**Unit 9 – Express**

1. What is express??
   1. A framework for node.js
      1. Because it’s minimal, it is **fast**.
      2. Because it’s unopinionated, we can write Express in a lot of different ways.
      3. And it can connect to any kind of database.
   2. Aims to be fast, unopinionated, and minimal
   3. Not the only Node framework, but easily the most used.
   4. Others include Hapi and Koa.
   5. API is intuitive and user-friendly.
2. Why use a framework…?
   1. Node is rather bulky and leads very quickly to callback hell, even just to set up a server and listen for a request.
   2. Express provides a lot of abstraction.
   3. No more listening for stream events!
   4. Express’s “req” and “res” objects wrap the vanilla Node request and response objects, adding tons of useful methods.
   5. It is powerful yet flexible due to *middleware design pattern.*
3. Middleware design pattern:
   1. Incoming request is passed to function “get” (a router function)
      1. And we can pass into our “get” function other functions
      2. And these are executed in order.

*A shortened syntax for making a ‘get’ request*

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

res.sendFile(path.join(\_\_dirname, './index.html'));

});

*Shortened syntax demonstrating the middleware pattern, where functions are in the middle of HTTP requests to help process those request. These are the* ***middleware functions.***

*“app.get” is a* ***router function/routing function.***

app.get('/', function1, function2, function3 (req, res) => {

res.sendFile(path.join(\_\_dirname, './index.html'));

});

* 1. In an Express application, we’ll be writing middleware to help modularize our server’s ffunctionality.
  2. Express provides built-in middleware for some common tasks
  3. And there are other open-source middleware libraries that are available.

1. Chaining middleware
   1. Express allows us to chain multiple pieces of middleware together to further modularize our functionality into separate reusable functions.

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

res.sendFile(path.join(\_\_dirname, './index.html'));

});

* 1. This will make a get request, then pass this to the middleware…
  2. In express, we have something called “next()”
  3. Middleware functions have three parameters: req, res, next.

function setCookie(req, res, next) {

someMethodToSetACookie('the cookie');

*return* next();

}

* 1. And because this is meant to be modular,
     1. **Middleware functions should not call other functions!!**
     2. **Middleware functions should not call other functions!!**
     3. **Middleware functions should not call other functions!!**
  2. Middleware takes req, res, and next –
     1. And so, we can use these objects to persist data!!
     2. On the **“res” object, there is a property called res.locals**.
     3. The locals key can be accessed, and properties to pass on can be stored as a key within “locals.”
     4. This information only persists for **a single request cycle.**

function getArticles(req, res, next) {

getArticles.find('#005', (err, data) => {

res.locals.db\_data = data;

*return* next();

});

}

1. The properties on the “req” object:
   1. Req.body
      1. Data is sent from the body in the **body** of the request
      2. Used mainly in POST or PUT requests, where info might be sensitive or long
      3. Don’t forget to add middleware to parse request body!!
   2. Req.params
      1. Data is sent from the client on the request parameters
      2. Properties are attached to the request route prefixed with a colon.
   3. Req.query
      1. Data is sent from the client on the request route (request endpoint)
      2. Used for searching, filtering, pagination
      3. Written as ?key=value
2. **Express Router**
   1. Modularizing server code
   2. We can use “collections”

|  |  |
| --- | --- |
| Index.js:  app.use(‘/article’, articleRouter)  “use” encompasses get, post, put, etc… Anything that comes into the “article” endpoint, we will send to “articleRouter.”  When “use” takes in the endpoint ‘/article,’…. 🡪 | articleRouter.js  … const router = express.Router()  Router.get(‘/’, ….)  Router.post(‘/’, ….)  module.exports = router;  … ‘article’ kind of gets sliced out and ‘/’ now refers to ‘/article’ in index.js. |

1. **Creating a “REST”ful server**
   1. Endpoint names should be **nouns** representing a namespace or collection
   2. The **request method** determines what functionality we’re performing.
      1. Ie, “Get” /articles /12
2. **MVC**
   1. An architectural pattern for software
   2. MVC stands for model, view, controller
   3. Model
      1. Any data that may be seen/used/processed, like data from a database
   4. View
      1. Rpresents the app’s UI which renders the data from the model in a user-friendly interface.
   5. Controller object
      1. In express, **controllers** are **collections of middleware functions, typically grouped around a specific topic.**

const articlesController = {}

articlesController.getArticles = (req, res, next) => {...}

articlesController.createArticles = (req, res, next => {...})

*// '\*' catches everything. So if we've got a request going to /articles, we process that first*

*// and then we catch anything that didn't get processed by the code above.*

app.use('\*', (req, res) => res.sendStatus(404))

*// global error handler*

app.use((err, req, res, next) => {

res.locals.message(err.message);

const errorStatus = err.status || 500;

*return* res.status(errorStatus).send(res.locals.message)

})

1. **Note about database requests:**
   1. When querying a database, we need to pass the params in {}

articlesController.update(resl.locals.article, { title }, (err, data) => {

})

articlesController.find({ id: 2345 }, () => {...})

1. **Delete request**
   1. We don’t need other parameters in delete.