**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 (unless the next() method is called). 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 (unless the next() method is called).
* 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 since the next() method is not used.
  + 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 since the next() method is no longer used.
  + 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 since the next() method is never used. 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:
  + 

**Middleware**

* Middleware is the name for any code that runs on the server between getting a request and sending a response.
* For example, the use() and get() methods are example of middleware
* We could have multiple pieces of middleware that run in the sever as shown in the example below
* A screenshot of a computer

  Description automatically generated with medium confidence
* In the above example, three functions are executed before a response is sent, all three functions are considered middleware
* Middleware runs from the top of our app.js page to the bottom until we finish the process or explicitly send a response to the browser
* Ex:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + If we go the the ‘/’ route, the request is sent to the server.
  + The first piece of middleware is the use() method which gets executed
  + The second piece of middleware is the get(‘/’) method which gets executed. Since the get(‘/’) method’s route of ‘/’ matches the request’s route of ‘/’, we send a response to the browser. Since a response is sent, no other middleware is executed so the bottom app.use() function is not executed.
* Examples of middleware:
  + Logger middleware to log details of every request
  + Authentication check middleware for protected routes
  + Middleware to parse json data from post requests
  + Return 404 pages
* Ex:
  + App.js:
  + Text

    Description automatically generated
  + Suppose we go to <http://localhost:3000/>
  + We made a request, and we start from the top of the app.js file and make our way down.
  + The first piece of middleware is the use() method which logs out the request’s hostname, path, and method.
  + Output:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + However, if we check the browser, we see that a response is never sent back. This is because after we run the use() method which is the first piece of middleware, express doesn’t know how to move on to the next piece of middleware which is the get(‘/’) method.

next()

* To explicitly tell express to move on the next piece of middleware, we need to use a method called next().
* We get access to this method in the parameters of the callback function of the middleware functions such as use() and get().
* We can then execute this next() method inside the callback function.
* Ex:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + If we go to <http://localhost:3000/>, we get the following logged to the console:
  + Text

    Description automatically generated
  + We also the get the ‘index.ejs’ file converted to an HTML file and rendered in the browser.
  + We start from the top of the file and reach the first app.use() method. In this method, we log out information about the request in the console and tell express to go move on to the next piece of middleware via the next() method.
  + We then move on to the app.get(‘/’) method which is the following piece of middleware. Since the get(‘/’) method’s route of ‘/’ matches the request’s route of ‘/’, the get() method’s callback function is executed. Since a response was sent inside the get(‘/’) method’s callback, all following middleware is not executed so ‘some random middleware is not logged’
  + If we go to <http://localhost:3000/about>, we get the following logged to the console:
  + Text

    Description automatically generated
  + We also the get the ‘about.ejs’ file converted to an HTML file and rendered in the browser.
  + We start from the top of the file and reach the first app.use() method. In this method, we log out information about the request in the console and tell express to go move on to the next piece of middleware via the next() method.
  + We then move on to the app.get(‘/’) method which is the following piece of middleware. Since the get(‘/’) method’s route of ‘/’ does not matches the request’s route of ‘/’, we move on to the next piece of middleware.
  + We then move on to the app.use() method. In this method, we log out ‘some random middleware’ and tell express to go move on to the next piece of middleware via the next() method.
  + We then move on to the app.get(‘/’) method which is the following piece of middleware. Since the get(‘/about’) method’s route of ‘/about’ matches the request’s route of ‘/about’, the get(‘/about’) method’s callback function is executed. Since a response was sent inside the get(‘/about’) method’s callback, all following middleware is not executed.
* Ex:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + If we go to <http://localhost:3000/>, we get the following logged to the console: Text

