* **s01 topics**

**e01**

1. MongoDB

* MongoDB
* Installation
  + Installation through Homebrew

brew update

brew install mongodb

* Setup
  + Create a data/db folder where you want your databases to be sourced

mkdir –p users/evanturner/src/data/db

* Run Mongo
  + You connect to the database server, which runs as mongod
  + **mongod** is the primary daemon process for the MongoDB system.
    - It handles data requests, manages data access, and performs background management operations
  + Specify the dbpath when loading mongod

mongod --dbpath ~/src/data/db

**Macintosh HD:Users:evanturner:Desktop:Screen Shot 2014-12-22 at 1.01.36 PM.png**

* In another terminal
  + Run the mongo command

mongo

* + After starting the mongo shell your session will use the test database by default.
  + At any time, issue the following operation at the mongo to report the name of the current database:



* + At this point, if you issue the show dbs operation again, it will not include the book\_library\_db database
  + MongoDB will not permanently create a database until you insert data into that database.

Mongod and Mongos

Mongod

* The **mongod** process is the primary database process that runs on an individual server.
* It handles data requests, manages data access, and performs background management operations
  + You can start MongoDB from a command line by issuing the mongod command and specifying option

Mongo

* mongos provides a coherent MongoDB interface equivalent to a mongod from the perspective of a client
  + The mongo binary provides the administrative shell.
  + mongos for “MongoDB Shard,” is a routing service for MongoDB shard configurations that processes queries from the application layer, and determines the location of this data in the sharded cluster, in order to complete these operations
* mongo is an interactive JavaScript shell interface to MongoDB
  + Provides a powerful interface for systems administrators
  + Provides a way for developers to test queries and operations directly with the database.

Provides a fully functional JavaScript environment for use with a MongoDB

Starting

* By default, MongoDB stores data in the /data/db directory. *On Windows, MongoDB stores data in C:\data\db*. On all platforms, MongoDB listens for connections from clients on port 27017

Specify a Data Directory

* If you want mongod to store data files at a path *other than* /data/db you can specify a dbPath.

At the moment only specifying a dbpath is working:

* mongod --dbpath /users/evanturner/src/mongodb/data/db



* + The dbPath must exist before you start mongod. If it does not exist, create the directory and the permissions so that mongod can read and write data to this path

Stopping

* + In a clean shutdown a mongod completes all pending operations, flushes all data to data files, and closes all data files.
    - Other shutdowns are *unclean* and can compromise the validity the data files
  + Clean shutdowns are executed with the following command:

db.shutdownServer() or Control-C or Command-period

Express.js

var express = require('express');

var app = express();

var bodyParser = require('body-parser');

var parseUrlencoded = bodyParser.urlencoded({ extended: false });

// In memory store for the cities in our application

var cities = {};

app.route('/cities')

.get(function (request, response) {

if(request.query.search) {

response.json(citySearch(request.query.search));

} else {

response.json(cities);

}

})

.post(parseUrlencoded, function (request, response) {

if(request.body.description.length > 4) {

var city = createCity(request.body.name, request.body.description);

response.status(201).json(city);

} else {

response.status(400).json('Invalid City');

}

});

app.route('/cities/:name')

.get(function (request, response) {

var cityInfo = cities[request.cityName];

if(cityInfo) {

response.json(cityInfo);

} else {

response.status(404).json('City not found');

}

})

.delete(function (request, response) {

if(cities[request.cityName]) {

delete cities[request.cityName];

response.sendStatus(200);

} else {

response.sendStatus(404);

}

});

// Adds a new city to the in memory store

function createCity(name, description) {

cities[name] = description;

return name;

}

app.listen(3000);

Node.js

Node works in as Non-Blocking code.

* + It does not do one task at a time.
    - Node will read files and carry out the tasks whenever it is finished while carrying out other tasks
      * Ruby is Blocking code



From the example above, non-blocking code can run two tasks parallel.

* + Javascript deals with this by using callbacks to ensure certain tasks are triggered following the execution of a function

Node operates in a constant event loop checking for events from a request called an Event Queue

* + request
  + connection
  + close

****

* + These events can trigger even more events form a thing called an Event Queue

setTimeout is put on these requests that calls a function or executes a code snippet after a specified delay.



Time line of requests, callback and setTimeouts



This process has no blocking code being that all these functions are being executed together and are only governed by timeouts which we can set to a time we choose.

This is with a Blocking code



In Node.js this is an asynchronous call to read a file



Sending the index.html to inside the server initialize:



Setting the server to a variable for reuse:



Adding event listeners on the server:



Express.js

Express Routers

Express CRUD

MongoDB

Mongoose Schema

Mongoose Models

Postman

Passport

Authentication

Setting User Schema and Bcrypt Password

Controllers

**Express**

**Express Application**

Macintosh HD:Users:evanturner:Desktop:Screen Shot 2015-01-22 at 5.25.09 PM.png

The Express Application is the main component for your web application. Among other things, it is used to define routes, start listening for http connections, and perform routing for requests.

