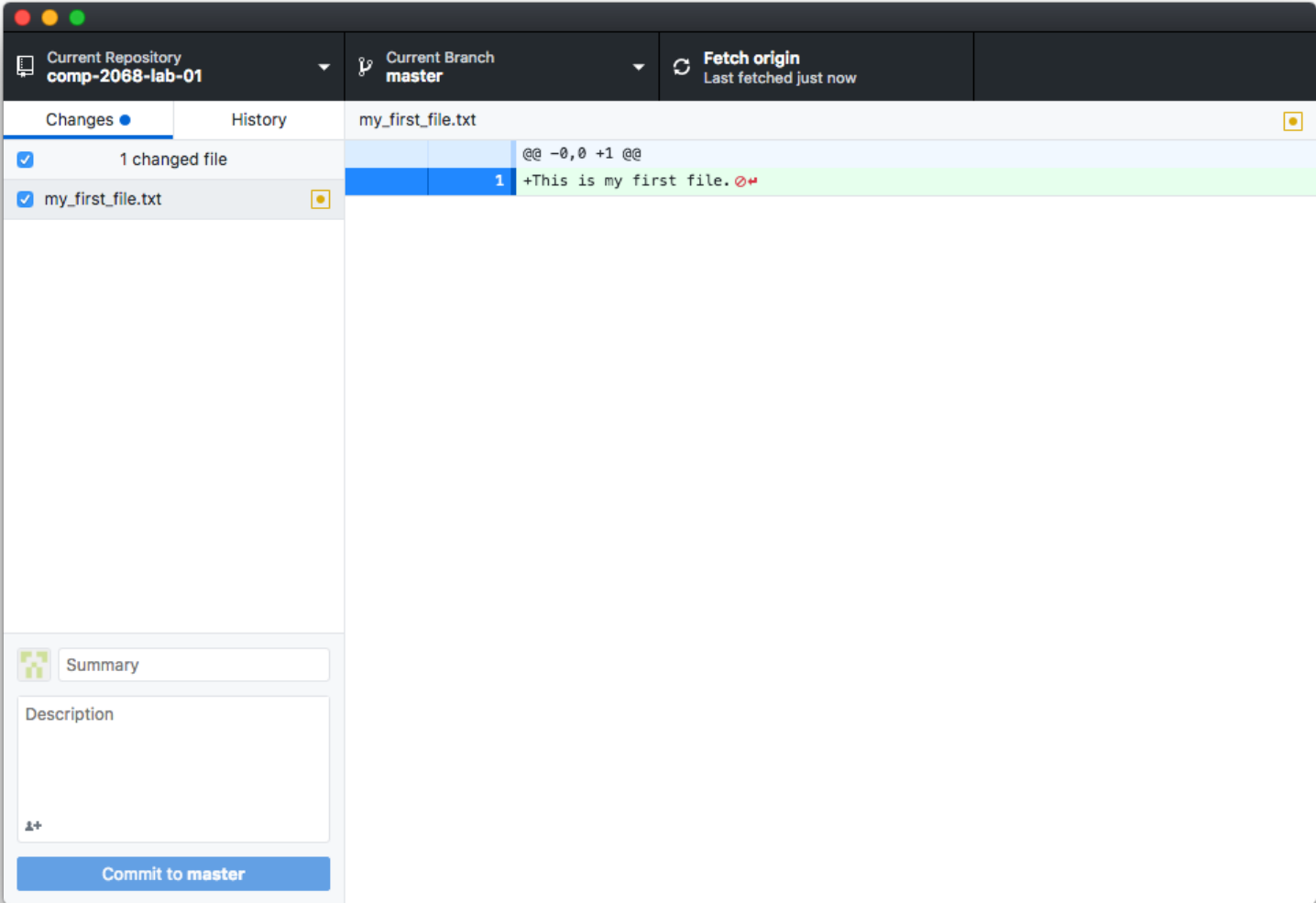
11.



Select **comp-2068-lab-01**, **Choose...** the directory you want to

- clone to, and click **clone**
12. Your repository is now cloned to your computer, and you can begin work. You are currently in the Master branch. This means changes you make will be pushed to this branch.
13. Let's make a small change and go through the steps of committing, pushing, and pulling changes
14. Add the file **my_first_file.txt**
15. Add the text **This is my first file.** and **save** the file.

