**Intro to Express**

**Lesson Duration: 60 minutes**

**Learning Objectives**

* Know what Express is
* Be able to use Express to serve up a JSON response

**Intro**

We’ve learned a lot about JavaScript already, but so far we’ve concentrated solely on front-end. Now it’s time to start looking at the the other end of the stack. We’re going to learn how to write a simple web server using a JavaScript library called Express. In this lesson we will write a basic web server which responds to requests with some JSON.

The ability to write a web server opens up a lot of possibilities. We could deploy our apps to a server in the real world, create our own API or add greater back-end functionality; a database layer, for example.

**What is Express?**

Express is an unopinionated, lightweight web framework for Node.js apps. We can use Express to write a web server that will listen for requests and serve our files to clients.

What do we mean when we say that a framework is unopinionated? This means that the framework is flexible and doesn’t force us to adhere to their way of doing things.

Instructor note: hand out start point.

**Start Point**

Take a few moments to look over the start point.

Inside, you’ll see - essentially - *two* software applications - the client, and server. This means two NPM packages, two package.json files, and at least two terminal windows open!

The client is our front-end, which in this case, is our Vue application. There isn’t much in it yet, but we’ll be building it out later in the lesson. Suffice it to say, you won’t be doing anything different to what you did last week!

In the server folder, we’re going to be creating our API; the back-end of our application.

It is quite normal to separate our back-end and front-end like this. In the real world, it’s very possible that you might build one back-end, and many front-ends! In this scenario, your back-end might be the single source of truth, or data, for your app, while you might build front-ends for many different devices or use cases:

* A web app
* Multiple mobile apps
* Alexa skills, or other voice applications

And so on.

**Creating a Web Server using Express**

To begin with, we are going to write a simple server that handles the following HTTP request:

* method: GET
* route: home route (‘/’)
* port: 3000

We are going to respond to that request, firstly, by sending back a simple string to the client. Once we have done that, we will see how to send back JSON, which will generally be much more useful.

**Installing Express**

Firstly, we want to work within the server directory. We also want to start using NPM in our server application. (Remember, this is treated separately from our client application!)

cd server

npm init -y

If we want to create a web server with Express then the first thing that we need to do is install Express.

npm install express

**Creating the Server**

Now that we’ve installed Express, getting a simple web server up and running and handling our request only requires us to write a few lines of code. First, we’ll create a file called server.js, which will contain our server code.

touch server.js

Now that we have a file to work in, we can require Express, so that its functionality is available to us.

**const** express **=** require('express'); *// NEW*

When we require Express, we are given a function. This function returns an object that provides us with various methods that we’ll need to create a server. Let’s invoke express and store the returned object in a variable called app.

**const** express **=** require('express');

**const** app **=** express(); *// NEW*

**Defining Routes**

Now we can use Express to define our routes. It has a set of methods that correspond the the HTTP request methods, including:

1. GET - get
2. POST - post
3. PUT - put
4. DELETE - delete

Each of these methods takes two arguments:

1. The route as a string.
2. A callback containing the behaviour that we would like to execute when we receive a request to this route.

We want to enable a client making a GET request to the home route, to see 'Hello World!' displayed on the page. To do this we are going to use the get method, passing it the following arguments:

1. Route - /
2. A callback that responds with the string 'Hello World!'

**const** express **=** require('express');

**const** app **=** express();

app.**get**('/', **function** () { *// NEW*

});

**Responding to Requests**

This callback should have two parameters: req and res. When a client makes a request to this route, our callback will be passed objects representing the request that we have received from the client and the response that we would like to send back to the client. We can then use these objects to do various things: access the data that was sent with a post request or decide how we would like to respond to the request, for example.

*// ...*

app.**get**('/', **function** (req, res) { *// MODIFIED*

});

The res object has a send method, which can be used to send various things to the client. In this case we will sending the string, 'Hello World!', back to the client.

