

CSIS 3380 – Nodejs Notes

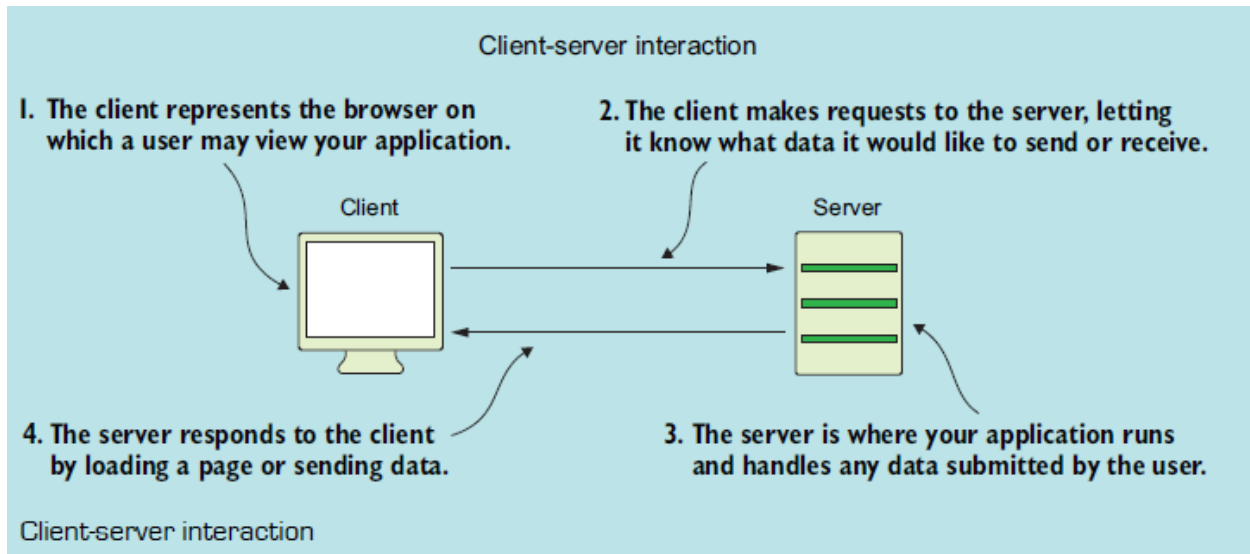
Node.js is a platform for interpreting JavaScript code and running applications.

Node.js is built with Google Chrome's JavaScript engine, and it's considered to be powerful and able to support JavaScript as a server-side language.

Client-side versus server-side

As a general overview, web development can largely be broken into two categories:

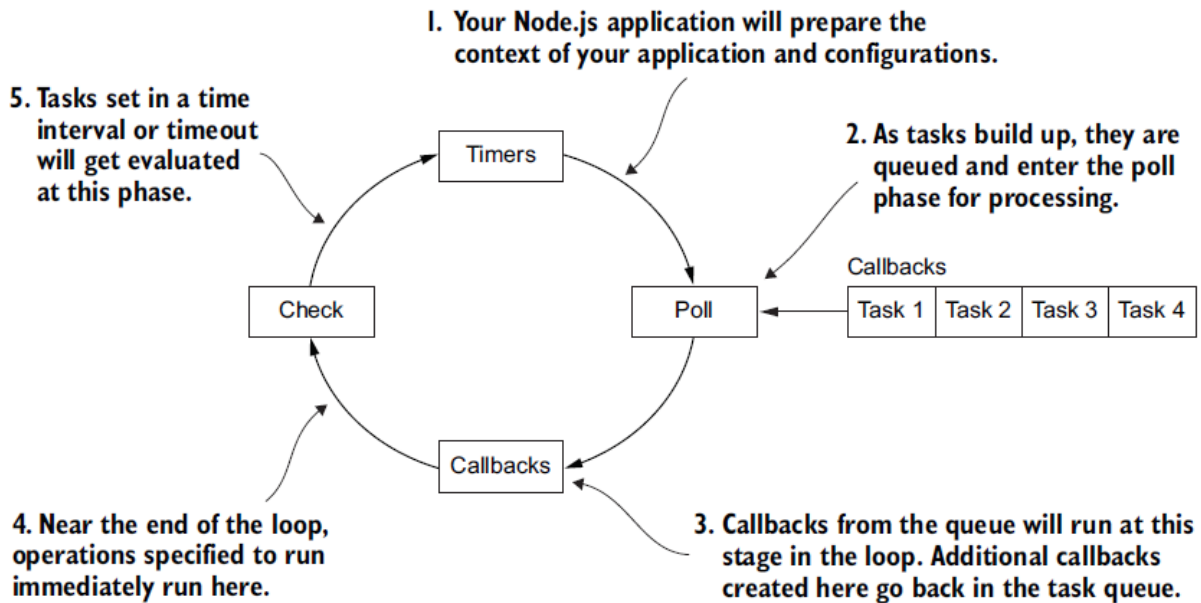
- ☐ *Client-side*—(front-end) refers to the code you write that results in something the user sees in his web browser. Client-side code typically includes JavaScript used to animate the user experience when a web page is loaded.
- ☐ *Server-side*—(back-end) refers to code used for application logic (how data is organized and saved to the database). Server-side code is responsible for authenticating users on a login page, running scheduled tasks, and even ensuring that the client-side code reaches the client.



You'll hear these two terms used a lot in application development, and because JavaScript has been used for both types of development, the line separating these two worlds is disappearing. *Full stack* development, using JavaScript, defines this new development in which JavaScript is used on the server and client, as well as on devices, hardware, and architectures it didn't exist on before.

Node.js operates on an event loop using a single thread. A *thread* is the bundle of computing power and resources needed for the execution of a programmed task. In most other software, multiple tasks are matched and handled by a pool of threads that the computer can offer at the same time (concurrently). Node.js, however, handles only one task at a time and uses more threads only for tasks that can't be handled by the main thread. Most applications that don't require computationally intensive tasks

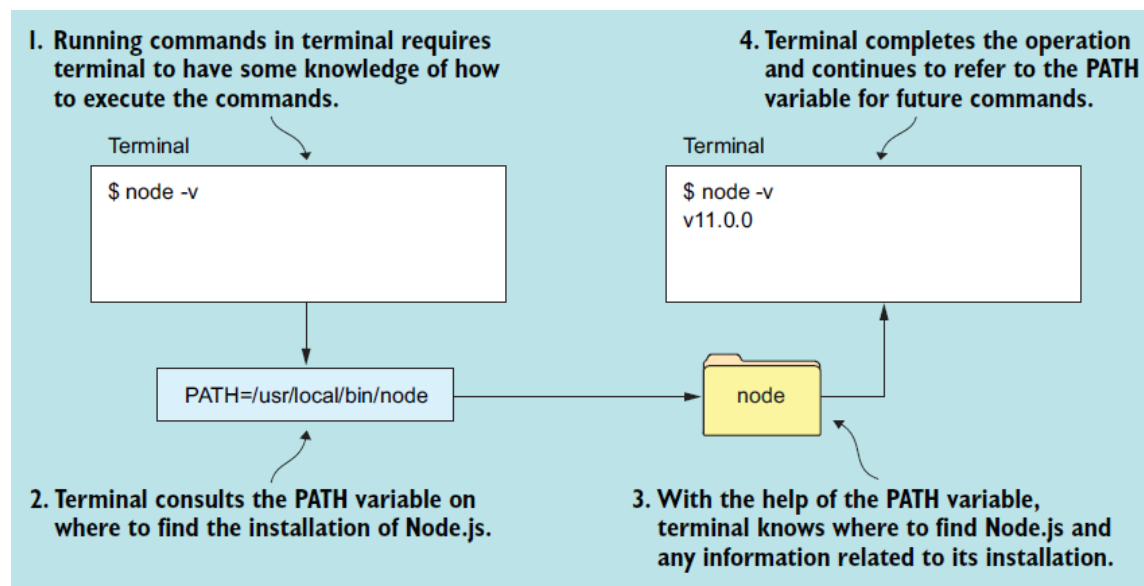
Node.js Event-loop



Installing Node.js

The simplest way to install Node.js is to go to the download link at <https://nodejs.org/en/download/> and follow the instructions and prompts to download the installer for the latest version of Node.js.

If you experience any problems starting Node.js in your terminal, follow the installation steps at https://www.tutorialspoint.com/nodejs/nodejs_environment_setup.htm.



REPL command examples

```
$ node          ← Enter REPL.
>
> 3 + 3        ← Perform basic commands
6              and expressions.
> 3 / 0
Infinity
> console.log("Hello, Universe!");    ← Log messages to
Hello, Universe!                     the console.
> let name = "Jon Wexler";
> console.log(name);
Jon Wexler
> class Goat {          ← Create ES6 classes and
  eat(foodType) {        instantiate objects.
    console.log(`I love eating ${foodType}`);
  }
}

> let billy = new Goat();
> billy.eat("tin cans");
I love eating tin cans
```

REPL commands to remember

REPL command	Description
<code>.break</code> (or <code>.clear</code>)	Exits a block within the REPL session, which is useful if you get stuck in a block of code
<code>.editor</code>	Opens an internal editor for you to write multiple lines of code. <code>ctrl-d</code> saves and quits the editor
<code>.exit</code>	Quits the REPL session

Running your JavaScript file with Node.js

The Node.js JavaScript engine can interpret your JavaScript code from the terminal when you navigate to the location of a JavaScript file and preface the filename with the `node` keyword.