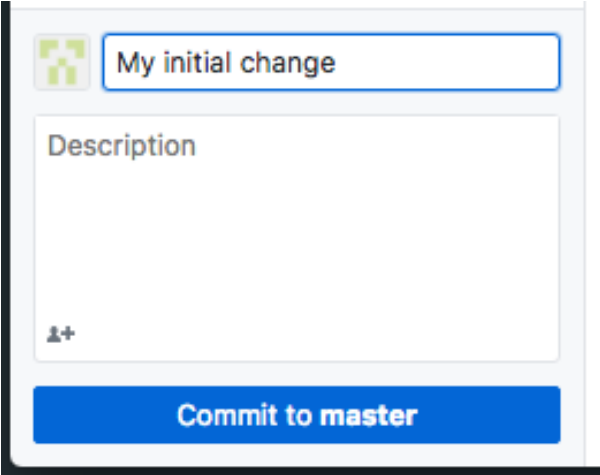
16.



The side

bar shows the files that were changed. The window to the right shows what lines changed within the file.

17.



In the side bar on the left, click on the file and add a summary. This is your

commit message that will be attached to your change

18. Click **Commit to master**
19. You will notice in the top bar you now have the option to **Push origin**

20. I recommend always **pulling** before **pushing** your changes. This will ensure your branch is up to date, and any conflicts that you may have will get resolved. Plus it avoids pesky errors as the result of not pulling branch changes.

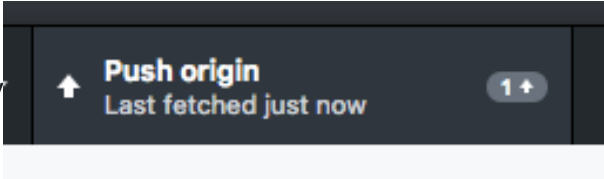
21.



If you click on **Repository** you will see the option to **Pull**. Click **Pull**

22. We had no changes (obviously) in origin, so we are good

23. You can now click **Push origin** to push your changes to the remote repository



24. If we navigate to the GitHub repository we can see our file's changes there

gcsmkinnon Made a slight change		Latest commit 5fdde5b 11 minutes ago
README.md	Initial commit	22 minutes ago
my_first_file.txt	Made a slight change	11 minutes ago

That's it! There is a LOT more to Git than just the above, but that will get you started. I recommend reviewing merging, creating branches, creating pull requests, resetting commits, and removing branches.

Author: Shaun McKinnon

BREAK — 00:15



TUTORIAL: Introducing Node.js — 00:15



Introducing NodeJS

- When Google announced Chrome and its new **V8 JavaScript engine** in late 2008, it was obvious that JavaScript could run faster than before—a lot faster.
- V8's greatest advantage over other JavaScript engines was the compiling of JavaScript code to native machine code before executing it.
- This and other optimizations made JavaScript a viable programming language capable of executing complex tasks.
- Developers decided to try a new idea: non-blocking sockets in JavaScript.
- They took the V8 engine, wrapped it with the already solid C code, and created the first version of **Node.js**.
- **Node.js** uses the event-driven nature of JavaScript to support non-blocking operations in the platform, a feature that enables its excellent efficiency.
- JavaScript is an **event-driven language**, which means that you register code to specific events, and that code will be executed once the event is emitted.
- This concept allows you to seamlessly execute asynchronous code without blocking the rest of the program from running.
- Let's examine how this works with some help from our friends at Code School: <https://www.youtube.com/watch?v=GJmFG4ffJZU>

Installing NodeJS

- browse to nodejs.org download & install
- installer also installs Node Package Manager (npm)
- some popular npm packages we will use:
 - express - primary http library
 - mongodb / mongoose - accessing a MongoDB object db (NOT a relational db, separate package for that)

- on subject of db's, node will work w/rel db's but generally it's not that well suited to it. We will look at nosql db's instead which play very well w/node
- jade (html templating)

What is a Package Manager?

