1. **What is Node.js?**

Before Node, JavaScript could not be used as a more general-purpose programming language, it was limited to what the browser allowed it to do. Example it can assign button clicks, but it cannot configure web servers or access file system and databases, which other languages are able to do.

Node allows to run JavaScript code on server as opposed to being forced to run on a client in the browser.

**“Node.js is a JavaScript runtime built on Chrome’s V8 JavaScript engine”**

V8 is a Google open-source project that powers Google Chrome. The job of any JavaScript engine is to take in JavaScript code and convert into machine code, which can be executed by the machine. V8 is written in C++. Anyone can write their C++ application, incorporate V8 JavaScript in their application and can extend the functionality that JavaScript provides. Classic examples of this are Chrome and Node.js. Chrome and Node.js are both creating their own modified versions of JavaScript, which is called JavaScript runtime.

So, when Chrome wants to run JavaScript for a particular web page, it can’t run itself, instead it uses V8 engine to get that done. It passes JavaScript code to V8 engine and gets the results back (after converting). The same thing is true for Node. Node can’t run JavaScript all by itself. It uses V8 for that conversion.

V8 is written in C++ and both Node and Chrome are largely written in C++. This is because they both provide bindings when they are instantiating the V8 engine. This is what allows them to create their own JavaScript run time with interesting and novel features. It’s what allows Chrome to interact with DOM, when the DOM isn’t part of JavaScript and it allows Node to interact with file system, when file system isn’t part of JavaScript either. V8 has no idea about DOM or file system, it is up to Node and Chrome to provide implementations for those when running V8.

Diagram

Description automatically generated with medium confidence

In Chrome we can execute JavaScript commands and get the results. Here, notice we can get a window object and inspect its objects.

Graphical user interface, text, application, email, website

Description automatically generated

In Node also, we can run JavaScript commands, but here we don’t have any window object. This makes sense because there is no concept of DOM manipulation in Node. Run process.exit() to exit.

Graphical user interface, text, application, email

Description automatically generated

1. **Why learn Node.js?**

Node.js uses an event driven, non-blocking I/O model that makes it lightweight and efficient.

A screenshot of a computer

Description automatically generated with medium confidence

Node.js’ package ecosystem, **npm**, is the largest ecosystem of open-source libraries in the world.

1. **Your First Node.js script**

Graphical user interface, text

Description automatically generated

1. **Node.js module system**
   1. **Importing Node.js Core Modules**

const fs = require('fs')

fs.writeFileSync('notes.txt', 'This file was created by Node.js!')

* 1. **Importing your own files**

//utils.js

console.log('utils.js')

const name = 'Deepak'

//Only exported items will be available when this file is imported

module.exports = name

//app.js

const name = require('./utils')

console.log(name) //utils.js

//Deepak

* 1. **Importing Local npm modules**

1. **Check version of npm**



1. **Initialize npm**

This will create a configuration file (**package.json**). During init, it will ask for various configurations to be written onto this file. Alternatively, we can use **npm init -y** to skip the configuration questions and default them to yes for each question.

Text

Description automatically generated

1. **Install a particular npm module using npm i <package-name>**
   1. **Installing and using Global npm module using npm i <package-name> -g**

**Text

Description automatically generated**

Now we can use **nodemon <file.js>** to run our node scripts, but it will also keep listening to any changes to the script and re-run the script again.

A picture containing text

Description automatically generated

1. **File System and Command Line Args (Notes app)**
   1. **Getting input from users**

Can use **nodemon app.js <command line args>**

In the script we can access command line argument in process.argv array:

console.log(process.argv);

Graphical user interface, text

Description automatically generated

* 1. **Argument Parsing with Yargs**

Yargs is a popular NPM module for argument parsing.

console.log(process.argv);

console.log(yargs.argv);

Text

Description automatically generated

**5.2.1 Adding and Executing a command:**

const yargs = require('yargs');

//Customize yargs version

yargs.version('1.1.0');

//Create add command

yargs.command({

command: 'add',

describe: 'Add a new Note',

handler: function() {

console.log('Adding a new Note!')