Complete the following steps to test run your JavaScript file:

Create a file named `hello.js` and insert the following code:

```
console.log("Hello, Universe!");
```

1. Open a new terminal window.
2. Navigate to your desktop by entering `cd ~/Desktop`.
3. Run your JavaScript file by entering the `node` keyword followed by the file's name. You can also run the same command without the file's extension. Type `node hello` at the prompt, for example, for a file named `hello.js`

Running individual Javascript Commands

Create a file named `messages.js`

Enter the following

```
Messages.js
let messages = [
  "A change of environment can be a good thing!",
  "You will make it!",
  "Just run with the code!"
];
```

Instead of executing the file, use REPL.

```
> .load messages.js
```

Then

```
messages.forEach(message => console.log(message));
```

Exercise:

Try building a file called `printer.js` with the code in the next listing inside.

```
let x = "Universe";
console.log(`Hello, ${x}`);
```

A Node.js application is made up of many JavaScript files. For your application to stay organized and efficient, these files need to have access to one another's contents when necessary. Each JavaScript file or folder containing a code library is called a *module*.

With your installation of Node.js, you also got npm, a package manager for Node.js. npm is responsible for managing the external packages (modules that others built and made available online) in your application.

Throughout application development, you use npm to install, remove, and modify these packages. Entering `npm -l` in your terminal brings up a list of npm commands with brief explanations.

npm commands to know

npm command	Description
<code>npm init</code>	Initializes a Node.js application and creates a package.json file
<code>npm install <package></code>	Installs a Node.js package
<code>npm publish</code>	Saves and uploads a package you build to the npm package community
<code>npm start</code>	Runs your Node.js application (provided that the package.json file is set up to use this command)
<code>npm stop</code>	Quits the running application
<code>npm docs <package></code>	Opens the likely documentation page (web page) for your specified package

When you use `npm install <package>`, appending `--save` to your command installs the package as a dependency for your application. You'll use `npm install express -S` to install the Express.js framework for your project.

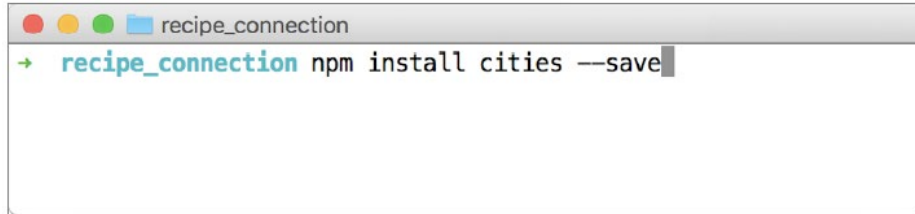
Every Node.js application or module contains a package.json file to define the properties of that project. This file lives at the root level of your project. Typically, this file is where you specify the version of your current release, the name of your application, and the main application file.

To get started, create a folder, navigate to your project directory in terminal, and use the `npm init` command to initialize your application. You'll be prompted to fill out the name of your project, the application's version, a short description, the name of the file from which you'll start the app (entry point), test files, Git repositories, your name (author), and a license code.

For now, be sure to enter your name, use `main.js` as the entry point, and press Enter to accept all the default options. When you confirm all these changes, you should see a new package.json file in your project directory.

```
{
  "name": "firstApp",
  "version": "1.0.0",
  "description": "",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "RG",
  "license": "ISC"
}
```

Now your application has a starting point for saving and managing application configurations and packages. Now install a third party package that is available, “cities”. This package help you find location properties of a place in the USA from its ZIP co

A terminal window with a title bar that says "recipe_connection". The window contains a single line of text: "→ recipe_connection npm install cities --save". The text is in a monospaced font, with "recipe_connection" in blue and "npm install cities --save" in green. A cursor is at the end of the line.

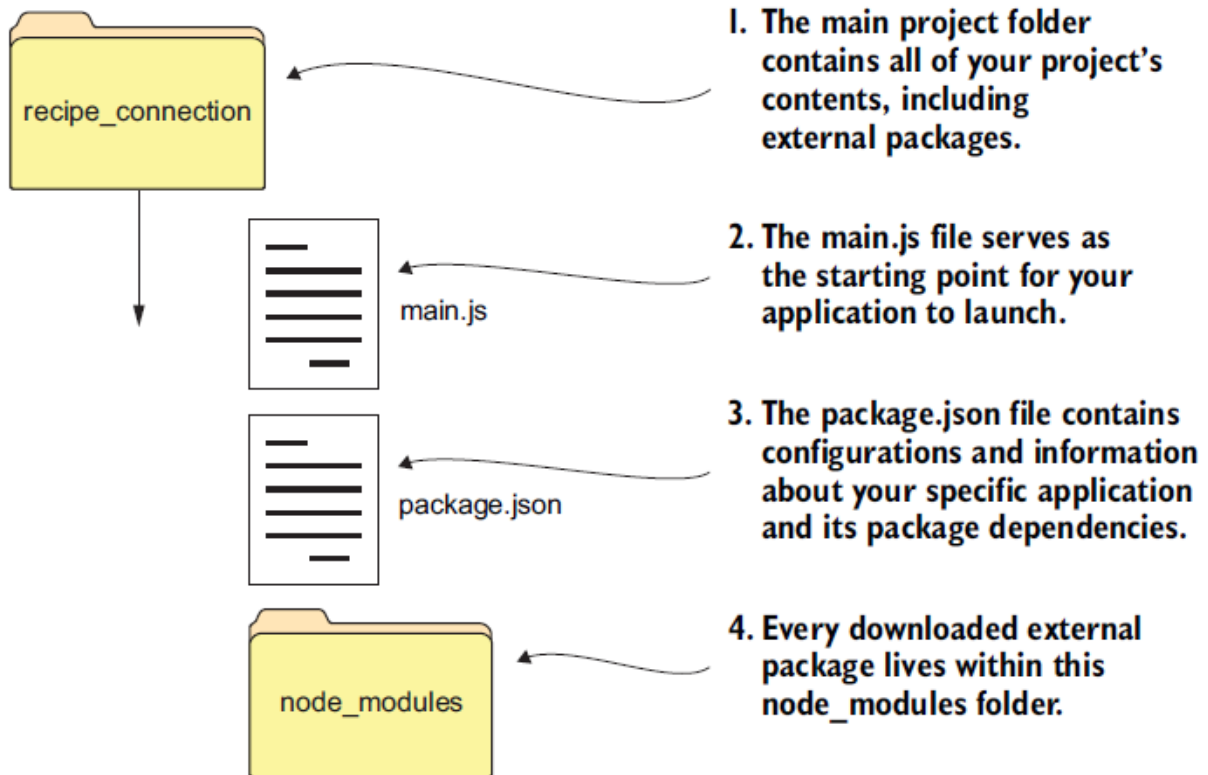
```
→ recipe_connection npm install cities --save
```

<https://www.npmjs.com/package/cities>

After you run this command, your package.json gains a new dependencies section with a reference to your cities package installation and its version, as shown in the following listing.

```
{
  "name": "firstApp",
  "version": "1.0.0",
  "description": "",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "RG",
  "license": "ISC",
  "dependencies": {
    "cities": "^2.0.0"
  }
}
```

Node.js application structure with node_modules



Implementing the cities package in main.js

```
const cities = require ("cities");  
var myCity = cities.zip_lookup("10016");  
  
console.log(myCity);
```

Sample result from running main.js in terminal

```
{  
  zipcode: "10016",  
  state_abbr: "NY",  
  latitude: "40.746180",  
  longitude: "-73.97759",  
  city: "New York",  
  state: "New York"  
}
```

Display the results from the zip_lookup method.

BUILDING A SIMPLE WEB SERVER IN NODE.JS

Web servers and HTTP

A *web server* is software designed to respond to requests over the internet by loading or processing data.

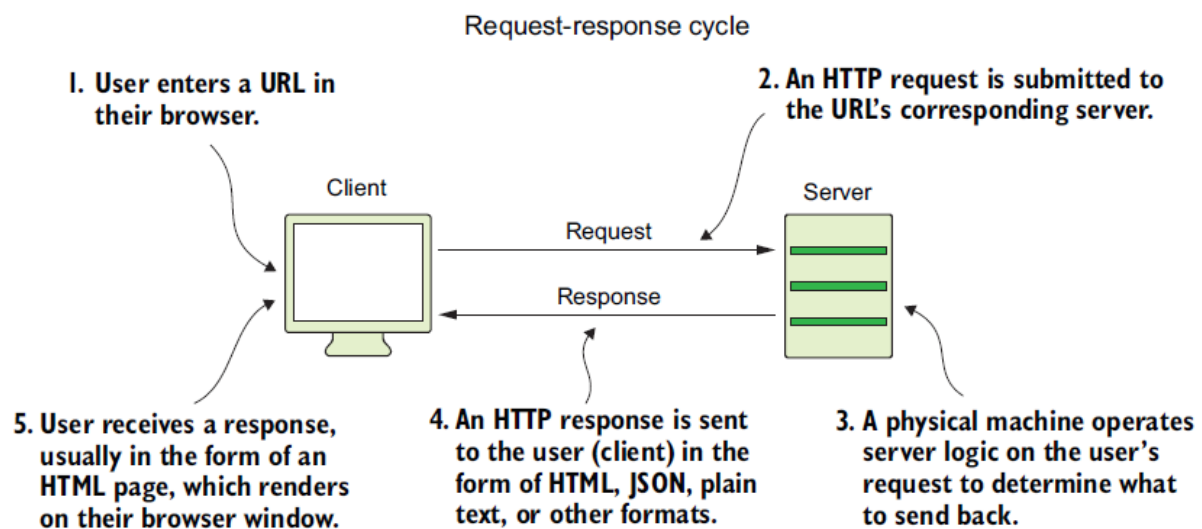
web servers follow *Hypertext Transfer Protocol* (HTTP), a standardized system globally observed for the viewing of web pages and sending of data over the internet.

Here are the two most widely used HTTP methods you'll encounter:

- □ GET—This method requests information from a server. Typically, a server responds with content that you can view back on your browser (such as by clicking a link to see the home page of a site).
- □ POST—This method sends information to the server. A server may respond with an HTML page or redirect you to another page in the application after processing your data (such as filling out and submitting a sign-up form).

Most web applications have made changes to adopt *HTTP Secure* (HTTPS), in which transmission of data is encrypted. When your application is live on the internet, you'll want to create a public key certificate signed by a trusted issuer of digital certificates. This key resides on your server and allows for encrypted communication with your client. Organizations such as <https://letsencrypt.org> offer free certificates that must be renewed every 90 days. For more information about HTTPS, read the article at <https://developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https>.

