Simple Node Server and Deploying to Heroku

Blocking VS Non-Blocking Code

What is Blocking and Non-Blocking Code?

Blocking (**synchronous**) code is code that prevents execution (stops the world) of other code till it has completed its operation. **Most** operations are blocking in JavaScript, but there are times where operations are **non-blocking**.

Non-blocking (**asynchronous**) code is code that **does not** prevent the execution of other code. It will break it's operations into events and add them to the stack as needed. This allows other events to complete their executions while the asynchronous code catches up. The following are examples of when asynchronocity occurs:

Asynchronous Operations

- HTTP requests (the response is handled asynchronously to avoid waiting for operations to be completed on the foreign server)
- File requests (these can be blocking, but usually file handling is asynchronous due to the unknown size or potential needs of the file)
- setInterval and setTimeout (these are asynchronous so they can perform logic repeatedly or after a delay. If they were synchronous you would be stuck in a loop or a queue till they completed)
- Event registered functions (any function that is bound to an event like **click**, **mousedown**, **focus**, etc are asynchronous so they can be handled once the event has occurred)

Let's look at an example of blocking and non-block code:

```
// This function takes one argument and console logs a statement
function sayHi(name) {
  console.log(`Hello ${name}!`);
}

// The below code is blocking. Each line will happen synchronously
sayHi(`Keval`);
sayHi(`Devon`);
sayHi(`Connor`);

// The below code is asynchronous. The lines sayHi(`Darryl`) and sayHi(`Ilia`)
// will execute before the setInterval lines
setInterval(sayHi, 2500, `Michael`);
setInterval(sayHi, 5000, `Gagandeep`);
sayHi(`Darryl`);
sayHi(`Ilia`);
```

Callbacks

Callbacks are essential to asynchronous programming but not exclusive to it. Simply put, ***a callback is a function that is passed as an argument to another function to be called at a later time***. Callbacks are probably one of the more confusing programming paradigms for beginning programmers, but really they're quite easy to use.

Let's look at an extremely basic example of a callback function:

```
// Our callback function that we'll pass as an argument later to our greeting function
function farewell(name) {
  console.log(`Goodbye ${name}.`);
}

// Creating parameters for our name and callback function
function greeting(name, callback) {
  console.log(`Hello ${name}.`);

  callback(name);
}

// Calling our greeting function, passing in our name and our callback arguments
greeting(`Shaun McKinnon`, farewell);
```

The above example illustrates a very basic example of using a callback. The benefits of a callback are as follows:

- Code separation and modularity (the **greeting** function doesn't need to care about what the **callback** function does)
- Priority execution (the **callback** can be executed where needed which is more beneficial to asynchronous operations)
- Provides an order-of-operation structure where one function can invoke another of it has executed

Using Callbacks for Asynchronous Operations

Callbacks are often used to combat asynchronocity. They allow us to execute code at the correct moment in time where we need to. For example, because an HTTP request can take an unknown amount of time, it is optimal to that HTTP requests are asynchronous. The issue lies in not knowing when to return our response. Let's take a look at the example below:

```
// https is a module that is apart of the core Node API
const https = require('https');

// Getting an insult from the infamous Matt Bas Insult API
const getInsult = function (callback) {
  const url = `https://insult.mattbas.org/api/insult`;

  https.get(url, function (client) { // callback 1
    client.on(`data`, function (data) { // callback 2
      callback(data.toString()); // callback 3
    });
  });
}

// We call getInsult and pass an anonymous function as a callback
// so we can utilize the returned data object
getInsult(function (insult) { // callback 4
  console.log(insult);
});
```

The above is also a good example of **callback hell** a term given to code that uses a large amount of callbacks. Lot's of callbacks render the code difficult to read, as you can see. Let's look at what each callback is doing:

1. The first callback is used to return the client agent to our function body from the https request
2. The second callback is used to capture the data once the **data** event has been executed by the client
3. The third callback is used to pass our data to a new function scope
4. The fourth callback is used as an argument to our getInsult function. This allows us to capture and use the returned **data**

Promises

Promises

Promises are meant to help callbacks be more readable. By encapsulating our function logic in a promise, we can then provide 2 new methods helping our function to make more sense: **.then** and **.catch**.