**Express Router**

Macintosh HD:Users:evanturner:Desktop:Screen Shot 2015-01-22 at 5.25.45 PM.png

A router is an isolated instance of middleware and routes. Routers can be thought of as “mini” applications only capable of performing middleware and routing. Every express application has a builtin app router. Routers behave like middleware themselves and can be “.use()’d” by the app or in other routers

**Creating a Route**



Here we are creating a route for ‘/’ to return a JSON object with message set to a constant string.

**Register Routes and Start Server**

****

 Register our previously defined routes with the application using the prefix ‘/api’. This means that all defined routes will be prefixed with ‘/api’.

Calling listen on our port to start listening for incoming connections and requests



**Nodemon**

A utility that will monitor for any changes in your source and automatically restart your server

* $ npm install -g nodemon
  + Instead of using node server.js to run your application, you can use nodemon server.js. It will watch for any changes in your application and automatically restart your server for you

Node Inspector

a debugger interface for Node.js applications that uses the Blink Developer Tools

* $ npm install -g node-inspector

Once it is installed, you can run it using the following command. This will start the debugger and open your browser.

* $ node-debug server.js





**Postman**

A powerful HTTP client to help test web services easily and efficiently. It lets you craft simple as well as complex HTTP requests quickly.

* + It also saves requests for future use so that you never have to repeat your keystrokes ever again. Postman is designed to save you tons of time

Enter our URL in the input field where it says, “Enter request URL here”. Type in ‘http://localhost:3000/api’ and press Send



**Mongoose**

**Storage on Node with MongoDB**

$ npm install mongoose --save



Inside a newly created models folder inside the directory, I have created a beer.js file to define a schema for beer.



* + We loaded the Mongoose package
  + Created a Mongoose schema which maps to a MongoDB collection and defines the shape of the documents within that collection.
  + We defined our schema to contain 2 strings and 1 number.
  + We exported the Mongoose beer model for use within our application.

The last step is to load this new beer model in our server.js file.



**POST or PUT data**

In order to accept POST or PUT data body-parser is required

$ npm install body-parser --save



Finally we need to use this package with our Express Application



**Adding Models**

Creating a new route with the prefix ‘/beers’

Setting up what to do when we POST to that endpoint

* + In this case we create a new Beer model
  + Set its properties to those passed in as data in the POST
  + Then call save on the Beer model which is a Mongoose function that will save the model to the MongoDB database.



Using **Postman** we test out the POST wrote but switch data type to x-www-form-urlencoded

* + - Enter in the url and 3 key value pairs



**GET route for all beers**

We are creating a new route to the prefix ‘/beers’

* + Setting up what to do when we make a GET request to that endpoint. I
  + In this case we use the Mongoose Beer model to call find which will query the MongoDB database and return all our beer



GET request on Postman should show all beer



**Get route for a single beer**

This new route contains the id of the beer we want /api/beers/:beer\_id'

* + With this new route, we then setup what to do when it is called with a GET
  + We end up using the Mongoose Beer model function findById() and pass in the beer\_id parameter to look up the requested beer



Pick out an id from one that you want to request individually.

* + Using **Postman** you can make a request to http://localhost:3000/api/beers/:beer\_id and replace :beer\_id with your id

****

**Updating with PUT**

For each model we remove, we need to update our quantity.

* + This is done by implementing support for the PUT method



Just like we did for getting a single beer, we used the same route but implemented functionality to handle **PUT** requests.

* + We lookup the beer the same way, update its quantity, and then save it back to **MongoDB**.
  + Using the same URL you used to **GET** a single beer,
  + We update **Postman** to use **PUT**, set data type to x-www-form-urlencoded, and add a key value pair quantity set to whatever number you want



**DELETE route**

Use the Mongoose findByIdAndRemove function to find and delete our object



Update **Postman** to use **DELETE** instead of **PUT** and send the request



You should now be able to switch the method to **GET** and receive and error since the object with that id no longer exists

**Passport**

Passport is authentication middleware for Node.js.

* + Passport can be dropped in to any Express-based web application
  + A comprehensive set of strategies support authentication using a username and password, Facebook, Twitter, etc
    - 140+ authentication strategies
    - Single sign-on with OpenID and OAuth
    - Easily handle success and failure
    - Supports persistent sessions
    - Dynamic scope and permissions
    - Pick and choose required strategies
    - Implement custom strategies
    - Does not mount routes in application
    - Lightweight code base

**Adding a Controller** – Beer Controller

Inside the app we make a folder where the models folder resides called controllers and a beer.js





Now server.js need to be refactored



**Creating a User Model**

Like with the beer model, we create a user.js file in the models folder



To properly hash out passwords we need to install bcrypt-nodejs so we do not store password in plain text

* + $ npm install bcrypt-nodejs --save
* The UserSchema we just created has some similarities to our BeerSchema. You will notice that our fields have an object defining their properties such as type, unique, and required.
  + This allows us to better control with is allowed and required in our models.
* We have also added a hook to be called before each call to save() on our User model.
  + This will allow us to check to see if the password has changed. If it has changed, we can then hash it and stored the hash in the model and MongoDB.