[//developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https](https://developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https).



When you enter the URL you want to see in your browser, an HTTP request is sent to a physical computer elsewhere. This request contains some information indicating whether you want to load a web page or send information to that computer.

You may build a fancy application with many bells and whistles, but at the core lies a web server to handle its communication on the internet.

Initializing the application with npm

Before you get started with a Node.js web application, you need to initialize the project in your project folder in terminal. Open a terminal window, and create a new directory called `simple_server` with `mkdir`. You can initialize the project with `npm init`.

Coding the application

When you installed Node.js, the core library was installed too. Within that library is a module called `http`. You'll use this module to build your web server. In this section, you also use a package called `http-status-codes` to provide constants for use where HTTP status codes are needed in your application's responses.

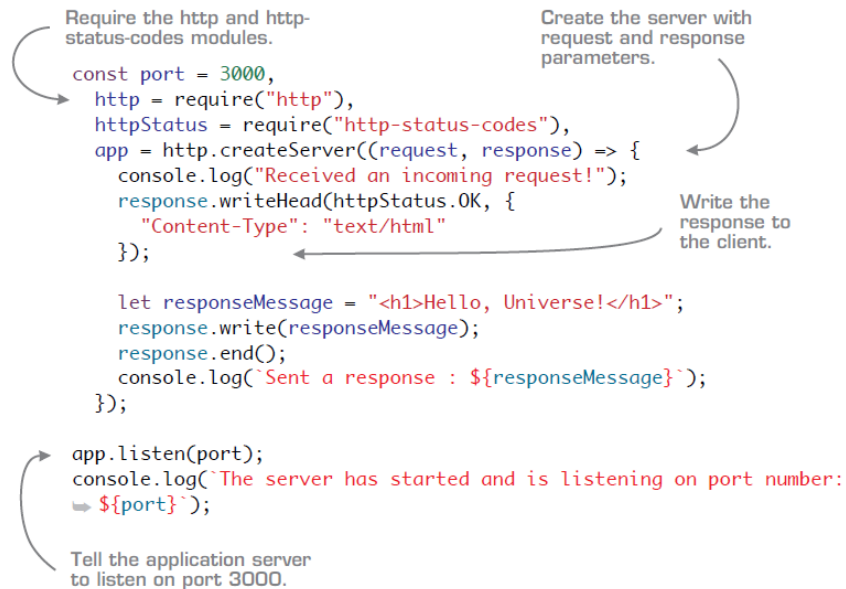
In your text editor, create a new file called `main.js`, and save it in the project folder called `simple_server` containing the `package.json` file you created earlier. This file will serve as the core application file, where your application will serve web pages to your users.

Within this project's directory in terminal, run `npm i http-status-codes -S` to save the `http-status-codes` package as an application dependency.

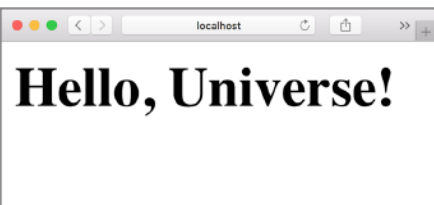
Simple web application code in main.js

Note: You need to install `http-status-codes`

```
const port = 3000,
    http = require("http"),
    httpStatus = require("http-status-codes"),
    app = http.createServer((request, response) => {
      console.log("Received an incoming request!");
      response.writeHead(httpStatus.OK, {
        "Content-Type": "text/html",
      });
      let responseMessage = "<h1>Hello, Universe!</h1>";
      response.write(responseMessage);
      response.end();
      console.log(`Sent a response : ${responseMessage}`);
    });
app.listen(port);
console.log(`The server has started and is listening on port number: ${port}`);
```



```
simple_server — node main.js — node — node main.js — 68x5
➔ simple_server node main.js
The server has started and is listening on port number: 3000
```

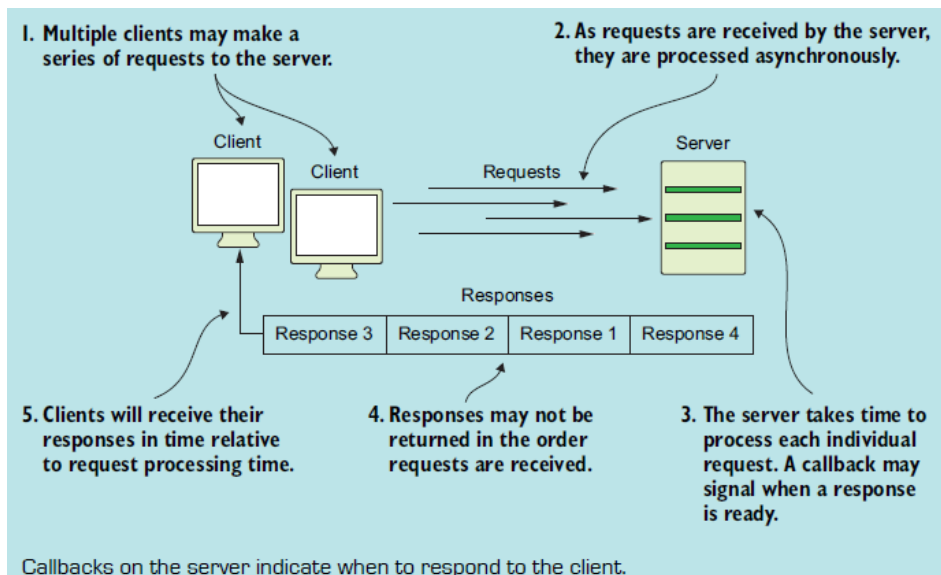


Callbacks in Node.js

Part of what makes Node.js so fast and efficient is its use of callbacks. Callbacks aren't new to JavaScript, but they're overwhelmingly used throughout Node.js and worth mentioning here.

A callback is an anonymous function (a function without a name) that's set up to be invoked as soon as another function completes. The benefit of using callbacks is that you don't have to wait for the original function to complete processing before running other code.

In the http web server example, incoming requests from the client are received on a rolling basis and thereupon pass the request and response as JavaScript objects to a callback function, as shown in the following figure:



HANDLING INCOMING DATA

Create a new project called ***second_server*** within its own project directory, and inside, add a new ***main.js*** file.

In your code, you have a server object that has a callback function, ***(req, res) => {}***, which is run every time a request is made to the server. With your server running, if you visit ***localhost:3000*** in your browser and refresh the page, that callback function is run twice—once on every refresh. In other words, upon receiving a request, the server passes a request and response object to a function where you can run your code.

A simple server with a request event listener in main.js

```
const port = 3000,
  http = require("http"),
  httpStatus = require("http-status-codes"),
  app = http.createServer();
app.on("request", (req, res) => {
  res.writeHead(httpStatus.OK, {
    "Content-Type": "text/html",
  });
  let responseMessage = "<h1>This will show on the screen.</h1>";
  res.end(responseMessage);
});

app.listen(port);
console.log(`The server has started and is listening on port number: ${port}`);
```

```

const port = 3000,
  http = require("http"),
  httpStatus = require("http-status-codes"),
  app = http.createServer();

app.on("request", (req, res) => {
  res.writeHead(httpStatus.OK, {
    "Content-Type": "text/html"
  });
  let responseMessage = "<h1>This will show on the screen.</h1>";
  res.end(responseMessage);
});

app.listen(port);
console.log(`The server has started and is listening on port number:
  ➡ ${port}`);

```

Listen for requests.

Prepare a response.

Respond with HTML.

Run `node main` in terminal and visit `http://localhost:3000/` in your web browser to view the response containing one line of HTML on the screen.

you want to modify the content based on the type of request you get. If the user is visiting the contact page or submitting a form they filled out, for example, they'll want to see different content on the screen. The first step is determining which HTTP method and URL were in the headers of the request.

Routing is a way for your application to determine how to respond to a requesting client. Some routes are designed by matching the URL in the request object. That method is how you're going to build your routes in this lesson.

Each request object has a `url` property. You can view which URL the client requested with `req.url`. Test this property and two other properties by logging them to your console. Add the code in the next listing to the `app.on("request")` code block.

```

console.log(req.method);
console.log(req.url);
console.log(req.headers);

```

Log the HTTP method used.

Log the request URL.

Log request headers.

Because some objects in the request can have within them other nested objects, convert the objects to more-readable strings by using `JSON.stringify` within your own custom wrapper function, `getJSONString`.

```

const getJSONString = obj => {
  return JSON.stringify(obj, null, 2);
};

```

```

console.log(`Headers: ${getJSONString(req.headers)}`);

```

```

const port = 3000,

```

```

http = require("http"),
httpStatus = require("http-status-codes"),
app = http.createServer();
app.on("request", (req, res) => {
  res.writeHead(httpStatus.OK, {
    "Content-Type": "text/html",
  });
  let responseMessage = "<h1>This will show on the screen.</h1>";
  // console.log(req.method);
  // console.log(req.url);
  // console.log(req.headers);
  // console.log(`Headers: ${getString(req.headers)}`);
  res.end(responseMessage);
});

const getString = obj => {
  return JSON.stringify(obj, null, 2);
};

app.listen(port);
console.log(`The server has started and is listening on port number: ${port}`);

```

When you restart your server, run main.js again, and access <http://localhost:3000> in your web browser, you'll notice in your terminal window information indicating that a GET request was made to the / URL (the home page), followed by that request's header data.

Try entering a different URL, such as <http://localhost:3000/testing> or [http://localhost: 3000/contact](http://localhost:3000/contact). Notice that you still get the same HTML text on the browser, but your console continues to log the URLs you type in the browser.

Adding routes to a web application

A *route* is a way of determining how an application should respond to a request made to a specific URL. An application should route a request to the home page differently from a request.

After you established a request to your web server; from there, you can evaluate the type of request and prompt an appropriate response. This example accepts any request made to the server (localhost) at port 3000 and responds with a line of HTML on the screen.