.then will execute regardless of what our response is from our promise. If we **resolve** our promise (meaning our function was successful), **.then** will be called. If we **reject** our promise, then **.then** will also be called. If want to instead funnel our rejection pass the **.then** we can either not reject it, or we can use **.catch** to aid in handling the error.

You may be asking where do **reject** and **resolve** come from? Both of these are callback functions that get passed to a **new Promise**. Let's convert our last example into a new a promise wrapped function.

In our IDE, we'll use the following code to recreate our last example using promises:

```
// https is a module that is apart of the core Node API
const https = require('https');

// Getting an insult from the infamous Matt Bas Insult API
const getInsult = function (callback) {
  const url = `https://insults.mattbas.org/api/insult`;

  // a new promise takes a callback as an argument that
  // has 2 callbacks as parameters: resolve and reject
  return new Promise(function (resolve, reject) {
    // We make our request to Matt Bas' url
    const req = https.get(url, function (client) {
      // When we have the data, we resolve
      client.on(`data`, function (data) {
        // Think of this as our successful callback
        resolve(data.toString());
      });

    });

    // If the request breaks, we can catch the error
    req.on(`error`, function (error) {
      // Think of this as our fail callback
      reject(error);
    });
  });
}

// Calling getInsult using our new methods
// .then will catch everything, but because we have
// .catch, we can ensure errors go to .catch and our
// successes go to .then
getInsult()
  .then(function (data) {
    console.log(`Resp:`, data);
  })
  .catch(function (error) {
    console.log(`Error:`, error);
  });
```

Promises can also be chained together allowing for them to execute in a sequential format. Promises are fantastic, but **Async/Await** makes life even better.

QUIZ

The quiz is 5 questions and you have 2 attempts. **ATTEMPT THIS ON YOUR OWN AS IT SHOWS YOUR UNDERSTANDING!**

BREAK

Introducing ES6 Syntax

ES6 Cheat Sheet

What is ES6?

ES6 stands for ECMA Script 6. ECMA is an organization that standardizes information. Because Javascript is implemented differently in various browsers, there has to be a standard and specification that they must follow to ensure the developers have a

common environment to work with. However, updating browsers to support every new standard can be costly, and therefore browsers will adopt when they choose. They also aren't required to update old browsers with the new standard, so it is very possible that new code will not work properly in older browsers.

Transpilers to the Rescue!

Transpilers allows us to use newer code functionality, regardless of support for that functionality, by transpiling the new code into legacy compatible code. This means that older browsers, and browsers that are taking their time, will still work.

However, we now have an extra step in our process. We must transpile in order to ensure we are cross browser compliant. That isn't a big deal considering we can add transpilers to our build process, which will result in us always having cross-browser compliant code, regardless of what we code.

Handy ES6 Stuff

We already showed how we can use **let** and **const** instead of **var**. However, ES6 introduces many other really cool features that make writing Javascript simple:

```
// no longer need IIFEs (Immediately Invoked Function Expressions) thanks to block scope :)
(function () {
  var myScopedVar = "Bob";
}());

// now becomes
{
  let myScopedVar = "Bob";
}
```

IIFEs are commonly used to maintain block scope. This aids in avoiding namespace collisions.

Arrow Functions are very handy, and not just because they're shorter to write than full functions. They maintain also maintain the scope of the keyword **this**. Many languages use the keyword **this** to represent the current scoped object. It gives you access to defined properties and methods and is fundamental to object oriented programming. However, in Javascript **this** changes scope to reference its current function container. That means if you call **this** from within a callback it is referring to its callback and not the surrounding function.

Many programmers have overcome this issue by assigning **this** to **that** or **self** or some other arbitrary variable that will maintain its scope for them. This also isn't ideal as you want **this** to be contextual. **Arrow Functions** do not have a sense of **this** in their own context. They pass **this**through with containing context. We definitely need an example:

```
function Person() {
  var that = this;
  that.age = 0;

  setInterval( function growUp() {
    // The callback refers to the `that` variable of which
    // the value is the expected object.
    that.age++;
    console.log( that.age );
  }, 1000 );
}

let peep = new Person();
```

The above may seem odd but is essentially a class with a constructor. The constructor is setting the property of **that** equal to **this** so it will maintain context when given to the callback function **growUp**, otherwise **growUp** will throw a reference error when we attempt to access **this.age** as it isn't present within its context.