*// ...*

app.**get**('/', **function** (req, res) {

res.send('Hello World!');

});

Okay, looks like we’re ready to go. Let’s run our server and make a request to it in the browser.

node server.js

*# File executes and completes with no output*

Uh oh! We didn’t tell our server to listen for requests. It just executes our code and then completes. A web server should keep running, waiting for clients to make requests to it so that it can respond appropriately.

app has a listen method which we can use to do this. When we call listen, we need to pass it the port number that we would like to listen for requests on and a callback to execute when the server is running.

All we really need this callback to do is log a string stating that the server is running. The server will then log our message and wait for requests.

*// ...*

app.listen(3000, **function** () { *// NEW*

console.log('App running on port 3000');

});

We have made changes to our server.js so we’ll need to restart our server. We could type node server.js again, but as this is something that we may do a lot it’s probably better to add an npm script. Like npm test, which we saw earlier, npm expects us to define a script called start. It’s common to use this as an alias for starting your server, as this is the script that you will have to run to start your project.

"scripts": {

"start": "node server.js", *// NEW*

"test": "echo \"Error: no test specified\" && exit 1"

},

Now we can start our server by typing npm start.

npm start

*# -> App running on port 3000*

You should now be able to navigate to http://localhost:3000 in the browser and our string should be displayed.

We’ve given ourselves a bit of a problem, however. If we change our code, and refresh the page:

app.**get**('/', **function** (req, res) {

*// CHANGED*

res.send('Hi, World!');

});

Unfortunately, our output doesn’t change. We would have have to restart our server in the terminal, and this gets old, fast.

**Nodemon**

You might be sick and tired of restarting your server by this point. Luckily there’s a handy utility that we can use to watch for modifications and restart our application if our server.js changes. This utility is called Nodemon.

Nodemon is a command-line app that can be installed via npm and used via npm scripts.

npm install --save-dev nodemon

Next we’ll add a script to our package.json, so that we can run our server with Nodemon.

"scripts": {

"start": "node server.js",

"server:dev": "nodemon server.js", *// NEW*

"test": "echo \"Error: no test specified\" && exit 1"

},

Now we can use npm run server:dev to run our server with Nodemon, so that it watches the file for changes and restarts the server as appropriate.

npm run server:dev

*# -> [nodemon] watching: \*.\**

*# -> [nodemon] starting `node server.js`*

*# -> App running on port 3000*

**JSON**

Rather than send a string as a response, let’s send some JSON instead. Our response object has a .json() method we can use here, which we can pass a JavaScript object which will be sent to the client.

app.**get**('/', **function** (req, res) {

*// CHANGED*

res.json({message: "Hello World!"});

});

If we look in the browser now, we should be seeing a JavaScript response. Success! We’ve built our first API!

Let’s finish off our first full-stack JavaScript application by writing a front-end Vue app that consumes this API.

Taking a new terminal window:

cd ../client

npm install

npm run serve

We’re going to try to grab a response from our API. Let’s see what happens.

*// App.vue*

mounted(){

fetch("http://localhost:3000/")

.then(response **=>** response.json())

.then(data **=>** console.log(data));

},

What do you see if you open your terminal?

Access to fetch at ‘http://localhost:3000/’ from origin ‘http://localhost:8080’ has been blocked by CORS policy: No ‘Access-Control-Allow-Origin’ header is present on the requested resource. If an opaque response serves your needs, set the request’s mode to ‘no-cors’ to fetch the resource with CORS disabled.

This is known as a CORS error. To understand this error, we first have to understand the [same origin policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy).

The same-origin policy is a security feature that prevents scripts running on different domains from talking to each other. Since our API is running on one port (3000), and our front-end application is running on another, (probably 8080), the request is blocked.

If we have control over the API (which in this case is our server application), we can relax this policy. We can state which domains and ports should be allowed to use our API, and which routes and HTTP methods should be affected.

For simplicity’s sake, we’re going to allow our API to be accessible anywhere.

Let’s navigate back to our server directory, and install Express’ [CORS](https://expressjs.com/en/resources/middleware/cors.html) package.

cd ../server

npm install --save-dev cors

Now that we’ve done this, we can configure our Express server to use CORS.

**const** express **=** require("express");

**const** app **=** express();

*// ADDED*

**const** cors **=** require("cors");

app.use(cors());

We should now be able to write our front-end application.