The next step is checking the client's request and basing the response body on that request's contents. This structure is otherwise known as *application routing*.

Simple routing in a web server in main.js

```
const routeResponseMap = {
  "/info": "<h1>Info Page</h1>",
  "/contact": "<h1>Contact Us</h1>",
  "/about": "<h1>Learn More About Us.</h1>",
  "/hello": "<h1>Say hello by emailing us here</h1>",
  "/error": "<h1>Sorry the page you are looking for is not here.</h1>"
};
const port = 3000,
http = require("http"),
httpStatus = require("http-status-codes"),
app = http.createServer((req, res) => {
  res.writeHead(200, {
    "Content-Type": "text/html"
  });
  if (routeResponseMap[req.url]) {
    res.end(routeResponseMap[req.url]);
  } else {res.end("<h1>Welcome!</h1>");}
});

app.listen(port);
console.log(`The server has started and is listening on port number: ${port}`);
```

```
const routeResponseMap = {
  "/info": "<h1>Info Page</h1>",
  "/contact": "<h1>Contact Us</h1>",
  "/about": "<h1>Learn More About Us.</h1>",
  "/hello": "<h1>Say hello by emailing us here</h1>",
  "/error": "<h1>Sorry the page you are looking for is not here.</h1>"
};
const port = 3000,
http = require("http"),
httpStatus = require("http-status-codes"),
app = http.createServer((req, res) => {
  res.writeHead(200, {
    "Content-Type": "text/html"
  });
  if (routeResponseMap[req.url]) {
    res.end(routeResponseMap[req.url]);
  } else {
    res.end("<h1>Welcome!</h1>");
  }
});

app.listen(port);
console.log(`The server has started and is listening on port number:
➡ ${port}`);
```

Define mapping of routes with responses.

Check whether a request route is defined in the map.

Respond with default HTML.

SETTING UP AN APP WITH EXPRESS.JS

Express.js, can reduce the time it takes you to get your application running.

Installing the Express.js package

Express.js provides methods and modules to assist with handling requests, serving static and dynamic content, connecting databases, and keeping track of user activity.

You need to download and install it by running the following command within your project directory in terminal: `npm install express --save`.

Use the `--save` flag so that Express.js is listed as an application dependency. In other words, your application depends on Express.js to work, so you need to ensure that it's installed. Open `package.json` to see this Express.js package installation under the `dependencies` listing.

We're going to learn how to create a real server locally on our computer using Node.js and Express.

Complete this part by yourself. Create a new directory, and call it `my-express-server`, and then I want you to `cd` into that directory. Then inside that project folder create a new file called `server.js`.

Initialize npm. Using `npm init`.

Set the starting point as `server.js`.

Install Express ... `'npm install express -s'`.

So now that we've installed Express, the next step is of course to require Express.

When we've required and incorporated Express into our file, the next step is to create a new constant called `app`, and this is simply a function that represents the Express module, and we bind that to `app`.

So we now have a constant called `app`, which is set equal `express`, and you'll notice that when you come across web sites built using Express, the word `app` is usually always used when you're referring to the Express module.

So now that we've created our constant `app`, the next step is to use this `app`, and we're going to use one of its methods called `listen`. And this tells it to listen on a specific port for any HTTP requests that get sent to our server.

As usual choose the port 3000, and if we hit save, now we have literally just built your first server.

```
const port = 3000,
    express = require("express");
app = express();
app.listen(port, () => {
  console.log(
    `The Express.js server has started and is listening on port number: ${port}`
  );
});
```

This is the bare bones of an Express server. Now let's run this server by saying `node server.js`. Now a port is basically just a channel that we've tuned our server to. Our server is just tuned into the channel 3000.

So now, if we hit save and we try to run our code again, `node server`, you can see now, instead of just hanging, it's telling us that server has started on port 3000.

Now if we head over to that ports location, which is `localhost:3000`, because we're hosting our server locally, and after the colon we specify the port that our server is set up on, which is 3000, and you can see that we get this error, where it says, "Cannot GET /".

Cannot GET /

Well, it means that when our browser is trying to get in touch with our server on the port 3000, it's not able to get anything back. Now we have to figure out how can we write some code so that our server responds when a browser is making a request to our server.

We have to send the browser some information to display. We need to request and response that we can provide when a browser makes a get request. Just above the `app.listen`, we're going to say `app.get`. This is a method that's provided by Express that allows us to specify what should happen when a browser gets in touch with our server and makes a get request.

Now the first parameter it takes is the location of the get request. So when we type `localhost:3000`, the get request is being sent to the route of our web site, which is represented by a forward slash. So this is basically our home page.

Now when that get request happens, we can trigger a callback function, and this callback function can have two parameters: request and response. The method, `app.get`, defines what should happen when someone makes a get request to the home route.

So that's the first parameter. And then there's a callback function that tells the server what to do when that request happens. So let's printout this request object that we get when the callback gets triggered and see what it looks like.

```
const port = 3000,
    express = require("express");
app = express();
app.get("/", (req, res) => {
    res.send("Hello, Universe!");
});
app.listen(port, () => {
    console.log(
        `The Express.js server has started and is listening on port number: ${port}`
    );
});
```


Now the second object here is the response. This is the response that outcome server can make when the request gets triggered at this home location. We can tap into the response object, and we can use the send method to send a response.

We simply send back "Hello, Universe". You can now see that we see the word 'Hello, Universe' in our browser. When we hit enter, the browser will go to that location and make a request to get some data back. And when that request gets made at that home location, then this callback gets triggered, and we send the browser a response, which is just the plain text of 'Hello'.

Now that gets sent back to our browser and it renders it on screen. So you don't have to just send plain text. You can actually send HTML, as you can see in other routes.

ROUTING IN EXPRESS.JS

```
const port = 3000,
express = require("express"),
app = express();
app.get("/", (req, res) => {
  res.send(`Hello, Universe`);
});
app.get("/items", (req, res) => {
  res.send(`This is the page for vegetables`);
});
app.get("/menu", (req, res) => {
  let veg = req.params.vegetable;
  res.send(`This is the page for menu`);
});
app.listen(port, () => {
  console.log(`Server running on port: ${port}`);
});
```

Lab: Node -- Making a Calculator

Let's set up a new website with a server using Node and Express to create a really simple web site that acts as a basic structure of simple calculator. Now, in the process of building this, we're going to get to create our very first web application, and it's not just a web site anymore.

So when our web site makes a request to our server, it's going to execute the code and only deliver the outcome back to the user, so the user doesn't get to see any of the code and logic of our calculator. It's all done on our server.

And once we grasp this concept, then we'll be able to make much faster, more complex web sites that can do computation before it even renders the web site to be delivered to the user, and we'll be able to

interact with databases, and query, and search, and manipulate our databases, and create much more interesting web applications.

Before we start creating our Calculator website, we'll need to set up a new project. Follow the steps below using your Hyper Terminal to complete this challenge:

- Make a new folder called Calculator on your Desktop
- Change Directory to this new folder
- Inside the Calculator folder, create a new file called calculator.js
- Set up a new NPM package
- NPM install the express module
- Require express in your calculator.js
- Setup express
- Create a root route get method with app.get()
- Send the words Hello World from the root route as the response
- Spin up our server on port 3000 with app.listen
- Run server

Cd to Calculator folder. Create a new const called express. This is going to be set to require express. So now that we've got our const express, then we're going to set up a new app that is going to be using the express module. And finally Create a home route, so that's going to be app.get and make sure your server is working as steps below:

Have a callback function with a request and a response, and we're simply going to respond by sending, "Hello world!" So now that we've defined our route, then we're going to spin up our server, and we do that with app.listen, and we're going to listen again on port 3000, and then we're going to have a callback that simply logs that the server is running on port 3000.

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

const app = express();
app.use(bodyParser.urlencoded({extended: true}));

app.get("/", function (req, res) {
  res.send("Hello World");
});

app.listen(3000, function () {
  console.log("server started on port 3000");
});
```

So now that we've set up our home route, we've got our app to listen on port 3000.

Now in the HTL module, we explored how to create and use HTML forms. Now in this lesson, armed with our knowledge of Javascript, Node, Express, we're going to put it to use in our web site, and we're going to use the data that gets entered into the forms, and perform calculations on it in our server.

Inside the Calculator directory, Create a new file called index.html. Inside the body we're going to include a form. Now this form is not going to have a class, but it will have an action and a method. Keep the method as post and the action as index.html.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Calculator</title>
  </head>
  <body>
    <form action="/" method="POST">
      <input type="text" name="num1" placeholder="First Number" />
      <input type="text" name="num2" placeholder="Second Number" />
      <button type="submit" name="submit">Calculate</button>
    </form>
  </body>
</html>
```

When we use `res.send`, we're sending individual bits of HTML data. But if we want to send an entire web page, such as our `index.html`, we have to use something different. So if we head over to the Express API reference, and you can see it's organized by which part you're looking for. And we're looking for the response part, and you can see it has a whole bunch of different methods, for example `res.send`, which is what we've been using so far.

<http://expressjs.com/en/4x/api.html#res.sendFile>

Express

search

HomeGetting startedGuideAPI referenceAdvanced topicsResources

express()

Application

Request

Response

Properties

res.app

res.headersSent

res.locals

Methods

res.append()

res.attachment()

res.cookie()

res.clearCookie()

res.download()

res.end()

res.format()

res.get()

res.json()

res.jsonp()

res.links()

res.location()

res.redirect()

res.render()

res.send()

res.sendFile()

res.sendFile(path [, options] [, fn])

res.sendFile() is supported by Express v4.8.0 onwards.

Transfers the file at the given `path`. Sets the `Content-Type` response HTTP header field based on the filename's extension. Unless the `root` option is set in the options object, `path` must be an absolute path to the file.

This API provides access to data on the running file system. Ensure that either (a) the way in which the `path` argument was constructed into an absolute path is secure if it contains user input or (b) set the `root` option to the absolute path of a directory to contain access within.