**Adding a Controller** – User Controller

With a model to store our user in, we need to make another controller in order to add and view users

* + Something you should consider not doing for applications you create.
    - You don’t want to provide a list of all usernames nor do you want to expose the hashed passwords.
  + In the controllers directory, create a new filed called user.js



With the controller created we need to define our routes so that we can add and view users by making calls to our API

* + In the server.js file, we update the code to require the new controller and include the new routes



Now use **Postman** and make POST and GET calls to <http://localhost:3000/api/users>.

* + For the POST, be sure to include username and password in order to create a new user



**Authentication**

$ npm install passport --save

$ npm install passport-http --save

* + This will install the standard passport package along with passport-http.
  + Passport-http will provide our API HTTP Basic and Digest authentication strategies

Before making and auth controller we need to update our User model to add a function capable of verifying a password in order to authenticate calls to the API



Now we can create our auth controller which will manage authentication for our API endpoints.

* + In the controllers directory, add a file auth.js



* What we have done here, is insert the isAuthenticated function in the callback chain for our endpoint handlers.
  + If a call is made to any of these endpoints without a valid username and password, the request will be denied with a 401 HTTP response.
* Now we can make a call to any of the endpoints we defined that call the isAuthenticated function to test that your authentication is working.
  + Using **Postman**, you can use the Basic Auth tab towards the top to enter your username and password.
    - **Postman** will then automatically create the Authorization header and value for you.







We still need a way to make sure when beer is added, removed, etc that it is done for the authenticated user

* + We need to do is add a field to our Beer model to store the id of the user that owns it. Update your Beer model in the beer.js file.



Now that our model can store a user id, we need to update our Beer endpoint handlers to set the id when adding and query with the id when getting, updating, and deleting. Open up the beer.js file in the controllers directory and update it to the following code





One of the great things about **passport** is that it will automatically set the authenticated user in the req.user object.

* + This allows us to easily get ahold of the user id in order to set it when adding beer to our locker. You can see in the postBeers() function we are setting the userId to req.user.\_id.
    - The other 4 endpoints have had their find(), update(), and remove() functions updated to pass in the userId in order to control which beer we can get, update and delete.

Parts of a URL

Headers

**Protocol**

How a request is transmitted

* + http
  + https
  + file
  + ftp

**Host**

Host identifies the server.

* + localhost
  + numeric IP address
    - Also, subdomains (www)

**Port**

Each server has a collection of numbered ports

* + Special ones like 80 or 443
    - Without a specified port, 80 is assumed for http
    - Without a specified port, 443 is assumed for https

**Path**

Generally the first part of the URL that you app cares about

* + uniquely identify pages

**Querystring**

Optional collection of name/value pairs

* + Starts with a (?) and name/value pairs separated by (&)

**Fragment**

AKA hash, is not passed to the server at all and is used for the browser.

* + Used for AJAX calls or to display a specific part of the document

**Request Headers**

Sent along with requests including information regarding the “user agent”

* + The browser, OS, and hardware.
  + What language the client prefers

**Response Headers**

Information sent back by the server that is usually not rendered

* + - Content Type (HTML, an image, CSS)
* In the console under the Network tab, reload the page and you can see

Form Handling

Whether you let the browser submit the form normally, use AJAX, or employ fancy frontend controls, the underlying mechanism is generally still an HTML form.

* There two options for sending client data to the server
  + query‐ string
    - Normally a GET request
  + the request body.
    - Normally a POST request

<form action="/process" method="POST">

<input type="hidden">

<input type="text" id="fieldColor" name="color">

<button type="submit">Submit</button>

</form>

* + The method is specified explicitly as POST in the <form> tag; if you don’t do this, it defaults to GET
  + The action attribute specifies the URL that will receive the form when it’s posted. If you omit this field, the form will be submitted to the same URL the form was loaded from
  + Always provide a valid action, even if you’re using AJAX (this is to prevent you from losing data)

From the server’s perspective, the important attribute in the <input> fields are the name attributes

* + This is how the server identifies the field.
    - The name attribute is distinct from the id attribute, which should be used for styling and frontend functionality only (it is not passed to the server)
  + The hidden field will not render in the user’s browser. However,
    - Do not use it for secret or sensitive information
  + A form should contain all the fields you would like submitted
    - And none that you don’t
  + Two different actions on a page then use two different forms
* When the user submits the form, the /process URL will be invoked, and the field values will be transmitted to the server in the request body