- Package Managers provide a simplified way to add, update, and remove modules to your environment
- Apt, Brew, Yum, Yarn, NPM are examples of package managers
- NPM is the preferred package manager for NodeJS and comes with it when you install Node
- **NPM** has the following main features:
 - A registry of packages to browse, download, and install third-party modules
<https://www.npmjs.com/>
 - You can search for packages that may help your application
 - Packages are solutions that can help mitigate reinventing the wheel
 - A CLI tool to manage local and global packages
 - This is accessible through your terminal or command line tool in your OS

Testing Our Installation

- Open an IDE
- Create a new javascript file
- Add the line
 - `console.log("Hello World");`
- Save the file **helloworld.js**
- Open command line and navigate to the folder that contains **helloworld.js**
- Run **node helloworld.js**
- The console should output **"Hello World"**

Author: Shaun McKinnon

TUTORIAL: Basic Server — 00:10



Our first Node application

Developing Node.js Web Applications

- **Node.js** is a platform that supports various types of applications, but the most popular kind is the development of **web applications**.
- Node's style of coding depends on the community to extend the platform through third-party modules; these modules are then built upon to create new modules, and so on.
- Companies and single developers around the globe are participating in this process by creating modules that wrap the basic Node APIs and deliver a better starting point for application development.
- There are many modules to support web application development but none as popular as the **Connect** module.
- The **Connect** module delivers a set of wrappers around the **Node.js** low-level APIs to enable the development of rich web application frameworks.
- To understand what Connect is all about, let's begin with a basic example of a basic Node web server.
- In your **/COMP_2068/Activities/**, create a file named **basic_server.js**, which contains the following code snippet:
 - ```
var http = require('http');

http.createServer(function(req, res) {
 res.writeHead(200, {
 'Content-Type': 'text/plain'
 });

 res.end('Hello World');
}).listen(3000);

console.log('Server running at http://localhost:3000/');
```
- To start your web server, use your command-line tool, and navigate to your working folder.
- Then, run the node CLI tool and run the server.js file as follows:



\$ node basic\_server

- Now open **http://localhost:3000** in your browser, and you'll see the **Hello World** response.

So how does this work?

- In this example, the **http** module is used to create a small **web server** listening to the 3000 port.
- You begin by requiring the **http** module and use the **createServer()** method to return a new server object.
- The **listen()** method is then used to listen to the 3000 port.
- Notice the callback function that is passed as an argument to the **createServer()** method.
- The callback function gets called whenever there's an **HTTP request** sent to the web server.
- The server object will then pass the **req** and **res** arguments, which contain the information and functionality needed to send back an HTTP response.
- The callback function will then do the following two steps:
  1. First, it will call the **writeHead()** method of the response object. This method is used to set the **response HTTP headers**. In this example, it will set the **Content-Type** header value to **text/plain**. For instance, when responding with HTML, you just need to replace **text/plain** with **html/plain**.
  2. Then, it will call the **end()** method of the response object. This method is used to finalize the response. The **end()** method takes a single string argument that it will use as the **HTTP response body**. Another common way of writing this is to add a **write()** method before the **end()** method and then call the **end()** method, as follows

```
1. res.write('Hello World');
 res.end();
```

Author: Shaun McKinnon

LECTURE: Introducing NPM — 00:15



Using NPM

Installing a package using NPM

Once you find the right package, you'll be able to install it using the command

**\$ npm install <Package Unique Name>**

Installing a module globally is similar to its local counterpart, but you'll have to add the **-g** flag as follows:

**\$ npm install -g <Package Unique Name>**

For example, to locally install Express, you'll need to navigate to your application folder and issue the following command:

**\$ npm install express**

- The preceding command will install the latest stable version of the Express package in your local **node\_modules** folder. Furthermore, NPM supports a wide range of semantic versioning, so to install a specific version of a package, you can use the **npm install** command as follows;

**\$ npm install <Package Unique Name>@<Package Version>**

For instance, to install a prior version of the Express package, you'll need to issue the following command:

**\$ npm install express@3.0.0**

If your want to know what versions are available you can issue the following command:

**\$ npm view <Package Unique Name> versions**

To install the latest stable version:

**\$ npm install <Package Unique Name>**

This will always install the latest stable version available

Removing a package using NPM

To remove an installed package, you'll have to navigate to your application folder and run the following command:

**\$ npm uninstall <Package Unique Name>**

NPM will then look for the package and try to remove it from the local **node\_modules** folder.

To remove a global package, you'll need to use the **-g** flag as follows:

**\$ npm uninstall -g <Package Unique Name>**

## Updating a package using NPM

To update a package to its latest version, issue the following command:

**\$ npm update <Package Unique Name>**

NPM will download and install the latest version of this package even if it doesn't exist yet.