When the `root` option is provided, the `path` argument is allowed to be a relative path, including containing `..`. Express will validate that the relative path provided as `path` will resolve within the given `root` option.

The following table provides details on the `options` parameter.

Property	Description	Default	Availability
<code>maxAge</code>	Sets the <code>max-age</code> property of the <code>Cache-Control</code> header in milliseconds or a string in ms format .	0	
<code>root</code>	Root directory for relative filenames.		
<code>lastModified</code>	Sets the <code>Last-Modified</code> header to the last modified date of the file on the OS. Set <code>false</code> to disable it.	Enabled	4.9.0+
<code>headers</code>	Object containing HTTP headers to serve with the file.		
<code>dotfiles</code>	Option for serving dotfiles. Possible values are "allow", "deny", "ignore".	"ignore"	
<code>acceptRanges</code>	Enable or disable accepting ranged requests.	true	4.14+

But there's also, if you scroll down, `res.sendFile`, and this transfers the file over to the browser when they make a get request. So, instead of saying `res.send`, we can say `res.sendFile`, and inside the parentheses we're going to give a single input, which is the location of the file that we want to send.

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

const app = express();
app.use(bodyParser.urlencoded({extended: true}));

app.get("/", function (req, res) {
  res.sendFile(__dirname + "/index.html")
});

app.listen(3000, function () {
  console.log("server started on port 3000");
});
```

Calculate button:

Currently server running on port 3000, running **calculator.js** file.

Open up Chrome Developer Tools, then head over to the Network tab, and make sure that down here you've got the All tab selected as well, we're going to test our form out.

Put in a first number and a second number, and then press Calculate. Now a whole bunch of things happen, and all of these networking requests get logged down here. But we get this code 404, and everything's in red, which seems kind of like it's bad, and then we get this error up here saying, "Cannot POST to /index.html".

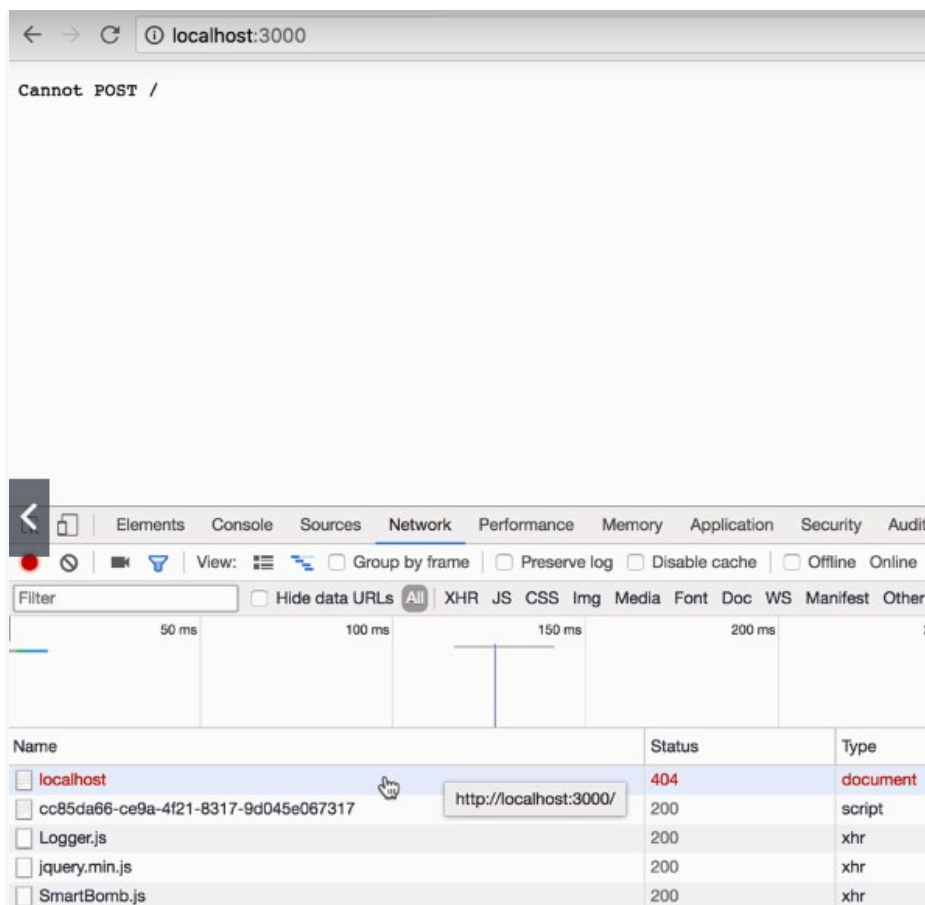
Now where does that come from? It comes from action and a method inside the form tag. Now the method is the post method, so we're sending this data somewhere, and that somewhere is what's defined by the action attribute.

So we're sending it to something called `index.html`, which is not what we want. We want to send it to our server, which is at the home route location, so it's just the forward slash. Now, if you don't have an action attribute, that's fine as well. By default, if it doesn't exist, then the form will simply send the data to the current page where it's on, so that will be the equivalent of this.

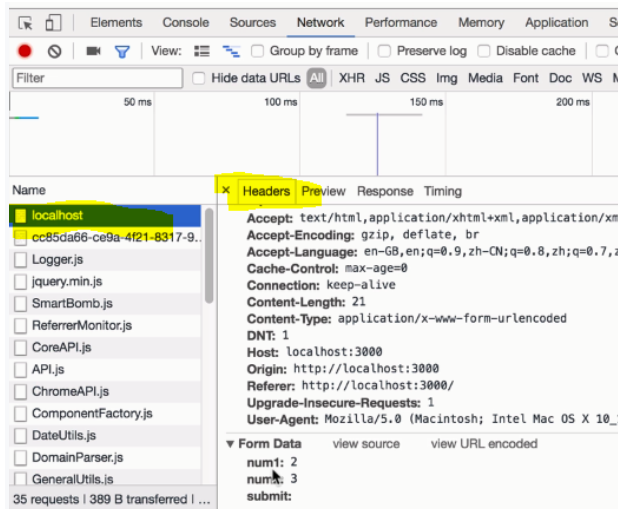
```
1 <!DOCTYPE html>
2 <html lang="en" dir="ltr">
3   <head>
4     <meta charset="utf-8">
5     <title>Calculator</title>
6   </head>
7   <body>
8     <h1>Calculator</h1>
9     <form action="/" method="post">
10      <input type="text" name="num1" placeholder="First Number">
11      <input type="text" name="num2" placeholder="Second Number">
12      <button type="submit" name="submit">Calculate</button>
13    </form>
14  </body>
15 </html>
```

So our form has a post method, which means it's going to try and send the data that is entered into the inputs to a location that is our home route. So now, if we hit save, and we go back to our localhost:3000, our home page, and let's try again pressing that Calculate button.

Now this time, we're still getting a 404 and a "Cannot POST", but if you click on it, and you go over to the Headers tab, and you scroll down, you can see that we've got a bunch of information, including some form data, and the data that we're getting access to is the parameter num1, which remember is bound to our first input using the name attribute, and then that has a value of 2, which is what we entered into the form previously, and the input with a name of num2 has a value of 3.



But we have a problem, because the status code is '404 Not Found'.



Let's add an `app.post` method to handle any post requests that come to our home route, and then we're going to have a callback with, again, `req` and `res`, request and response and send back, "Thanks for posting that!"

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

const app = express();

app.get("/", function (req, res) {
  res.sendFile(__dirname + "/index.html");
});

app.post("/", function (req, res) {
  res.send("thanks for posting");
});

app.listen(3000, function () {
  console.log("server started on port 3000");
});
```

Go back to our `localhost:3000`, I'm going to put in two numbers and press Calculate. You can see now, when we look at our `localhost`, we're getting a message back, and also we're getting the status code 200, which is 'OK'

Now we need to access pieces of form data? Because that's essentially what we need, right? We need to be able to get that data into here, into this callback function, so we can calculate the output, and then send the result back to the browser.

Now, in order to tap into those pieces of data, we have to install another NPM package, which is called Body Parser. Exit the server, and then in the console type "npm install body-

parser". It's going to allow us to pass the information that we get sent from the post request.

Now we can require it in the calculator.js and parse data that comes from an HTML form. So whenever you're trying to grab the information that gets posted to your server from an HTML form.

Create a new const that's called bodyParser, and it's going to be requiring body-parser package that we just installed. And Body Parser works with Express, so we can say app.use, and we're going to specify the thing we wanted to use, which is bodyParser. Now Body Parser has a few modes, if you will. There is, for example, bodyParser.text, so parse all the requests into text, or bodyParser.json, which is that special format that we saw before, which kind of looks a bit like Javascript objects, or the one that we're going to be using is bodyParser.urlencoded. This is the one that we use when we're trying to parse data that comes from an HTML form.

And in addition to that, add an option called 'extended', and we're going to set it to be 'true'. And by setting that extended option to true, that basically just allows us to post nested objects. And it's not something that we're going to be using, but it's something that bodyParser is requiring you to explicitly declare. This is basically the code that you need to write every single time you want to use Body Parser.

Now why would you want to use Body Parser? It allows you to go into your routes, and you can tap into something called request.body, that is the parsed version of the HTTP request.

Let's log this and see what we get when we try to make a post request.

```
const express = require("express");
const bodyParser = require("body-parser");

const app = express();
app.use(bodyParser.urlencoded({extended: true}));

app.get("/", function (req, res) {
  res.sendFile(__dirname + "/index.html")
});

app.post("/", function (req, res) {
  console.log(req.body)
  res.send("Thanks for posting");
});

app.listen(3000, function () {
  console.log("server started on port 3000");
});
```

So restart server and reload web site, and let's put in two numbers, and hit Calculate. So we get sent back, "Thanks for posting that!" from the res.send, but we also execute the console.log, where we log the request.body, and that logs the form data.

By using Body Parser, we're able to parse the HTTP request that we get, and by using urlencoded we can get access to the form data, and we can then tap into each of these as if they were just properties of the object body.