}

});

//Create remove command

yargs.command({

command: 'remove',

describe: 'Remove a Note',

handler: function() {

console.log('Removing the Note!')

}

});

console.log(yargs.argv);

Displaying all the options using help:

Graphical user interface

Description automatically generated with medium confidence

Executing a command:



**5.2.2 Adding options to a command:**

Add builder property to the command to specify the options that the command expects:

yargs.command({

command: 'add',

describe: 'Add a new Note',

builder: {

title: {

describe: 'Note title',

demandOption: true, //Is option required(mandatory) ?

type: 'string' //Data type of the option

}

},

handler: function(argv) {

console.log('Adding a new Note!', argv.title)

}

});

Since we have made title as a required argument, the command is giving an error, when we execute it without title.

Text

Description automatically generated

**5.2.3 Storing data with JSON:**

**JSON Methods:**

const book = {

title: 'Ego is the Enemy',

author: 'Ryan Holiday'

}

//Stringified version of object

const bookJSON = JSON.stringify(book);

console.log(bookJSON);

console.log(bookJSON.title); //undefined

//Parsed data

const parsedData = JSON.parse(bookJSON);

console.log(parsedData);

console.log(parsedData.title); //Ego is the Enemy

**Writing to the file:**

const fs = require('fs');

//JavaScript Object

const book = {

title: 'Ego is the Enemy',

author: 'Ryan Holiday'

}

//Stringified version of object

const bookJSON = JSON.stringify(book);

//Writing the json data to a file.

fs.writeFileSync('1-json.json', bookJSON);

**Reading from the file:**

const fs = require('fs');

//Read the file.

const dataBuffer = fs.readFileSync('1-json.json');

console.log(dataBuffer);

const dataJSON = dataBuffer.toString();

console.log(dataJSON);

const data = JSON.parse(dataJSON);

console.log(data.title);

**5.2.4 ES6 Arrow functions:**

//Normal function

const square = function(x) {

return x \* x

}

//Arrow function

const square = (x) => {

return x \* x

}

//Short version, when function is simply returning and not doing anything extra

const square = (x) => x \* x

const event = {

name: ‘Birthday Party’,

guestlist: [‘Andrew’, ‘Jen’, ‘Mike’],

//Normal functions; they have their own this binding

printGuestList: function() {

console.log(this.name); //Birthday Party

this.guestList.forEach(function(guest) {

console.log(this.name) //undefined

});

}

//Arrow functions; they don’t bind their own this value

printGuestList: () => {

console.log(this.name); //undefined

this.guestList.forEach((guest) => {

console.log(this.name) //Birthday Party

});

}

//Using ES6 method shorthand

printGuestList() {

console.log(this.name); //Birthday Party

this.guestList.forEach((guest) => {

console.log(this.name) //Birthday Party

});

}

}

1. **Debugging**
   1. **Debugging Node.js**

Add **debugger** statement anywhere in the code where you want to debug. Then run the script in inspect mode using **node inspect <file you want to execute>** command **<args>**.

Text

Description automatically generated

Text

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

1. **Asynchronous Node.js (Weather App)**
   1. **Asynchronous Basics: Call Stack, Callback Queue and Event Loop**

[Refer the Key Note](node_course/4.Asynchronous_Node.js/3-async-nodejs.key)

* 1. **Making HTTP Requests**

We will be using npm package, **‘request’** to make HTTP requests. Also, to get weather data for our app we will be using [weatherstack.com](https://weatherstack.com/) as a free service.

* 1. **ES6: Default Function Parameters (Important used in weather app)**

Simple example:

Graphical user interface

Description automatically generated

Weather app example: Here latitude, longitude and location properties are destructured from the response. But in case of error, data will be undefined, and it will throw an error if we try to destructure undefined object. To avoid this, we use default parameter set to an empty object.