Below we attempt the same example, but instead use the **arrow function**:

```
function Person(){
  this.age = 0;

  setInterval(() => {
    this.age++; // |this| properly refers to the person object
    console.log( this.age );
  }, 1000);
}

let peep = new Person();
```

In the above example, we are passing an **arrow function** as the callback. The **arrow function** maintains the context of **this** throughout the body of the function. Much cleaner. Much easier to read.

Arrow functions can be a little confusing because you will see them in different syntaxes. Below is a table showing the various ways you'll see them formatted and what they mean:

Function Syntax	What's it's Doing
`(param1, param2) => { return param1 + param2; }`	The () holds the parameters. The arrow then passes those parameters into our function block, where we are returning them concatenated.
`param1 => { return param1; }`	If you have only 1 parameter, you can forgo the parenthesis.
`param1 => param1;`	If you only have one line to execute you can do it inline. It is implied you want to return it though. The example will pass param1 in as an argument to the function block and then the function block will return param1.
`() => { return "hi"; }`	If you have no parameters, you must use a set of parenthesis to signify it's an arrow function.
`() => "hi";`	This will return the value "hi".
`(() => { return "hi"; })();`	This is an instantly invoked function execution, or IIFE (iffy) for short. This function will automatically run and not be executed. It is most commonly seen when either encapsulating scope or creating an async operation.

Mapis very cool method of array. It allows us to walk through the array and perform operations on each element:

```
['me', 'you', 'them'].map(n => console.log(n.length))
```

The above is an inline expression. Below is a full operation:

```
['me', 'you', 'them'].map(n => {
  console.log( n.length )
})
```

Template Literals (string interpolation) are in so many languages. Some even do it right (I'm lookin' at you Ruby). Template Literals allow you to embed expressions in strings. These expressions will be parsed within the string, making it very easy to embed variables, calculations, function calls, method calls, properties, etc...

```
let name = "Shaun McKinnon"
let age = 39

let text = `${name} is ${age - 20} years old.`
console.log( text );
```

Destructuring is one my most favourite constructs available in ES6. It makes working with arguments more readable. The goal is to get to a point where commenting isn't necessary because the syntax is understandable. **Destructuring**allows us to extract values and then immediately store them in variables that have a more meaningful syntax:

```
let [day, month, year] = [21, 5, 2018]
console.log(day, month, year)
```

The above **destructures** the values in the array. We can now access those values with the more contextual **day**, **month**, and **year** variables. We can literally call the variables whatever we want (following variable naming rules).

```
let Student = {
  name: 'Shaun',
  id: 456321,
  age: 39
}

function myFunc ( {id, name, age} = options ) {
  console.log( name, id, age )
}

myFunc( Student )
```

In the above code, we **destructure** the options and pull out the name, id, and age from the passed object. Keep in mind, we MUST pass in an object that has those values, or we will wind up with variables being undefined.

Also notice that the order of the destructuring isn't the same as the order of the keys in the object. This is important because when destructuring an object is uses the keys to match up the variables. That means we don't have the flexibility in naming the variables like we did in the array. They MUST be the same as the key names.

You can read more about desctructuring. It is a very power feature that helps make code a lot more human readable.

Classes

Classes is so important it gets its own heading. Prior to ES6, we had to this fun:

```
// This is basically our class and constructor
function Student ( id, name, age ) {
  // these are the properties
  this.id = id;
  this.name = name;
  this.age = age;
}

// This is a method of the class
Student.prototype.output = function () {
  console.log( this.id, this.name, this.age );
};

// here's instantiation of the class
( new Student( 1234, 'Shaun', 39 ) ).output();
```

Clear as the windows in a sketchy bar, right? This is prototyping. A very different approach to building 'classes' in Javascript.
If you're looking for a more traditional version of classes, look no further than our friend, ES6:

```
class Student {
  constructor ( id, name, age ) {
    this.id = id
    this.name = name
    this.age = age
  }

  output () {
    console.log( this.id, this.name, this.age )
  }
}

( new Student( 1234, 'Shaun', 39 ) ).output()
```

Annnnnnd, if you want inheritance, you can use the keyword **extends**:

```
class StudentGroup extends Student {}
```

ES6 and ES7 and ES8 are making huge changes to how we code Javascript. These massive changes are going to make a more approachable language for the layman and will definitely increase its popularity.