    Description automatically generated
  + We also get the ‘404.ejs’ file converted to an HTML file and rendered in the browser.
  + We start from the top of the file and reach the first app.use() method. In this method, we log out information about the request in the console and tell express to go move on to the next piece of middleware via the next() method.
  + We then move on to the app.get(‘/’) method which is the following piece of middleware. Since the get(‘/’) method’s route of ‘/’ matches the request’s route of ‘/’, the get(‘/’) method’s callback function is executed. In this method, we log out ‘Yo’ to the console and tell express to go move on to the next piece of middleware via the next() method.
  + We then move on to the app.use() method. In this method, we log out ‘some random middleware’ and tell express to go move on to the next piece of middleware via the next() method.
  + We then move on to the app.get(‘/’) method which is the following piece of middleware. Since the get(‘/about’) method’s route of ‘/about’ does not match the request’s route of ‘/’, we move on to the next piece of middleware.
  + We then move on to the app.get(‘/about-us’) method which is the following piece of middleware. Since the get(‘/about-us’) method’s route of ‘/about-us’ does not match the request’s route of ‘/’, we move on to the next piece of middleware.
  + We then move on to the app.use() method. In this method, we log out ‘YO’ and send a response containing the ‘404.ejs’ file. Since a response was sent inside the use() method’s callback, all following middleware is not executed.

**3rd party middleware**

* There are many middleware functions that are already created for us to use
* For example, there’s middleware for logging, security, sessions/cookies validation, etc
* For example, we can install the morgan logging middleware by running ‘npm i morgan’
* Now, in our app.js file, we can require the package and invoke the morgan function inside an app.use
* Text

  Description automatically generated
* Now if we go to <http://localhost:3000/>, we see the following logged to the console:
* 

**Static Files**

* If we add static files to our root folder (such as images or CSS files), we wouldn’t be able to access them from the browser.
* Ex:
  + A picture containing text

    Description automatically generated
  + Output:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Notice we don’t get a css file returned.
  + Since we can’t access the css file from the browser, even if we place a link to the css file inside the ejs templates (as shown below), it still doesn’t work.
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + A picture containing text, scoreboard, screenshot

    Description automatically generated
* The reason we can’t access the style.css file is because the server automatically protects all of our files from the browsers (so the client can’t just access any file from the backend)
* In order to access toe style.css file from the browser, we need to specify which files should be accessible for the browser.
* To do so, we can make use of the static middleware via the ‘express’ constant’s ‘static’ method. This method takes in a folder name, MY\_FOLDER\_NAME\_EXAMPLE, as its parameter. Now, if we create a folder called MY\_FOLDER\_NAME\_EXAMPLE in the root folder, any files inside the MY\_FOLDER\_NAME\_EXAMPLE will be made available as a static file to the browser.
* Ex:
  + A picture containing text

    Description automatically generated
  + app.js:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + In the app.js file, we set any files inside the ‘public’ folder to be public.
  + style.css:
  + Graphical user interface

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

    Description automatically generated with medium confidence
  + Notice in the index.ejs file, we don’t set the link’s href value to be ‘public/style.css’, but rather just ‘style.css’. This is because we already specified in app.js that our static files will be the ‘public’ folder.
  + Now, if we go to <http://localhost:3000/>, we get the following in the browser:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Graphical user interface, application

    Description automatically generated
  + Now, we are able to access the style.css file from the index.ejs file and from the browser.
  + Graphical user interface, text, application

    Description automatically generated

Request Types

* GET requests are used to get a resource
* POST requests are used to create new data
* DELETE requests are used to delete new data
* PUT requests are used to update new data
* Ex:
  + A picture containing graphical user interface

    Description automatically generated
  + Notice above, the same url of ‘localhosts:3000/blohs’ is used twice, but because they have different requests types (one is GET and the other is POST), the server can handle them differently. A similar idea applies to the url of ‘localhost:3000/blogs:/:id’
  + Notice if we want to update a blog for example, we pass in the id of the blog we want to update in the url

**POST Request**

* To emulate a post request, we could create a form on the frontend and that form data will be sent to the server.
* In the frontend (in our index.ejs file), we can create the following form:
* A screenshot of a computer

  Description automatically generated with medium confidence
* Notice in the form tag above, we have an action attribute which has a value that represents which path this form data should be sent to. In the above example, this path is ‘/’. Also notice in the form tag, we have a method attribute which represents the request method. In the above example, this request method is ‘POST’. Thus, when we submit the form, a POST request will be sent to the url of ‘/’.
* Notice in the input tag, we had a name property. In the above example, the value of the name property is ‘height’. When we get the form data from the server, we can access these form fields via their names (more on this in the following bullet points).
* In the backend (in our app.js file), we have the following:
* Text