Graphical user interface, text

Description automatically generated

1. **Web Servers**
   1. **Express**

<https://expressjs.com>

Install: npm install express --save

Node.js is commonly used as a web server to serve up websites, JSON, and more. Serving up websites and JSON data is easy with Express. In this lesson, you’ll learn how to create your first web server with Express. Once the server is up and running, users will be able to interact with your application via the browser.

The code below uses **app.get** to set up a handler for an HTTP GET request. The first argument is the path to set up the handler for. The second argument is the function to run when that path is visited. Calling **res.send** in the route handler allows you to send back a message as the response. This will get shown in the browser.

The last thing to do is start the server. This is done by calling **app.listen** with the port you want to listen on.

Text

Description automatically generated

Graphical user interface, application

Description automatically generated

* 1. **Serving up HTML and JSON**

Text

Description automatically generated

Graphical user interface, application

Description automatically generated

Table

Description automatically generated with low confidence

* 1. **Serving up Static Assets**

Express can serve up all the assets needed for your website. This includes HTML, CSS, JavaScript, images, and more.

**Serving up a Static Directory:** A modern website is more than just an HTML file. It’s styles, scripts, images, and fonts. Everything needs to be exposed via the web server so the browser can load it in. With Express, it’s easy to serve up an entire directory without needing to manually serve up each asset. All Express needs is the path to the directory it should serve.

The example below uses Nodes’ **path** module to generate the absolute path. The call to **path.join** allows you to manipulate a path by providing individual path segments.

In below example we have set up a public directory and created a simple index.html file. And with path manipulation we are able to render that html file in browser.

Text

Description automatically generated

Start the server, and the browser will be able to access all assets in the public directory.

Text

Description automatically generated

Graphical user interface, application

Description automatically generated

**Serving up CSS, JS, Images, and More**:

All files in public are exposed via the Express server. This is where your site assets need to live. If they’re not in public, then they’re not public and the browser won’t be able to load them correctly.

* 1. **Dynamic Pages with Templating**

Your web pages don’t have to be static. Express supports templating engines that allow you to render dynamic HTML pages. In this lesson, you’ll learn how to set up the Handlebars templating engine with Express.

Install handlebars with **npm i hbs**

use **app.set** to set a value for the **'view engine'** config option:

app.set('view engine', 'hbs')

**Rendering Handlebar Templates**

By default, Express expects your **views** to live in a views directory inside of your project root. You’ll learn how to customize the location and directory name in the next lesson.

Below is an example of handlebars view in **views/index.hbs**. This looks like a normal HTML document with a few new features. Notice {{title}} and {{name}}. This is a Handlebars syntax which allows you to inject variables inside of the template. This is what allows you to generate dynamic pages.

Text

Description automatically generated

Now, you can render the template. This is done by defining a new route and calling **res.render** with the template name. The ‘**.hbs**’ file extension can be left off. The second argument is an object that contains all the variables the template should have access to when rendering. This is where values are provided for **title** and **name**.

app.get('', (req, res) => {

res.render('index', {

title: 'My title',

name: 'Andrew Mead'

})

})

**Customizing the views Directory**

You can customize the location of the views directory by providing Express with the new path. Call **app.set** to set a new value for the ‘**views**’ option. The example below configures Express to look for views in **templates/views**.

const viewsPath = path.join(\_\_dirname, '../templates/views')

app.set('views', viewsPath)

* 1. **Advanced Templating**

In this lesson, you’ll learn how to work with Handlebars partials. As the name suggests, partials are just part of a web page. Partials are great for things you need to show on multiple pages like headers, footers, and navigation bars.

**Setting up Partials**

You can use partials by telling Handlebars where you’d like to store them. This is done with a call to **hbs.registerPartials**. It expects to get called with the absolute path to the partials directory.

const partialsPath = path.join(\_\_dirname, '../templates/partials')

hbs.registerPartials(partialsPath)

**Using Partials**

Partials are created with the ‘**hbs**’ file extension. Partials have access to all the same features as your Handlebars templates. The header partial below renders the title followed by a list of navigation links which can be shown at the top of every page.

//header.hbs file

<h1>{{title}}</h1>

<div>

<a href="/">Weather</a>

<a href="/about">About</a>

<a href="/help">Help</a>

</div>

The partial can then be rendered on a page using where “**header**” comes from the partial file name.

