**Express.js Notes**

**Introduction**

* Express is a js framework that helps us easily manage our routing, requests, server-side logic, and responses in a way that is more elegant and extendable than just using Node.js
* While everything done with express can be done with just node, express makes the development process a lot easier and cleaner

**Installing express**

* Make sure we already have node installed
* Make sure we have a package.json file in the root folder, (if not run ‘npm init -y’)
* Install express by running ‘npm i express’

**Creating an express app**

* A new file will be created for our express app, the convention is to name it ‘app.js’
* First, we will need to require the ‘express’ module which returns a function which we are storing in a constant called ‘express’. We can then execute this ‘express’ function to return an instance of an express app which we will store in a constant called ‘app’
* 
* Now, we have to set up the server to listen for requests by calling the ‘app’ constant’s ‘listen’ method. This function takes in the port number as its first argument. The second argument is optional and takes in the IP address of the host. If no second argument is passed in, the IP address will default to localhost. The third argument is a callback function that gets executed when the server starts. This ‘listen’ method returns an instance of the server which we can reuse later on for something else (but generally, we don’t use the return value of the ‘listen’ method).
* 
* To respond to requests, we will need to use the ‘app’ constant. This ‘app’ constant also provides a ‘get’ method which allows us to listen to get requests. This ‘get’ method takes in two arguments. The first argument is the path we want to listen to such as ‘/’ which is the root of our domain. The second argument is a function. This function takes in a request and response object and gets executed whenever there is a get request sent to the path specified in the first parameter.
* 
* This callback function allows us to send responses which we can do via res.write() and then res.end() much like node. But express provides a new method in the response object called ‘send’. This method infers the type of content that we are responding with (so the content-type header is automatically set). This method also infers the status code of the request.
* Text

  Description automatically generated
* Now if we run this app.js file by running in cmd ‘node app’ and we go to localhost:3000, we see the following
* Logo

  Description automatically generated with medium confidence
* A screenshot of a computer

  Description automatically generated with medium confidence
* Notice the status code and content-type are automatically set.
* While we can use ‘res.send’, there are some alternatives
  + res.sendStatus(INSERT\_STATUS\_CODE) returns a status code of INSERT\_STATUS\_CODE
    - Ex: res.sendStatus(404) returns a 404 error
  + res.json(INSERT\_JSON\_OBJECT) returns a json object
    - Ex: res.json({ name: ‘Grant’ }) returns the json object: { name: ‘Grant’ }
  + res.download(INSER\_FILE\_PATH) returns a file that will be automatically downloaded
    - Ex: res.json(‘app.js’) returns the ‘app.js’ file which automatically gets downloaded by the browser

**Routing and HTML pages**

* We can handle different routes by having multiple different ‘get’ handlers.
* Text

  Description automatically generated
* Now, if we go to <http://localhost:3000/about> we see the following:
* 
* However, we don’t want to write HTML in this app.js file. Instead, we want to respond with an HTML file.
* To do so, we can create a new folder called ‘views’ and inside it, we can create a ‘about.html’ and ‘index.html’ page. These are just html files with the words ‘about’ and ‘index’ in the body.
* To send back a file, we use the response object’s ‘sendFile’ method. This first parameter is a relative path to the file. This second parameter takes in an object which has a ‘root’ property which specifies what the path in the first parameter is relative to. By default, the value of the second parameter is the path from our root of our computer which essentially makes the path an absolute path.
* To make the path specified in the first parameter relative to the root folder of our project, we set the value associated with the ‘root’ object to be \_\_dirname. We can also specify this root folder using the path module.
* Text

  Description automatically generated
* Now, we can run our server and go the <http://localhost:3000/> and <http://localhost:3000/about> and we see the ‘index.html’ and ‘about.html’ pages being rendered.