  Description automatically generated
* Notice in line 13, we have a post handler via the ‘app’ constant’s ‘post’ method. We also passed in a url and a callback function into the ‘post’ method. Whenever, a POST request is made to that given url, the callback function will be executed. In our example, whenever a POST request is made to ‘/’, the callback function will be executed.
* Inside this callback function, we want to access the POST request’s body. In order to do this, we will need to use express.urlencoded({ extended : true}) middleware (as shown on line 7). This boilerplate code of ‘express.urlencoded({ extended : true})’ is middleware that parses the request’s incoming data into a workable format and this workable data will be passed to the request object. Since we used the this middleware to parse the incoming data, we can now access it via req.body.
* Thus, if we go to <http://localhost:3000/>,
* We see the following:
* Shape, rectangle

  Description automatically generated
* If we enter a height of 23 and click enter to submit the form, a POST request is sent to ‘/’.
* From the server, we receive the request and run the middleware starting from the top of the file.
* We run the middleware in line 6 which is not really relevant to this example.
* We then run the middleware on line 7. This middleware is the app.use(express.urlencoded({ extended : true}) function which gets executed, parsing the POST request’s incoming data into a workable format and passing it to the request object. We move on to the next piece of middleware.
* We skip the app.get(‘/’) request handler since the incoming request has a method of POST, not GET. We move on to the next piece of middleware.
* We then move on to the app.post(‘/’) method which is the following piece of middleware. Since the post(‘/’) method’s route of ‘/’ matches the request’s route of ‘/’ and the request method’s type is also a POST request, the post() method’s callback function is executed. In this callback function, we log the value of req.body (which was parsed into workable data via the express.urlencoder middleware). By default, express does not allow us to access the body of a request which is why we need middleware. Thus, we get the following logged to the console:
* 
* Notice we have an object that represents the data sent by the form. Notice we can also access the form fields via the value of input field’s ‘name’ property. In the frontend, the input field’s name value was ‘height’ and that is why ‘height’ is the name of the property in the req.body object.
* In this callback function, we then respond with nothing, so we get a blank page in the browser. Since we made a response, all following middleware is not executed.
* express.json()
  + Similar to express.urlencoded({ extended : true}) which is a method inbuilt in express to recognize the incoming Request Object as strings or arrays, we also have express.json() which is a method inbuilt in express to recognize the incoming Request Object as a JSON Object. By adding the express.json() middleware as shown below, we will be able to parse incoming JSON data.
  + Text

    Description automatically generated
  + .json() vs .urlencoded()
  + express.json() expects request data to be sent in JSON format, which often resembles a simple JS object: 
  + express.urlencoded() expects request data to be sent encoded in the URL, usually in strings or arrays: 

**Route Parameters**

* route parameters are the part of a route that are variable
* Graphical user interface, text, application

  Description automatically generated
* In the above example, by saying /blogs/:id, we say the /:id part of the url is variable and can change. For example, the value could be 12345, 50, or hello.
* An example of route parameters being useful is when we want to target a specific resource. For example, if we want to update the email for a specific user, we could make a POST request with the new email address in the request’s body. We would then specify which user this new address belongs to via a route parameter such as ‘/users/1’
* To use route parameters in the frontend (index.ejs), we can do the following:
* A screenshot of a computer

  Description automatically generated with medium confidence
* Now, when we send the form, we sending a POST request to ‘/1’.
* In the backend we have the following (app.js):
* Text

  Description automatically generated
* Notice in line 19, we used the post request handler method and for the path, we passed in “/:id”. We didn’t have to name it ‘/:id’, we could name it whatever such as ‘/:lmfao’. We can then access this route parameter via the ‘req.params’ object. More specifically we access that object’s ‘id’ property since that is what we called the route parameter. If we name the route parameter to ‘/:lmfao’, we would use req.params.lmfao to access the route parameter.
* Thus, when we go the make a post request to <http://localhost:3000/1>, the server receives the request and starts running middleware from the top down.
* We run the middleware in line 6 which is not really relevant to this example.
* We then run the middleware on line 7 which parses the POST request’s incoming data into a workable format and passing it to the request object. We move on to the next piece of middleware.
* We skip the app.get(‘/’) request handler since the incoming request has a method of POST, not GET. We move on to the next piece of middleware.
* We skip the app.post(‘/’) request handler since the incoming request has a path of ‘/1’, not ‘/’. We move on to the next piece of middleware.
* We then move on to the app.post(‘/:id’) method which is the following piece of middleware. Since the route of ‘/1’ matches the route of ‘/:id’, the callback function is executed. In the callback function, we get access to the route parameter of ‘1’ via req.params.id and we log it to the console:
* 
* Multiple Route Parameters
* Ex:
  + Text