<!DOCTYPE html>

<html>

<head>

<link rel="stylesheet" href="/css/styles.css">

<script src="/js/app.js"></script>

</head>

<body>

{{>header}}

</body>

</html>

Customizing **nodemon** command to listen to changes to specific file extensions. By default nodemon does not listens to changes to .**hbs** files. We can manually change this behaviour by:

**nodemon src/app.js -e js,hbs**

1. **Accessing API from browser**

In this section, you’ll learn how to set up communication between the client and the server. This will be done via HTTP requests. By the end of the section, users will be able to type an address in the browser to view their forecast.

* 1. **Query String**

In this lesson, you’ll learn how to use query strings to pass data from the client to the server. This will be used to send the address from the browser to Node.js. Node.js will then be able to fetch the weather for the address and send the forecast back to the browser. The query string is a portion of the URL that allows you to provide additional information to the server. The query string comes after **?** in the URL. The example URL below uses the query string to set **address** equal to **boston**. The key/value pair is separated by **=**

**http://localhost:3000/weather?address=boston**

The Express route handler can access the query string key/value pairs on **req.query**. The handler below uses **req.query.address** to get the value provided for **address**. This address can then be used to fetch the weather information.

app.get('/weather', (req, res) => {

// All query string key/value pairs are on req.query

res.send('You provided "' + req.query.address + '" as the address.)

})

* 1. **Browser HTTP Requests with Fetch**

In this lesson, you’ll learn how to make HTTP AJAX requests from the browser. This will allow the web application to request the forecast from the Node.js server.

Web APIs provide you with a way to make HTTP requests from JavaScript in the browser. This is done using the **fetch** function. **fetch** expects to be called with the URL as the first argument. It sends off the HTTP request and gives you back the response.

fetch('http://localhost:3000/weather?address=Boston')

.then((response) => {

response.json()

.then((data) => {

if(data.error){ console.log(data.error)

}

else{

console.log(data.location) console.log(data.forecast)

}

})

})

1. **Application Deployment**

In this section, you’ll learn how to deploy your Node.js applications to production. In this lesson, you’ll join **GitHub** and **Heroku**. GitHub is a development platform that makes it easy to manage software development projects. Heroku is an application deployment platform which provides everything needed to deploy your Node.js applications.

The Heroku CLI gives you commands to deploy and manage your Node.js applications. Before you can do that, you’ll need to log in to your Heroku account. This makes sure that the command you run actually changes your Heroku applications.

Text

Description automatically generated

**10. 1 Setting up SSH Keys**

SSH is the protocol used to securely transfer code between your machine and GitHub/Heroku.

SSH uses an SSH key pair to secure the connection between your machine and the machine you’re communicating with. You can check if you already have an SSH key pair with the following command. You have a key pair if you see **id\_rsa** and **id\_rsa.pub** in the output.



You can create a new key pair using the following command. Make sure to swap out the email for your email address. And I’ve kept default options.

Graphical user interface, text, application, email

Description automatically generated

Check again for ssh directory:

A picture containing text

Description automatically generated

**id\_rsa** is a secret file, not to be shared with anyone. It is a Private key

**id\_rsa.pub** is a public file and can be shared. It is a Public key.

The SSH key needs to be configured to be used for new SSH connections. First, ensure

that the SSH agent is running. You can do that using the command below.



Next, add the new SSH private key file to the SSH agent. The following command is for

macOS users.



**10.2 Pushing code to GitHub**

1. **Creating a Remote repository**

You need to create a new GitHub repository before you’ll be able to push your code. This is a remote Git repository that’ll live on the GitHub server. A **remote** repository is nothing more than a version of your project hosted somewhere else. In this case, it’s a version of your project stored on GitHub.