**Redirect**

* To redirect from a given url (lets call it url1) to a new url (call it url2), we first set up a get request handler for url1. To redirect them to url2, run res.redirect(url2).
* Under the hood, automatically sets the status code as well
* Text

  Description automatically generated

**404 Pages**

* To create a 404 page, we make use of the ‘app’ constant’s ‘use’ method. This ‘use’ method allows us to create middleware (we will talk more about this later) and fire middleware functions.
* This ‘use’ method takes in a callback function. This callback function takes in a request and response object much like the callback functions in ‘app.get’. Inside this callback function, we return an HTML page
* Text

  Description automatically generated
* This ‘use’ function is fired for every single request coming in, but only if it is reached.
* When a request comes in, express is going to run through the app.js file from top to bottom and it’s going to look through each of the ‘app’ constant’s http request handler methods such as ‘get’. If there is a match for the request’s url, the callback function is executed and express no longer carries on down the code (regardless if a response object was sent in the callback function). If there is no match for the request’s url, the callback function is not executed and we continue down the file.
* If we keep going down the file and get to the ‘app.use’ method, the ‘use’ method’s callback function is executed, regardless of the request’s url (which is why we didn’t need to pass in a url as a parameter). Regardless of whether there is a response object being sent in the ‘use’ method, express no longer carries on down the code.
* Ex:
  + Text

    Description automatically generated
  + If we go to <http://localhost:3000/wtf> the request method’s url is ‘/wtf’.
  + We go from the top to the bottom of the file, looking for http request handler methods.
  + The first http request handler method we see is ‘app.get(‘/’)’ on line 4. Since the paths ‘/’ and ‘/wtf’ don’t match, we carry on to the next request handler method.
  + The next http request handler method we see is ‘app.get(‘/about’)’ on line 8. Since the paths ‘/about’ and ‘/wtf’ don’t match, we carry on to the next request handler method.
  + The next http request handler method we see is ‘app.get(‘/about-us’)’ on line 12. Since the paths ‘/about-us’ and ‘/wtf’ don’t match, we carry on to the next request handler method.
  + The next http request handler method we see is ‘app.use()’ on line 16. Since we reach this ‘app.use()’ method, its callback function is executed and send a response containing a the 404.html file that will be rendered by the browser. We no longer carry on to the next request handler method.
  + Thus, we see the following in the browser.
  + 
* Ex:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + If we go to <http://localhost:3000/> the request method’s url is ‘/’.
  + We go from the top to the bottom of the file, looking for http request handler methods.
  + The first http request handler method we see is ‘app.use()’ on line 4. Since we reach this ‘app.use()’ method, its callback function is executed and send a response containing a the 404.html file that will be rendered by the browser. We no longer carry on to the next request handler method.
  + Thus, we see the following in the browser:
  + 
  + So even though there are matching urls in other http request handler methods later down the code, it doesn’t matter since they are never reached.
* Ex:
  + Text

    Description automatically generated
  + If we go to <http://localhost:3000/> the request method’s url is ‘/’.
  + We go from the top to the bottom of the file, looking for http request handler methods.
  + The first http request handler method we see is ‘app.use()’ on line 4. Since we reach this ‘app.use()’ method, its callback function is executed and we log ‘hi’ to the console. We no longer carry on to the next request handler method. Thus, the app.use() method on line 8 is never reached.
  + Thus, we just see a loading page in the browser as no response object was sent.
* As of now, we returned a 404.html page, but express doesn’t know that the response has a status code of 404 so we have to manually set it. We can chain on the redirect after the status method since the status method returns the response object itself.
* Text

  Description automatically generated
* Now in the browser, we see the 404.html page and if we check the networks tab in developer tools, we see the 404 error status.

**View Engines**

* We are currently serving static HTML files. By static, we mean all the content within the HTML file is predefined and never changes.
* However, we may want to inject dynamic data into this HTML file.
* To do so, we will use a view engine (or template engine) which is something express apps can use very easily.
* View engines allow us to write HTML templates that resemble HTML syntax but also allows us to inject dynamic data and logic into them like variables, if statements, and loops.
* There are different view engines such as express-handlebars, pug, or ejs
* We will use ejs

**EJS view engine**

* To install it, run ‘npm i ejs’
* To tell express that we want to use ejs as our view engine of choice, we need to use the ‘app’ constants ‘set’ method. This ‘set’ method allows us to configure settings such as the view engine we will use. To configure the view engine to be ejs, do the following: app.set('view engine', 'ejs')
* Automatically, express is going to look inside a folder called ‘views’ for any ejs files.
* If we want express to look in some other folder for our ejs files, we have to use the ‘set’ method once again. We pass ‘views’ as the first parameter to the ‘set’ method and path.join(\_\_dirname, './myViews') as the second parameter.
* A screenshot of a computer