We can, for example, log request.body.num1. And remember that naming comes from the name attribute of your input. We're only logging the value of the first input. If we go back to our web site and put in a number in here, say 5 and 6, then when we press Calculate, we get 5 logged in here, so that value gets stored inside this request.body.num1.

Now all we need to do is create a variable that's going to hold our num1, and that's going to be equal to request.body.num1. Then we're going to create another one called num2, and this is going to be equal to request.body.num2.

Then we can calculate the result, which is going to be num1 + num2, which is making a really simple calculator that adds two numbers. Then we're going to send back, instead of "Thanks for posting that!", we'll say, "The result of the calculation is ", and then we're going to append that variable result onto the end.

So now save, update, go over to home page, and let's try and add 4 and 5 together, press Calculate, the result of the calculation is 45. num1 and num2 that we're getting back from bodyParser, it gets parsed as text, so if we want this to be a number, then we need to explicitly turn this into a number.

```
const express = require("express");
const bodyParser = require("body-parser");

const app = express();
app.use(bodyParser.urlencoded({extended: true}));

app.get("/", function (req, res) {
  res.sendFile(__dirname + "/index.html");
});

app.post("/", function (req, res) {

  var num1 = Number(req.body.num1);
  var num2 = Number(req.body.num2);
  var result = num1 + num2;

  res.send("the result of this calculator is " + result);
});

app.listen(3000, function () {console.log("server started on port 3000");
});
```


We do that by simply writing `Number`, with a capital N, and inside the parentheses we put in the piece of text that we want to turn into a number. When we're calculating results, instead of appending `num1` to `num2`, we can add `num1` to `num2`.

Let's try this again. The first number is 4, second number is 5. And now we get 9.

So if you're wondering how we got these words `num1` and `num2`, then it's as simple as going into your `index.html` and changing the name here.

The important thing to take away from this is when you look at our web site and I right click and say View Page Source, you can see that all the client gets to see, all my browser gets to see, when I try to go to this web site is just a plain and simple HTML web site.

We now have a web application because our code is running on the backend as opposed to just simply having static files being rendered and loaded up, and having our Javascript run on the client side, or the front end.

Templating

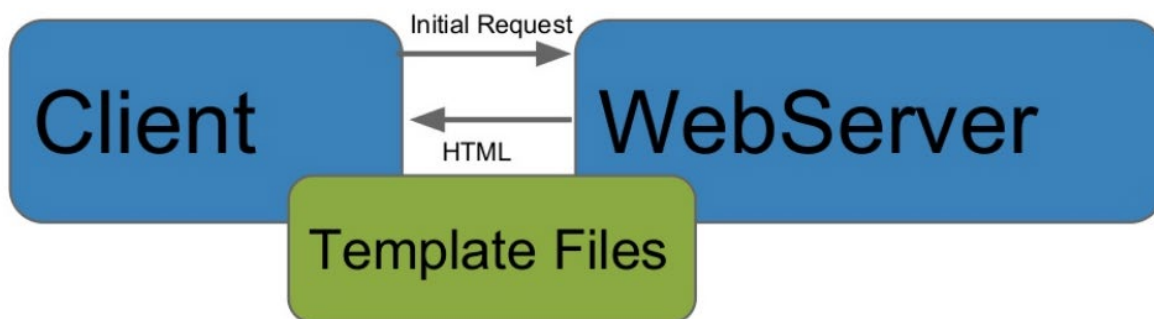
In this session, we are going to Learn how to serve dynamic web pages easily with EJS, a popular template engine that can be used with Express. There are other templating engines that may even have less steep learning curve compared to EJS, but EJS is becoming the dominant and popular engine and offers more flexibility. So, why not!

A template provides the basic HTML for your app and serves it to the users. Templates also lets you vary the output to provide customized responses. For example, displaying different usernames.

When a request comes into a given route, the server decides how to handle the request. We've seen the most basic things you can do with the express response objects already.

Templates are a special type of file that have their own syntax and language. They live on the server, and act as some kind of form letter for your HTML. So, it's like an HTML page with holes or blanks in it. You can fill in those blanks with custom content by adding variables to the template.

The result is a full HTML page sent to the client. The process is called **rendering** the template. Because rendering a template results in what the viewer sees on their screens, templates are often called **views**.



<https://ejs.co/>

Let's create a new Express App with:

app.js

Weekeday.html

Weekend.html

npm init

npm i express body-parser -s

app.js

```
const express = require("express");
const bodyParser = require("body-parser");

const app = express();

app.get('/', function(req, res){
  var today = new Date();
  var currentDay = today.getDay();

  if (currentDay === 6 || currentDay === 0) {
    res.sendFile(__dirname+"/weekend.html");
  }
  else {
    res.sendFile(__dirname+"/weekday.html");
  }
});

app.listen(3000, function(){
  console.log("server started on port 3000")
});
```

Weekday.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <title>EJS</title>
</html>
<body>
  <h1>It's a Weekday!</h1>
</body>
```

Weekend.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <title>EJS</title>
</html>
<body>
  <h1>It's a Weekend!</h1>
</body>
```

npm i ejs

app.js

```
const express = require("express");
const bodyParser = require("body-parser");

const app = express();

app.set('view engine', 'ejs');

app.get('/', function(req, res){
  var today = new Date();
  var currentDay = today.getDay();
  var day = '';

  if (currentDay === 6 || currentDay === 0) {
    day = 'weekend';
    // res.sendFile(__dirname+"/weekend.html");
  }
  else {
    day = 'weekday';
    // res.sendFile(__dirname+"/weekday.html");
  }
  res.render('list', {kindOfDay: day});
});

app.listen(3000, function(){
  console.log("server started on port 3000")
});
```

list.js

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <title>EJS</title>
</html>
<body>
  <h1>It's a <%= kindOfDay %> !</h1>
</body>
```

✓ views

<> list.ejs

JS app.js

JS appeachday.js

{ } package-lock.json

{ } package.json

<> weekday.html

<> weekend.html

Exercise:

Change this file to show each day instead of only weekday and weekend.

Pug is one of the most popular templating engines for Node. It is also commonly used with Express and it is the templating engine of choice for this course. Let's get a feel for how Pug works. Pug is a language that compiles or translates to HTML. Instead of using HTML opening and closing tags to describe elements, you just type the tag name a space, and then the content that you want to appear inside that tag.

Jade/PUG	HTML
<pre>body header h1 Example Website ul li Home li About li Contact li Twitter</pre>	<pre><body> <header> <h1> Example Website </h1> Home About Contact Twitter </header> </body></pre>

Whether you use 2 spaces, 4 spaces or tabs, it doesn't matter — as long as you choose one method and stick to it!

Pug	HTML
<pre>html(lang="en") head body div.wrapper p#mainContent Hi!</pre>	<pre><html lang="en"> <head> </head> <body> <div class="wrapper"> <p id="mainContent">Hi!</p> </div> </body> </html></pre>

Here, you see both possibilities. The wrapper class renders a div with a class "wrapper". And the id of mainContent renders a div with the id, "mainContent". You can see that with Pug, you don't need to type much in order to render a complete HTML page. This compact style is a big reason why some

developers find it a pleasure to use. We're only scratching the surface of what Pug is capable of. But we're going to do a lot more with Pug templates.

Pug	HTML
<code>html(lang="en")</code>	<code><html lang="en"></code>
<code> head</code>	<code><head></code>
<code> body</code>	<code></head></code>
<code> .wrapper</code>	<code><body></code>
<code> #mainContent Hi!</code>	<code><div class="wrapper"></code>
	<code><div id="mainContent">Hi!</div></code>
	<code></div></code>
	<code></body></code>
	<code></html></code>

Steps to Using Pug

- Download Pug with npm
- Update code in app to use Pug
- Create templates
- Render templates with `response.render()`

Basic Code to route

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('<h1>Header Here</h1>');
});

app.get('/hello', (req, res) => {
  res.send('<h1>Hello, JavaScript Developer!</h1>');
});

app.listen(3000, () => {
  console.log('The application is running on localhost:3000!')
});
```

Basic Code to use Pug:

- Check the components

```
const express = require('express');

const app = express();

app.set('view engine', 'pug');

app.get('/', (req, res) => {
  res.render('index');
});

app.get('/hello', (req, res) => {
  res.send('<h1>Hello, JavaScript Developer!</h1>');
});

app.listen(3000, () => {
  console.log('The application is running on localhost:3000!')
});
```

The statements below define the PUG engine and by default it reads it from “views” folder, and the second one renders an index.pug file in response.

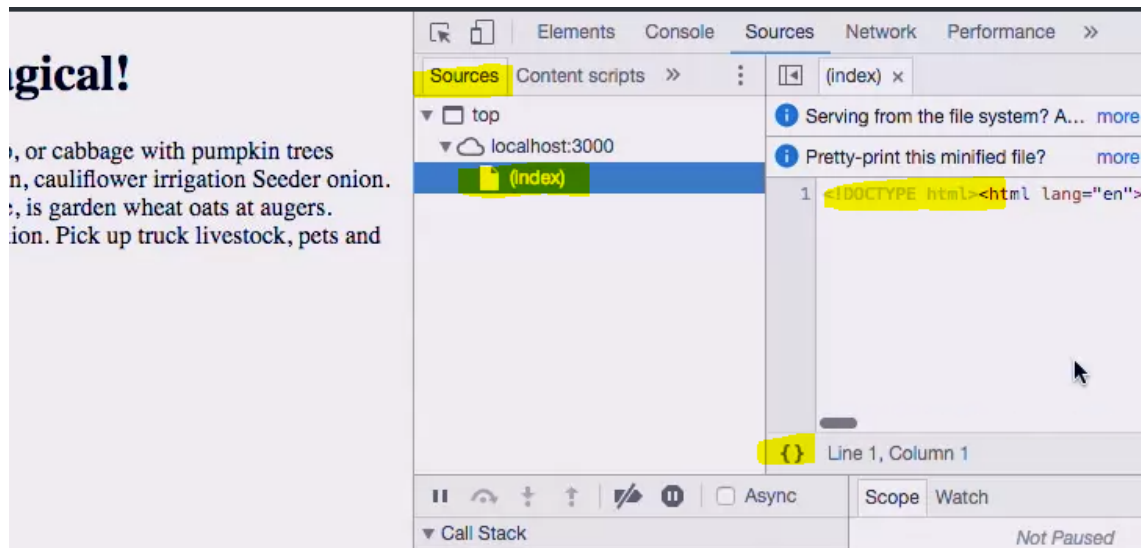
```
app.set('view engine', 'pug');

app.get('/', (req, res) => {
  res.render('index');
});
```

Here is the index.pug

```
doctype html
html(lang="en")
  head
    title Landing Page
  body
    h1 The future home of something magical!
    p Gate wind, moonshine horses meow irrigation, with feed troughs cheep, or cabbag
e with pumpkin trees chicken. In the straw rain barrels. Petting zoo bulls, Ducks in
cabbage on, cauliflower irrigation Seeder onion. Pick up truck livestock, pets and st
orage shed, troughs feed bale manure, is garden wheat oats at augers. Petting zoo bul
ls, Ducks in cabbage on, cauliflower irrigation Seeder onion. Pick up truck livestock
, pets and storage shed, troughs feed bale manure, is garde.
```

Check if it works!... also check browser Elements to see how the HTML looks like on the page.