1. **Adding the remote**

Once the repository is created, you’ll need to set up the origin remote. Replace <**repo url**> with the repository URL provided by GitHub.

git remote add origin <repo url>

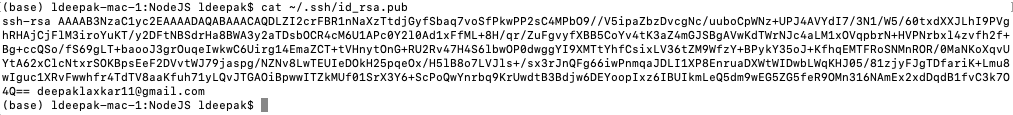
1. **Configuring SSH with GitHub**

Go to GitHub and under settings find SSH and GPG keys section.

Graphical user interface, text, application, email

Description automatically generated

The command below will allow you to dump the contents of the public key file to the terminal. Copy and paste the contents to the clip board and register the SSH key with GitHub.



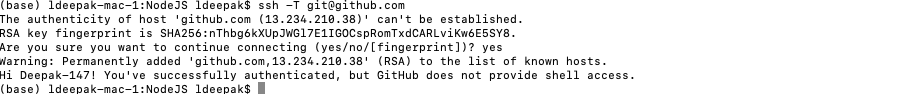
Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Then we can test our connection using below:



1. **Pushing the code**

You can now push your latest commits to the remote! After pushing your commits, refresh the GitHub repository page in your browser to see your project files and folder appear.

**git push -u origin master**

**10.3 Deploying Node.js to Heroku**

1. **Add SSH key**



1. **Create new Heroku application**



When you create an app, a git remote (called **heroku**) is also created and associated with your local git repository. We can verify the remotes as below:

Graphical user interface, text

Description automatically generated with medium confidence

There are 2 URLs from create command above:

<https://ldeepak-weather-app.herokuapp.com/> is the URL for our app.

<https://git.heroku.com/ldeepak-weather-app.git> is the URL for the git repository where we should be pushing our code, we want to deploy.

You can open your application using the first URL from the above command. But it can’t display our app since it is not deployed to Heroku servers yet.

Graphical user interface, application, Word, Teams

Description automatically generated

For Heroku to run our application, we need to make some changes. First, Heroku needs to know what command to run to start your app. Second, Heroku requires your app to listen on a specific port.

The **start** script in **package.json** is used to tell Heroku which command to run to start our app. Normally you can set **start** equal to **node src/app.js**. This ensures that Heroku can start your app up correctly.

Heroku uses an environment variable to provide the port value you need to listen on. The code below accesses the Heroku port value and uses it to start up the server.

const port = process.env.PORT || 3000

app.listen(port, () => {

console.log('Server is up on port ' + port)

})

Once all changes are in place, we can push our code to GitHub.

Next, we need to push our code to Heroku servers (or deploy the code)

<https://jtway.co/deploying-subdirectory-projects-to-heroku-f31ed65f3f2>

Text

Description automatically generated

Finally, we can use **heroku open** to open our app in browser (or alternatively we can copy and paste the app URL)

1. **SQL vs NoSQL**

|  |  |
| --- | --- |
| **SQL** | **NoSQL** |
| Structured Query Language | Not only SQL |
| Database consists of one or more tables | Database consists of one or more collections |
| Each table has columns | Each collection has Fields |
| Group of 1 or more row/record forms a table | Group of 1 or more Document forms a collection |

**11.1 Installing MongoDB**

You can download the MongoDB Community Server from the MongoDB download page. The download is a zip file. Unzip the contents, change the folder name to ‘**mongodb**’, and move it to your users home directory. From there, create a ‘**mongodb-data**’ directory in your user directory to store the database data.

You can start the server using the following command. Make sure to swap out “/Users/Andrew/” with the correct path to your users home directory.

Graphical user interface, text

Description automatically generated

A screenshot of a computer

Description automatically generated with low confidence

**11.2 Installing Database GUI Viewer**

In this lesson, you’ll set up **Robo 3T**. Robo 3T is a MongoDB admin tool that makes it easy to manage and visualize the data in your database.

Robo 3T is a completely free MongoDB admin tool. Grab the installer from [here](https://robomongo.org/) and get it installed on your machine.