To update a global package, use the following command:

**\$ npm update -g <Package Unique Name>**

## Managing Dependencies Using package.json File

- Installing a single package is nice, but pretty soon, your application will need to use several packages, and so you'll need a better way to manage these **package dependencies**.
- For this purpose, NPM allows you to use a configuration file named **package.json** in the root folder of your application.
- In your package.json file, you'll be able to define various metadata properties of your application, including properties such as the **name**, **version**, and **author** of your application.
- This is also where you define your **application dependencies**.
- The package.json file is basically a JSON file that contains the different **attributes** you'll need to describe your application properties.
- An application using the latest Express and Grunt packages will have a **package.json** file as follows:

```
{
 "name" : "MEAN",
 "version" : "0.0.1",
 "dependencies" : {
 "express" : "latest",
 "grunt" : "latest"
 }
}
```

## Creating a Package Dependency File

While you can manually create a package.json file, an easier approach would be to use the npm init command. To do so, use your command-line tool and issue the following command:

**\$ npm init**

NPM will ask you a few questions about your application and will automatically create a new **package.json** file for you.

A sample process should look similar to the following screenshot:

## Installing the Package Dependencies

After creating your package.json file, you'll be able to install your application dependencies by navigating to your application's root folder and using the npm install command as follows:

**\$ npm install**

NPM will automatically detect your package.json file and will install all your application dependencies, placing them under a local node\_modules folder.

An alternative and sometimes better approach to install your dependencies is to use the following npm update command:

**\$ npm update**

This will install any missing packages and will update all of your existing dependencies to their specified version.

## Updating the Package Dependencies

Another robust feature of the **npm install** command is the ability to install a new package and save the package information as a dependency in your **package.json** file.

This can be accomplished using the **--save** optional flag when installing a specific package.

For example, to install the latest version of Express and save it as a dependency, you can issue the following command:

```
$ npm install express --save
```

Author: Shaun McKinnon

BREAK — 00:15



Introducing Modules — 00:10



A quick look at modules

Modules

- Modules are essentially libraries
- There are Core Modules
  - Such as the HTTP module
  - Core modules are libraries that are available with the Node installation
- There are Third-Party Modules
  - These are modules you install using NPM
  - Express would be considered a third-party module
- There are custom modules
  1. Create a new file called **random\_insult.js**
  2. Place the following in that file

```
3. var https = require('https');

exports.get_insult = function(callback) {
 var body = '';

 var req = https.get('https://insult.mattbas.org/api/insult', function(res) {
 res.on('data', function (chunk) {
 body += chunk;
 callback(body);
 });
 });
};
```

4. We will have to modify our server.js file to accommodate our new module a little
5. Add this line after the **http module**
6. **var random\_insult = require('./random\_insult');**
7. replace **res.end('Hello World');** with

```
8. random_insult.get_insult(function (insult) {
 res.end(insult);
});
```

9. Navigate in your browser to **localhost:3000** and let the app insult you

Author: Shaun McKinnon

TUTORIAL: Connect Module & Middleware — 00:20



The Connect Module

- **Connect** is a module built to support interception of requests in a more modular approach.



- In the first web server example, you learned how to build a simple web server using the **http** module.
- If you wish to extend this example, you'd have to write code that manages the different HTTP requests sent to your server, handles them properly, and responds to each request with the correct response.
- **Connect** creates an API exactly for that purpose – it uses a modular component called **middleware**, which allows you to simply register your application logic to **predefined** HTTP request scenarios.
- Connect **middleware** are basically callback functions, which get executed when an HTTP request occurs.
- The **middleware** can then **perform some logic, return a response**, or call the next registered middleware.
- While you will mostly write **custom middleware** to support your application needs, Connect also includes some common middleware to support **logging, static file serving**, and more.
- The way a Connect application works is by using an object called **dispatcher**.
- The **dispatcher** object handles each HTTP request received by the server and then decides, in a cascading way, the order of middleware execution.
  - If you have experience with MVC, you will quickly recognize that the **dispatcher** is a **router** in other frameworks.
- In the next section, you'll create your first **Express** application, but **Express** is based on **Connect's** approach, so in order to understand how **Express** works, we'll begin with creating a **Connect** application.
- However, **Connect** isn't a core module, so you'll have to install it using **NPM**.
- To do so, use your command-line tool, and navigate to your working folder. Then execute the following command:

**\$ npm install --save connect**

- **NPM** will install the connect module inside a **node\_modules** folder, which will enable you to **require** it in your application file.
- In your working folder, create a file named **connect\_server.js** that contains the following code snippet:

```
var connect = require('connect');
var app = connect();
app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

- As you can see, your application file is using the **connect** module to create a new **web server**.
- To run your Connect web server, just use Node's CLI and execute the following command:

**\$ node server**

- Node will run your application, reporting the server status using the **console.log()** method.
- You can try reaching your application in the browser by visiting **http:// localhost:3000**.
- However, you should get a response:

**Cannot GET /**

- What this response means is that there isn't any **middleware** registered to handle the **GET HTTP** request.

## Connect Middleware

- Connect **middleware** is just **JavaScript function** with a unique signature.
- Each middleware function is defined with the following three arguments:
  - **req:** This is an object that holds the HTTP **request** information
  - **res:** This is an object that holds the HTTP **response** information and allows you to set the response properties
  - **next:** This is the next middleware function defined in the ordered set of Connect middleware
- When you have a **middleware** defined, you'll just have to register it with the **Connect** application using the **app.use()** method.
- Let's revise the previous example to include your first **middleware**.
- Change your **server.js** file to look like the following code snippet:

```
var connect = require('connect');
var app = connect();

var helloWorld = function(req, res, next) {
 res.setHeader('Content-Type', 'text/plain');
 res.end('Hello World');
};

app.use(helloWorld);

app.listen(3000);

console.log('Server running at http://localhost:3000/');
```

- Then, start your **connect** server again by issuing the following command in your command-line tool:

**\$ node server**

- Try visiting **http://localhost:3000** again.
- You will now get a response similar to that in the following screenshot:

**Hello World**

## Mounting Connect Middleware

- As you may have noticed, the **middleware** you registered responds to **any request** regardless of the request path.
- This does not comply with modern web application development because responding to different paths is an integral part of all web applications.
- Fortunately, Connect middleware supports a feature called **mounting**, which enables you to determine which request path is required for the middleware function to get executed.
- **Mounting** is done by adding the **path** argument to the **app.use()** method.
- To understand this better, let's revisit our previous example.
- To understand this better, let's revisit our previous example -- modify your **server.js** file to look like the following code snippet:

```
var connect = require('connect');
var random_insult = require('./random_insult');
var app = connect();

var logger = function(req, res, next) {
 console.log(req.method, req.url);
 next();
};

var helloWorld = function(req, res, next) {
 res.setHeader('Content-Type', 'text/plain');
 res.end('Hello World');
};

var insultMe = function(req, res, next) {
 res.setHeader('Content-Type', 'text/plain');
 random_insult.get_insult(function (insult) {
 res.end(insult);
 });
};

var goodbyeWorld = function(req, res, next) {
 res.setHeader('Content-Type', 'text/plain');
 res.end('Goodbye World');
};

app.use(logger);
app.use('/hello', helloWorld);
app.use('/goodbye', goodbyeWorld);

app.listen(3000);

console.log('Server running at http://localhost:3000/');
```

Author: Shaun McKinnon

## LAB: Cloud-based Hosting - Heroku — 00:20



1. Navigate to <https://signup.heroku.com/login>
2. Fill in the information
3. Choose '**Node.js**' for the '**Primary Development Language**'
4. Click '**Create Free Account**'
5. Once you have confirmed your email and signed in:
  1. Click **Create new app**
  2. Name it your **lab-01-comp2068-STUDENTID**
  3. Click **Create app**
  4. Under the tab **Deploy**
    1. Scroll down till you see the section **Deploy using Heroku Git**
    2. We have a choice
      1. We can push everything to GitHub and deploy from there
      2. We can copy everything to DropBox and deploy from there
      3. We can push to Heroku's repository and deploy from there
    3. We can always change our deploy method at a later date. For now, let's just use Heroku
5. Navigate to <https://devcenter.heroku.com/articles/heroku-cli>
  1. Download and install for your OS
  2. Follow the installer instructions and install
  3. If you are in Windows
    1. Ensure **Set PATH to heroku** is checked
  4. Once you have it installed
    1. Close your CLI tool and reopen it
    2. Navigate to our COMP2068/Labs/Lab\_01/ folder
    3. Copy the **server.js** file over to this folder
    4. Run **npm init** to initialize our **package.json** file
    5. Run **npm install connect** to install the **Connect Module**
    6. In the command line follow along
      1. Login to heroku

**\$ heroku login**

2. Initialize your GIT

**\$ git init**

3. Pair up Heroku with your GIT

**\$ heroku git:remote -a lab-01-comp2068-STUDENTID**

4. Add your changes, commit, and push

**\$ git add .**

**\$ git commit -am "Pushing for the first time."**

**\$ git push heroku master**

5. Heroku will move your changes and build your environment for you automatically (if it can find a build)

6. To open our application we need to run

**\$ heroku open**

6. Your app didn't work
  1. Let's see why

1. In the CLI you can output the Heroku errors from the log file