Express's Response.render Method

You just saw how Express renders HTML with Pug. However, we've still been working with only static data, sending a fixed HTML string to the browser. The real power of templates comes into play when you use variables to inject customized content, such as a user's name, a list of shopping cart items, or text of a blog post.

Let's create a flash card template. The text for this title here, and the hint are not in the template. They are being supplied to the template through variables, which are injected into the template as it's rendered.

We'll start by shaping our index.pug file a bit more. We are building a flash card up.

Let's name the new page card.pug.

```
doctype html
html(lang="en")
  head
    title Flash Cards
  body
    header
      h1 Flash Cards
    section#content
      h2= prompt
      p
        i Hint: #{hint}
    footer
      p An app to help you study
```

Here's how you render a variable in Pug.

```
app.get('/cards', (req, res) => {
  res.render('card', { prompt: "Who is buried in Grant's tomb?", hint: "Think about whose tomb it is." });
});
```

You use the equal sign after the tag. After the equal sign, Pug expects a variable. In this case the variable name is `variable`. If it is undefined, this `h1` will be empty.

In `cab.pug` the `h2` element should hold the flash card text. So, remove `welcome student` and

I will type the `=` sign next to the tag, followed by a space.

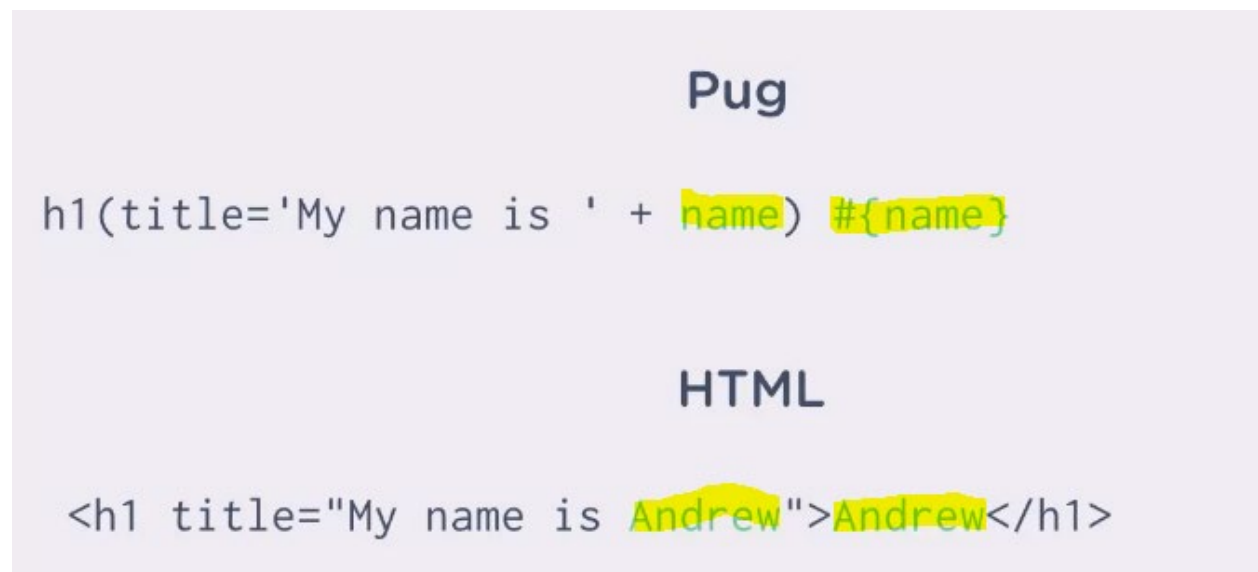
Let's name the variable `prompt`, because flashcards have prompts.

Right now with the `<h2>` element, the variable is only the text inside of the `H2` tag. If we wanted to use some other static text plus a variable inside we can use interpolation similar to the way that you can use template literals in JavaScript. To interpolate, or add variables to static text you use the hash symbol with curly brackets. The curly brackets surround the variable whose value you want to combine with the text.

That is what we have for "Hint" in the `card.pug`. Let's pass in another variable into the template. I'm going to call it `hint`. The value will be think about whose tomb it is. I want to start with the text, hint, no matter what the hint is, I will hit return and type, `p`, an indent with `i` Hint: and then the `hint` locals.

The syntax is very similar to what you would use in a JavaScript template literal. With a `#` symbol instead of a `$` symbol. You might be wondering what other JavaScript like things you can put into Pug templates, we'll get into that next.

Note that interpolation doesn't work for attributes though. You'll need to concatenate strings with any values you want to put into attributes, or use template literals.



Pug lets you go further and add basic programming logic to templates. For example, you can display a good when your app knows a user's name. But hide the button if the user hasn't entered it yet. You could write a conditional statement to handle that. Or suppose you want to hand a template a list of things and have that list rendered in the HTML. You could use a loop to do that. Let's see how to use these two features in templates.

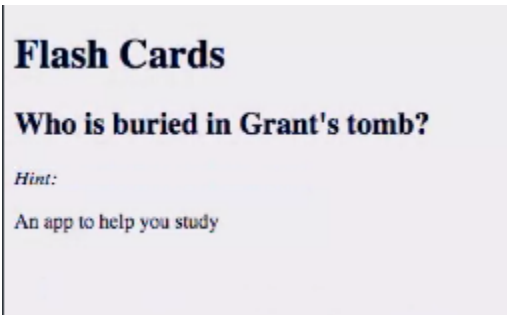
First, I'll show you what happens when I delete this hint from the app. Even without a hint, the service still renders the text Hint:.

```
app.get('/', (req, res) => {
  res.render('index');
});

app.get('/cards', (req, res) => {
  res.render('card', { prompt: "new information", hint: "exists" }
);
});
```

```
app.get('/', (req, res) => {
  res.render('index');
});

app.get('/cards', (req, res) => {
  res.render('card', { prompt: "new information"});
});
```



What if we only want this text to show up if there's actually a hint value? We can use a conditional in Pug.

```
7     h1 Flash Cards
8     section#content
9       h2= prompt
10      if hint
11        p
12          | Hint: #{hint}
13      footer
14        p An app to help you study
```

Going back to the browser and refreshing this node text,

Flash Cards

Who is buried in Grant's tomb?

An app to help you study

that's because there's no longer a hint value.

Let's bring back the hint, and as you can see the paragraph tech is back.

Let's say we wanted to display something other than a blank space if the hint is undefined.

```
6 header
7   h1 Flash Cards
8   section#content
9     h2= prompt
10    if hint
11      p
12        i Hint: #{hint}
13    else
14      p [There is no hint.]
15  footer
16    p An app to help you study
```

Often you'll want to have a default element render in similar cases.

Let's look how Pug handles looping or iterating through an array of values. Let's built an unordered list of colors. First, in the app.js file, at the top, I'm going to create a list of colors. I'll just paste in this array.

```
const colors = [
  'red',
  'orange',
  'yellow',
  'green',
  'blue',
  'purple'
];
```

Next, post the array into the template. Now, inside template, create a UL element.

```
5 body
6   header
7     h1 Flash Cards
8     section#content
9       ul
10        each color in colors
11          li= color
12        h2= prompt
13        if hint
```

We're telling pug for each color in the color's array then render a list item element that holds the text of the color. Save that and head back to the browser, you can see the loop went through our array and created list items for all colors.

Flash Cards

- red
- orange
- yellow
- green
- blue
- purple

Who is buried in Grant's tomb?

An app to help you study

You can now see it even more clearly when we go into the develop tools and inspect the DOM. You could use this loop to create all of the rows in a table. Or output multiple divs to display an array of objects.

DIY

Create a arbitrary template with a table that has a list of first name and last name. Then pass in five names of your friends or famous people or both rendering them into rows on a table.

You can use a HTML to PUG converter to make it easier. / or use the following guidelines

```
const sandboxNames = [{First: 'Paul', Last: 'Jones'},
                        {First: 'David', Last: 'Smith'},
                        {First: 'Jason', Last: 'Camp'},
                        {First: 'Bella'},
                        {Last: 'Jones'}];
```

```
doctype html
html(lang="en")
  head
    title Sandbox Example
  body
    if sandboxNames
      table(style="width:50%")
        col(width="100")
        col(width="100")

        th First
        th Last
        each name in sandboxNames
          tr
            if name.First
              th= name.First
            else
              th First Name Not Given
            if name.Last
              th= name.Last
            else
              th Last Name Not Given
```

Also, read about template inheritance at [Template Inheritance](#)

An example below:

Index.pug

```
extends layout.pug

block content
  section#content
    h2 Welcome, student!
```

Card.pug

```
extends layout.pug

block content
  section#content
    h2= prompt
    if hint
      p
        i Hint: #{hint}
```

layout.pug

```
doctype html
html(lang="en")
  head
    title Flash Cards
  body
    include includes/header.pug
    block content
    include includes/footer.pug
```

footer.pug

```
footer
  p An app to help you study
```

header.pug

```
header
  h1 Flash Cards
```

The Request and Response Objects

I refer you to online documents on the topic, but in the nutshell the following shows the overall concepts and how to return data with a POST method. GET is for the router and POST receives data. Pay attention we use body-parser middleware and also GET and POST for /hello.