Lastly (this is a bonus) I have shown you the ES6 for/of and for/in loops. The for/of loop iterates over the elements in an array. The for/in loop iterates over the keys in an object. that's often not that usable, as you are more likely going to utilize the values, not the keys. Below is an example of how you can output the values:

```
for (let [key, value] of Object.entries(obj)) {
  console.log(`${key}: ${value}`);
}
```

In the example we use **Object.entries** which converts our object to a nested array of key/values. We then use **destructuring** to cast each value into a key and a value variable.

BREAK

Introducing NPM (Node Package Manager)

What is a Package Manager?

- Package Managers provide a simplified way to add, update, and remove modules (vendor libraries) to your environment
- Apt, WGET, Brew, Yum, Yarn, RubyGems, Composer, and 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

So how do I get NPM?

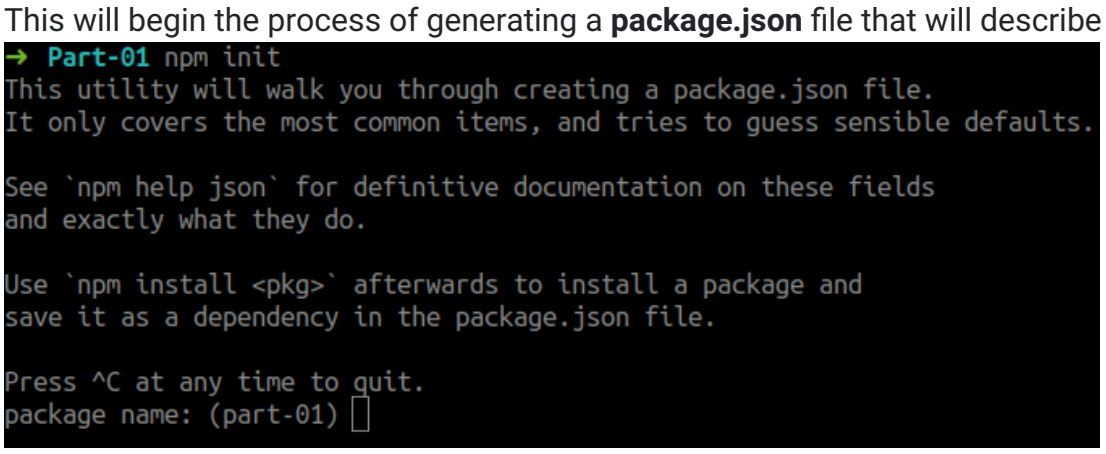
- Installed when you install NodeJS
- If you don't have NodeJs? Install NodeJS

Using NPM

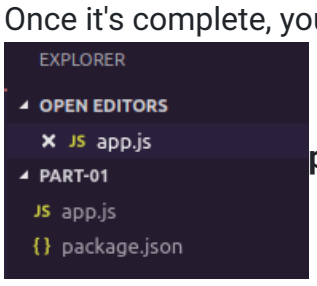
In your project directory in terminal/commandprompt/powershell

```
npm init
```

This will begin the process of generating a **package.json** file that will describe your application:



Once it's complete, you will have a new file in your project directory:



package.json describes your application. It provides a name for your application, the version number, your entry point file (**app.js**) and information about who created it. The **2 most imporant sections** are the **dependencies** and **scripts**

sections. The **scripts** section provides easy to use custom **CLI (command line interface)** commands that you can use to run tests, builds, or just the app in general. The **dependencies** list the **node modules** we have installed, and their information including their version numbers.

An example package.json file

```
{
  "name": "part-01",
  "version": "1.0.0",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {},
  "description": ""
}
```

When we first run **npm init** the **package.json** file will be quite sparse. As we install modules, our dependencies will grow. In addtition we'll also need to add a new script to run our application.