```
// this will show you the most recent logs since the last action
$ heroku logs
```

```
// this will stream the logs
$heroku logs --tail
```

2. It appears that our app crashed

```
2018-05-06T19:33:36.166730+00:00 heroku[web.1]: Restarting
2018-05-06T19:33:36.167314+00:00 heroku[web.1]: State changed from up to starting
2018-05-06T19:33:37.190899+00:00 heroku[web.1]: Stopping all processes with SIGTERM
2018-05-06T19:33:37.281826+00:00 heroku[web.1]: Process exited with status 143
2018-05-06T19:33:38.890012+00:00 heroku[web.1]: Starting process with command `npm start`
2018-05-06T19:33:41.725393+00:00 heroku[web.1]: Process exited with status 1
2018-05-06T19:33:41.642817+00:00 app[web.1]: npm ERR! missing script: start
```

3. When Heroku attempted to spin up our application, it called a script **npm start**
  1. This script should be defined in our **package.json** file
  2. There are several scripts that we can define here
    1. Scripts help us perform actions before our application starts
    2. These scripts can be preprocessors, notifiers, loggers, special startup operations, etc...
  3. We need to add the **start script** to our **package.json** file

```
"scripts": {
 "start": "node connect_server.js"
}
```

4. This calls our command for getting node to execute our express server script

7. Let's try that again

1. Add your changes, commit, and push

```
$ git add .
$ git commit -am "Pushing for the second time."
$ git push heroku master
```

8. Your app didn't work

1. Let's see why
  1. In the CLI you can output the Heroku errors from the log file

```
2018-05-06T19:45:50.796538+00:00 heroku[web.1]: Starting process with command `npm start`
2018-05-06T19:45:53.068278+00:00 app[web.1]:
2018-05-06T19:45:53.068292+00:00 app[web.1]: > helloworld@1.0.0 start /app
2018-05-06T19:45:53.068293+00:00 app[web.1]: > node express_server.js
2018-05-06T19:45:53.068295+00:00 app[web.1]:
2018-05-06T19:45:53.360734+00:00 app[web.1]: Server running at http://localhost:3000
2018-05-06T19:46:51.359500+00:00 heroku[web.1]: Error R10 (Boot timeout) -> Web process failed to
bind to $PORT within 60 seconds of launch
```

2. So Heroku executed the script **npm start**
3. However, it looks like it blew up when it attempted to run the server on **localhost:3000**
4. The truth is we don't know what port Heroku wants to run on. In fact we don't really want to guess in case they ever change it. In addition, we want to be able to run it locally still so we can continue to develop.
  1. So how do get the best of both worlds?
  2. The wonderful **||** logical operator

```
app.listen(process.env.PORT || 3000);
```

3. This is a shorthand version of a ternary:

```
app.listen((process.env.PORT) ? process.env.PORT : 3000);
```

4. Basically this says if the **environment variable** is missing, use port **3000** instead

9. Let's try that one last time

1. Add your changes, commit, and push

```
$ git add .
$ git commit -am "Pushing for the third time."
$ git push heroku master
```

2. Run **heroku open**

**Hello World**

10. **YAY WE DID IT!!!**

## Heroku

- We will require an app for everything we want to deploy
- Heroku is free unless you exceed their traffic limit, so don't
- Test in Development ALWAYS before deploying to Heroku
- Heroku is only a suggestion
  - You are welcome to deploy to where ever you choose

Author: Shaun McKinnon

## Submitting Our New Lab to Blackboard — 00:05



Copy and paste the link into the submission and click submit!

**CONGRATULATIONS!!! LAB 1 IS COMPLETE!!!**

Author: Shaun McKinnon