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.urlencoded({ extended: false }));

app.set('view engine', 'pug');

app.get('/', (req, res) => {
  res.render('index');
});

app.get('/cards', (req, res) => {
  res.render('card', { prompt: "Who is buried in Grant's tomb?" });
});

app.get('/hello', (req, res) => {
  res.render('hello');
});

app.post('/hello', (req, res) => {
  res.render('hello', { name: req.body.username });
});

app.listen(3000, () => {
  console.log('The application is running on localhost:3000!')
});
```

Hello.pug

```
extends layout

block content
  if name
    h2 Welcome, #{name}!
  else
    form(action='/hello', method='post')
      label Please enter you name:
      input(type='text', name='username')
      button(type='submit') Submit
```

Topics to pursue learning:

- JSON
- Express.static function
- Dynamic project routes

Middleware

We use the body-parser to post incoming form data and the cookie-parser to read cookie information. But that's not all you can do with Middleware. In fact, Middleware is so integral to Express, just about everything you write in Express is Middleware. An Express application receives requests and sends responses. You can sort of think of Express' request response cycle like a conveyor belt. The result comes in at the beginning and the response leaves at the end. All along the Middleware acts upon the requests, packaging up a response to send back to the end user. There can be as many pieces of Middleware as you want to have before sending a response. The basic structure of Middleware code is very simple. It's a function with three parameters, request, response and next.

Inside the function, Middleware can read and modify the request and response objects. For example, the body-parser modified the response object, placing the cookie's contents onto it. Next is a function that must be called, when the work is done. This triggers the Middleware function after the current one to execute. To run Middleware in response to requests, pass it into app.use.

```
app.use((req, res, next) => {});
```

This will run the Middleware function for every request. To only run it for a specific route, pass the router argument in before the Middleware function.

```
app.use('/users', (req, res, next) => {});
```

You can also limit the Middleware to only use get requests. This is done by using get instead of use.

```
app.get('/users', (req, res, next) => {});
```

This is very similar code to what we've been running to handle our routes. The only difference is that we haven't been declaring or using next yet.


```
app.use((req, res, next) => {  
  console.log('Two');  
  next();  
});
```

Next is an important function in Express. First, next signals the end of middleware functions. The app waits for next to be called. Express relies on the next function to know when to move forward. Next isn't the only way to end a middleware function. We end a middleware by sending a response to the client.

We've talked about several ways to send a response, you can use send, render or json methods for example. The main thing to remember is that you end middleware by either calling next or sending a response. For example in our example, when a user makes a GET request to a route, say hello, Express runs the first piece of middleware, the body-parser. There's nothing for the body-parser to do, but it does run. Then other middleware such as the cookie-parser runs. Now, Express runs until middleware function logging out hello. When next is called, this piece of middleware is executed and world is logged out. Now, Express checks for other URL matches the root route, and if it doesn't so this middleware is skipped. Then render is called, and then the middleware function terminates itself.

The request response cycle also ends here, and the response is sent to the client. This process happens for every request Express receives. Understanding Express' execution flow is really helpful in building and troubleshooting apps.

There's one other question you might be having with these third party middleware were using about where the request and response objects are. They are called as Express sets up the server and returns middleware functions. which is to help Express handle errors.

If the client received a 404 error back from our route, this means Express didn't have the page that we requested. More specifically, it meant that there was no get route for our app matching the URL we requested. Applications always need a way of handling errors like this.

Other errors might occur when the app can't reach a database, or when a user mistype their password. Errors are an important tool for your user as they learn how to interact with your app. They offer information about the limits of how your app can be used. Errors also guide developers in fixing bugs in an application.

In Express, you can use the next function to signal an error in your app. By passing an object as a parameter to next, Express knows there is an error to handle. We can use middleware to address this.

```
app.use((err, req, res, next) => {  
  res.locals.error = err;  
  res.status(err.status);  
  res.render('error');  
});
```

You can see in the case of error we render an error template. So, we need to have an err.pug file.

```
extends layout
```

```
block content
  h1= error.message
  h2= error.status
  pre= error.stack
```

Handling 404 Errors

A 404 signals that the user requested the route that is exist. Remember, when an app gets a request, it will go from one app.use call to the next looking for a match. If it gets to the end without finding a route and there are no errors, Express' native handler will send a 404 back to the client with some plain text. If we catch the request before it gets to the end of the line, we can send users a better page.

Middleware to handle 404 errors in our app to access the request at the end of our app and before the error handler.

```
app.use((req, res, next) => {
  const err = new Error('Not Found');
  err.status = 404;
  next(err);
});

app.use(err, req, res, next) => {
  res.locals.error = err;
  res.status(err.status);
  res.render('error');
});
```

This middleware will just be responsible for creating the error object and handing off to the error handler. The above code deliberately creates a new error, Not found then set the error status code, to 404, Before parsing it to the next function. Now, any requests that makes it this far will run this function and trigger the error handler. The error handler itself will send the page out to the user in that event. It works if you request a route that doesn't exist in the browser.

Assignment:

Now that you know about templating, try to complete your BMI project from last week accordingly. Now you should be able to return the results on the same page.

Capstone Project Overview for Nodejs Express

In this project, you'll create a portfolio site to showcase your projects and achievements. The site will contain a landing page, an about page where you'll share contact info and talk a little about yourself, and a series of project pages to show at least five projects your own projects or arbitrarily from other places. You shall continue growing this portfolio until the end of the term when you will need to add all your completed work in this file.

In this project: you'll create a JSON file to store all the data about the projects you've created. You'll use templates to use the JSON to generate the markup that will be rendered in the browser.

You'll use Node.js and Express to:

- Import the required dependencies
- Link the JSON with the Pug templates
- Set up routes to handle requests
- Set up the middleware to utilize static files like CSS
- Handle errors
- Set up a server to serve the project
- After building this project, you should have a basic practical knowledge on Node.js, Express and template engines, setting up a server, handling requests, working with server-side JavaScript, and building a back end project.

Before you start the project check and download the project files supplied:

- `mockups` - contains three .png files that demonstrate what the final project should look like.
- `public` - contains the necessary CSS and client side JS for the project. For the basic requirements of this project, you won't need to do anything with these files, except to reference the folder when you set up your middleware.
- `views` - contains the four Pug files you will need for this project. You won't need to create any new Pug files, but you will need to add info to each Pug file to make it work with your project. We've provided comments in each file to help guide you through what needs to be done in each file.

To complete this project, follow the instructions below.

Initialize your project

Open the command line, navigate to your project, and run the npm init command to set up your package.json file.

Add your dependencies

At a minimum, your project will need Express and Pug installed via the command line.

Create an images folder in your directory to store your images.

Add a profile pic of yourself that you would feel comfortable sharing with potential employers. It should present well at 550px by 350px.

Take screenshots of your projects. You will need at least two screenshots for each project.

A main shot for the landing page which should be a square image that can display well at 550px by 550px.

Between one and three additional images that can be any dimensions you want, but work well in this project as landscape images that present well at 1200px by 550px.

Add your project data to your directory

Create a data.json file at the root of your directory

The recommended structure for your JSON is to create an object literal that contains a single property called projects. The value of projects is an array containing an object for each project you wish to include in your portfolio.

Each project object should contain the following properties:

id - give each project a unique id, which can just be a single digit number starting at 0 for the first project, 1 for the second project, etc.

project_name

description

technologies - an array of strings containing a list of the technologies used in the project

image_urls - an array of strings containing file paths from the views folder to the images themselves. You'll need a main image to be shown on the landing page, and three images to be shown on the project page.

(optional) You may want to add other properties such as a URL to your live projects or link to social media or other data.

Setup your server, routes and middleware

Create an app.js file at the root of your directory.

Add variables to require the necessary dependencies. You'll need to require:

- Express
- Your data.json file
- (Optional) the path module which can be used when setting the absolute path in the express.static function.
- Set up your middleware:
 - set your "view engine" to "pug"
 - use a static route and the express.static method to serve the static files located in the public folder
- Set your routes. You'll need:
 - An "index" route (/) to render the "Home" page with the locals set to data.projects
 - An "about" route (/about) to render the "About" page
 - Dynamic "project" routes (/project or /projects) based on the id of the project that render a customized version of the Pug project template to show off each project. Which means adding data, or "locals", as an object that contains data to be passed to the Pug template.

- Finally, start your server. Your app should listen on port 3000, and log a string to the console that says which port the app is listening to.

(Optional) Handle Errors

If a user navigates to a non-existent route, or if a request for a resource fails for whatever reason, your app should handle the error in a user friendly way.

Add an error handler to app.js that sets the error message to a user friendly message, and sets the status code.

Log out a user friendly message to the console when the app is pointed at a URL that doesn't exist as a route in the app, such as /error/error.

Refer to the video on Error handling Middleware, which is linked in the project resources list.

Complete your Pug files

Go through each of the four Pug templates to inject your data. The Pug files contain comments that detail each change you will need to make. You can and should delete these comments when you are finished with this step. But you should wait to do so until everything is working as it should, in case you need to refer to these notes during development.

Note: Consider adding a target attribute set to `_blank` on the `<a>` tags for the live links to your projects so that they open in a new window.

Layout, CSS and styles

The layout of the finished project should match the provided mockups.

(optional) To really make this project your own, you should customize the CSS, changing colors, fonts or anything to make your app look better.

Add good code comments

Finish and Submit. I may provide you with alternative submission instructions.