  Description automatically generated with medium confidence
* To create a view inside the ‘views’ folder, we create a .ejs file. For example, let’s create index.ejs
* Text

  Description automatically generated
* Inside this index.ejs file, we include HTML syntax.
* A screenshot of a computer

  Description automatically generated
* Now, we want to serve this index.ejs file as a response to the browser.
* Inside app.js, we no longer want to use res.sendFile to return an HTML file.
* Instead, we want to render a view by using res.render(‘VIEW\_FILE\_NAME’) as shown below
* Text

  Description automatically generated
* Notice we don’t include the .ejs extension, only the file name. This res.render(‘index’) tells express to look for a ‘index.ejs’ file (we already know the extension is ejs since we used app.set to configure that) inside the ‘views’ folder (since that is the default folder when we set up the ejs view engine).
* Now, if we go to <http://localhost:3000/> we see the following ejs file being rendered.
* 

**Passing Dynamic Data Into Views**

* To use dynamic data, we use ejs tags, the opening and closing tags are <% and %> respectively.
* Inside these tags, we can write javascript.
* For example, <% const hello = 123 %> defines a new constant called ‘hello’ with a value of 123 that we can use later on in the ejs file.
* If we want to return a value to be displayed, we use the <%= %> tags.
* For example <%= hello %> returns the value of the ‘hello’ constant which is 123. Thus 123 is displayed
* Ex:
  + Text

    Description automatically generated
  + Now, if we return the above .ejs file, the browser would display the following.
  + Text

    Description automatically generated with medium confidence
  + Notice the value of the ‘hello’ constant which is 123 is shown inside <h1> tags.
* Now, we want to pass data from our app.js file into our ejs file. A real world application fetch some data from the database and then pass that data to the ejs file.
* To replicate this, we will pass dummy data into the templates.
* In the ‘res.render’ method, we can pass in a second parameter which is an object. This object will be passed into the ejs file. Thus, we can access any properties within this object from inside the ejs file.
* Ex:
  + App.js:
  + Graphical user interface, text

    Description automatically generated
  + Index.ejs:
  + Text

    Description automatically generated
  + Output:
  + Text

    Description automatically generated
  + Notice we were able to access the ‘name’ property of the { name : ‘grant’ } object that was passed in the second parameter of the ‘render’ method. The value associated with this ‘name’ property is ‘grant’ which is what is rendered in between the <h1> tags.
* Suppose we want to pass in an array to an ejs file and we want to render some HTML for each element in the array.
* Ex:
  + App.js
  + Text

    Description automatically generated
  + Index.ejs
  + Text

    Description automatically generated
  + Output:
  + Text, letter

    Description automatically generated
  + Notice above we first check if the length of the ‘blogs’ array is greater than 0 (which it is). Since that is true, we then go on to iterate through each item in the ‘blogs’ array and return an <h3> and <p> tag for each item.
  + If the value of ‘blogs’ array was [], we would instead return an <h3> tag with ‘No blogs’ rendered in between the tags.
* EJS templates are processed through the EJS view engine on the server. Our .ejs files live on the server. When we want to render one to the browser, that .ejs file is passed into he EJS view engine to be processed. This engine looks for any kind of dynamic content (such as variables, loops, conditionals, etc) and find the resulting HTML code. This EJS view engine then returns a valid HTML file based on the template we wrote and that HTML page is returned to the browser. This process is called server-side rendering.

**Partials**

* Partials are parts of an ejs template that can be reused in different views
* To import a partial, we use <%- include(‘RELATIVE\_PATH\_TO\_PARTIAL.ejs’) %>
* Note that we used <%- and note <%=. This is because when we us <%=, we escape special characters and end up with a string instead of raw HTML.
* Ex:
  + Graphical user interface, text, application

    Description automatically generated
  + Index.ejs:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Hello.ejs:
  + 
  + Now, when we go to <http://localhost:3000/> which renders the index.ejs page, we see:
  + Logo

    Description automatically generated
* We can also passing dynamic data to partial templates by adding a second parameter to the includes method. This second parameter is an object that has key value pairs, and we can reference the keys in the partial to get the associated value.
* Ex:
  + Index.ejs
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Hello.ejs
  + 
  + Output:
  + 