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 the latest major version for the Express package, you'll need to issue the following command:

```
npm install express-current-version
```

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>
```

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.

A super simple Node server using Express

What is Middleware?

- Middleware glues together complex modular systems
- This could be
  - A messaging system
  - An event system
  - Task/Job scheduler
  - A router
  - An ORM or ODM
- Middleware can also be considered as a communication or interpreter system that allows output from one system to be work as acceptable input on another system

Why is Express Considered Middleware?

- Our user makes a request to our server which is listening
- Express intercepts the request and evaluates the request packet the user is sending
  - For example, Express evaluates the request path
  - From that, Express determines which registered middleware to execute
  - The interesting thing here is that the middleware is calling more middleware where we can perform operations based on the request packet
- The first intercept is known as routing

A Quick 2 Minutes on Express

- Express is a Node.js web server project
  - As stated earlier, the community is driving the web application side of Node.js
  - Express is one of those projects
- Express is a bit of sugar on top of Node.js' createServer
- Declarative routing
  - Routing is the process of analyzing, modifying, and delivering to the appropriate end-point

Building an Express Web Application

- The **Express framework** is a small set of common web application features, kept to a minimum in order to maintain the **Node.js** style.
- It is built on top of **Connect** and makes use of its middleware architecture.
- Its features extend **Connect** to allow a variety of common web applications' use cases, such as the inclusion of **modular HTML template engines**, extending the response object to support various data format outputs, a routing system, and much more.
- In order for us to use Express, we should initialize our package dependency file and install the Express module:

1. Create a new project folder on your computer called **BloggingProject**
2. Navigate (in your command prompt/terminal) to your **BloggingProject** folder
3. Once you are there, run the following commands:

```
npm init
npm install --save express
```

4. After creating your **package.json** file and installing express as one of your dependencies, you can now create your first **Express** application.
5. Open your **BloggingProject** folder in your **IDE**
6. Create a new file called **app.js**
7. Add the following lines:

Our very simple server with 2 routes/responses

```
// Our imported libraries
const express = require('express');

// Assigning Express to an app constant
const app = express();

// Creating our first route which is looking for requests from http://localhost:4000/
app.get(`/`, (req, res) => { // res is the response object and req is the request object
  // Our response
  res.send(`Home`);
});

// Creating our first route which is looking for requests from http://localhost:4000/
app.get(`/greeting`, (req, res) => {
  // Our response
  res.send(`Hey 'dere world!`);
});

// Starting our server on port 4000
app.listen(4000, () => console.log('Listening on 4000'));
```

Understanding Our Application

- **Express** presents three major objects that you'll frequently use.
  - The **application object** is the instance of an Express application you created in the first example and is usually used to configure your application.
  - The **request object** is a wrapper of Node's HTTP request object and is used to extract information about the currently handled HTTP request.
  - The **response object** is a wrapper of Node's HTTP response object and is used to set the response data and headers.
- The **application object** contains the following methods to help you configure your application:
  - **app.set(name, value)**: This is used to set environment variables that Express will use in its configuration.
  - **app.get(name):** This is used to get environment variables that Express is using in its configuration.
  - **app.engine(ext, callback):** This is used to define a given template engine to render certain file types, for example, you can tell the **EJS template engine** to use HTML files as templates like this: **app.engine('html', require('ejs').renderFile).**
  - **app.locals:** This is used to send application-level variables to all rendered templates.
  - **app.use([path], callback):** This is used to create an Express middleware to handle HTTP requests sent to the server. Optionally, you'll be able to mount middleware to respond to certain paths.
  - **app.VERB(path, [callback...], callback):** This is used to define one or more middleware functions to respond to HTTP requests made to a certain path in conjunction with the HTTP verb declared. For instance, when you want to respond to requests that are using the GET verb, then you can just assign the middleware using the app.get() method. For POST requests you'll use **app.post()**, and so on.
  - **app.route(path).VERB([callback...], callback):** This is used to define one or more middleware functions to respond to HTTP requests made to a certain unified path in conjunction with multiple HTTP verbs. For instance, when you want to respond to requests that are using the GET and POST verbs, you can just assign the appropriate middleware functions using **app.route(path).get(callback).post(callback)**.

- **app.param([name], callback):** This is used to attach a certain functionality to any request made to a path that includes a certain routing parameter. For instance, you can map logic to any request that includes the userId parameter using **app.param('userId', callback)**.

Deploying to Heroku

Signing up to Heroku

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 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 **BloggingProject** folder
            3. 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

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:4000**
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
    3. ```
       app.listen((process.env.PORT || 4000), () => console.log('Listening on 4000'));
       ```
    4. Basically this says if the **environment variable** is missing, use port **4000** instead
7. 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

8. **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
       ```