    Description automatically generated
  + Notice :fruitName and :fruitColor are both route parameters.
  + If we go to <http://localhost:3000/fruit/apple/green>, we send a GET request to the server with the value of the :fruitName and :fruitColor route parameters being apple and green respectively.
  + In the server, we receive the request and log out the data to the console:  
    
  + Then, we send that object back to the browser, so the browser renders the following:
  + 

**Query Parameters**

* Suppose we go to <http://localhost:3000/?gender=male> in the browser
* In the server, we can access this query parameter of gender=male via the req.query object.
* Text

  Description automatically generated
* Notice we were able to get access of the value of ‘male’ via the req.query object’s ‘gender’ property which was also the same parameter we used int the url

**PUT/DELETE requests**

* Form’s can only send GET and POST requests, so we will need to use the fetch function to make a PUT/DELETE request from the browser.
* Index.ejs:
* Text

  Description automatically generated
* In app.js:
* Text

  Description automatically generated
* When we go to <http://localhost:3000/>, we see the following in the browser:
* 
* When we click the PUT button, a PUT request is sent to the server, and it logs out in the server console:
* 
* Then a response is sent back to the browser and the following is logged in the browser console:
* 
* When we click the DELETE button, a DELETE request is sent to the server, and it logs out in the server console:
* 
* Then a response is sent back to the browser. This response contains the JSON object {redirect : ‘/about’}.
* In the frontend, we obtain the redirect url and redirect to that url via window.location.href, sending us to <http://localhost:3000/about>

**Express Router**

* Express Router is built into Express and is used to manage our routes more efficiently
* Express Router is used to split our routes into different files and manage them in small groups (called ‘mini-apps’), so we don’t have everything in one big, messy app.js file
* Suppose our app.js file looks like the following:
* A screenshot of a computer

  Description automatically generated with medium confidence
* When we look at the routes, many of the routes have ‘/fruits’. Thus, we can split those routes into their own file since they are all concerned with the same resource of ‘fruits’.
* To split them into their own file, create a new folder called routes. Inside that folder, create a new file called fruitRoutes.js
* Text

  Description automatically generated with medium confidence
* Now cut all the routes related to the ‘fruits’ resource from the app.js file and paste them into the fruitRoutes.js file. We then make some changes to the fruitRoutes.js file to look like the following:
* Text

  Description automatically generated
* Notice we say ‘const router = express.Router()’ which creates a new instance of a Router object. We can then attach the above request handlers to the router object instead of the app object.
* Notice we export this router object.
* Now we modify app.js to look like the following:
* Text

  Description automatically generated
* Notice we imported the router object from ‘./routes/fruitRouters’ and stored it in a constant called fruitRoutes.
* To make use of this fruitRoutes router object, we say app.use(fruitRoutes) as shown above.
* Now, if we make a GET request to ‘/fruits/’, we eventually arrive at app.use(fruitRoutes) which uses the route handlers from ./routes/fruitRoutes and in this case, we send back the text of ‘/fruits/’
* We can also scope routes to specific urls.
* For example, we could have the following app.js file:
* Text

  Description automatically generated
* Notice we added the ‘/fruits’ url to the app.use() middleware
* We could also have the following fruitRoutes.js file:
* Text

  Description automatically generated
* Notice instead we removed the /fruits url prefix in request handlers. For example, router.get(‘/fruits/’) is now router.get(‘/’).
* Now, when we make a POST request to <http://localhost:3000/fruits/3>, the /fruits part of the request url matches the server’s app.use(‘fruits’) url, so the request handlers attached to the fruitRoutes router object will be used. However, only the path after the <http://localhost:3000/fruits/> is used in the fruitRoutes router object’s request handlers. Thus, only ‘/3’ is sent to the fruitRoutes router object’s request handlers. This path of ‘/3’ matches the router.post(“/:id”), and so we respond with the text of “/fruits/:id”.