Task - 5 minutes - display the message sent by the server.

Solution:

**<**template**>**

**<**div id**=**"app"**>**

**<**h1**>**My App**<**/h1>

**<**p v**-if=**"serverMessage"**>**The server said: {{ serverMessage }}**<**/p>

**<**/div>

**<**/template>

**<**script**>**

**export** **default** {

name: 'app',

mounted(){

fetch("http://localhost:3000/")

.then(response **=>** response.json())

.then(data **=>** **this**.serverMessage **=** data.message);

},

data () {

**return** {

serverMessage: ""

}

}

}

**<**/script>

**Recap**

What is Express?

Answer

When creating a server that responds to a request, what three pieces of information will you need?

Answer

**Conclusion**

We just wrote our first web server using JavaScript. If we wanted to, we would now be able to deploy our apps in the real world.

Next we’ll look at how we can create our own RESTful JSON API, just like the ones that we’ve been using to gather data. Later we’ll add a database layer, allowing us to persist and modify the data provided by our API

**RESTful API in Express**

**Lesson Duration: 120 minutes**

**Learning Objectives**

* Be able to create a RESTful JSON API with multiple resources
* Be able to use middleware to handle post request bodies in a RESTful JSON API
* Be able to use Express Router to implement modular routers

**Introduction**

A RESTful API defines a set of methods (GET, POST, PUT and DELETE) which developers can use to make requests and receive responses using HTTP protocol. We have been consuming a number of JSON API’s that other developers have built. In this lesson we are going to build our own JSON API using an Express server. We will be defining a set of endpoints that handle requests and respond with JSON data, using API design principles to ensure that our service is easy and intuitive for other developers to use.

**RESTful API**

When creating an API we want to design it in a way that ensures it is intuitive for other developers to use. To do this we are going to adhere to the following principles:

1. Use the HTTP verbs (GET, POST, PUT and DELETE) appropriately. GET is considered a safe method and shouldn’t modify any data.
2. Create endpoints that use the hierarchical nature of the URL to represent the structure of the request. For example, ‘https://codeclan.com/students/5’ should indicate a request regarding one student from the collection ‘students’ with id of ‘5’. Collections should be named with the pluralised resource name.

Our API will have two resources, teas and biscuits.

**Teas and Biscuits Application**

The application has an Express server. Run the start code by running the server:

cd server

npm install

npm run server:dev

Visit http://localhost:3000/ in the browser.

**Directory Structure**

In addition to the ‘client’ directory, the application also has a ‘server’ directory, which contains server.js. server.js is responsible for configuring the server.

**Teas and Biscuits API**

The front-end application in the client folder makes two initial requests to the following URLs and displays a list for each set of data it gets back:

1. http://localhost:3000/api/teas
2. http://localhost:3000/api/biscuits

In this lesson we are going to be working entirely server-side to create a RESTful API for this front-end to consume.

**Creating a RESTful API using an Express Server**

We want to create a RESTful API that serves up two resources: teas and biscuits. We are going to start by creating the teas resource, adhering to API design principles to create the following endpoints with the corresponding actions:

* /api/teas - Index (GET)
* /api/teas/:id - Show (GET)
* /api/teas - Create (POST)
* /api/teas/:id - Destroy (DELETE)
* /api/teas/:id - Update (PUT)

We are using /api in the routes to distinguish it from our front-end application. This way, if we wanted a second URL, http://localhost:3000/teas, that served HTML, it wouldn’t conflict with the API.

Each of these routes will respond with JSON data that our front-end application will consume.

**Index**

The start code has an array of teas defined in server.js. Later in the module we will be building API’s with data persisted to a database, but as the focus of this lesson is serving up the JSON data on a set of RESTful routes, we are using hardcoded data.

Let’s start by handling a GET request to the index (/). We will use Express’s get method, passing it the path and a callback that will invoked when a request hits this route, and is passed a request (req) and response (res) object. We will then send back the teas array as JSON using the response object’s json method.

*// server.js*

**const** teas **=** [

*//...*

];

app.**get**('/api/teas', (req, res) **=>** { *// NEW*

res.json(teas);

});

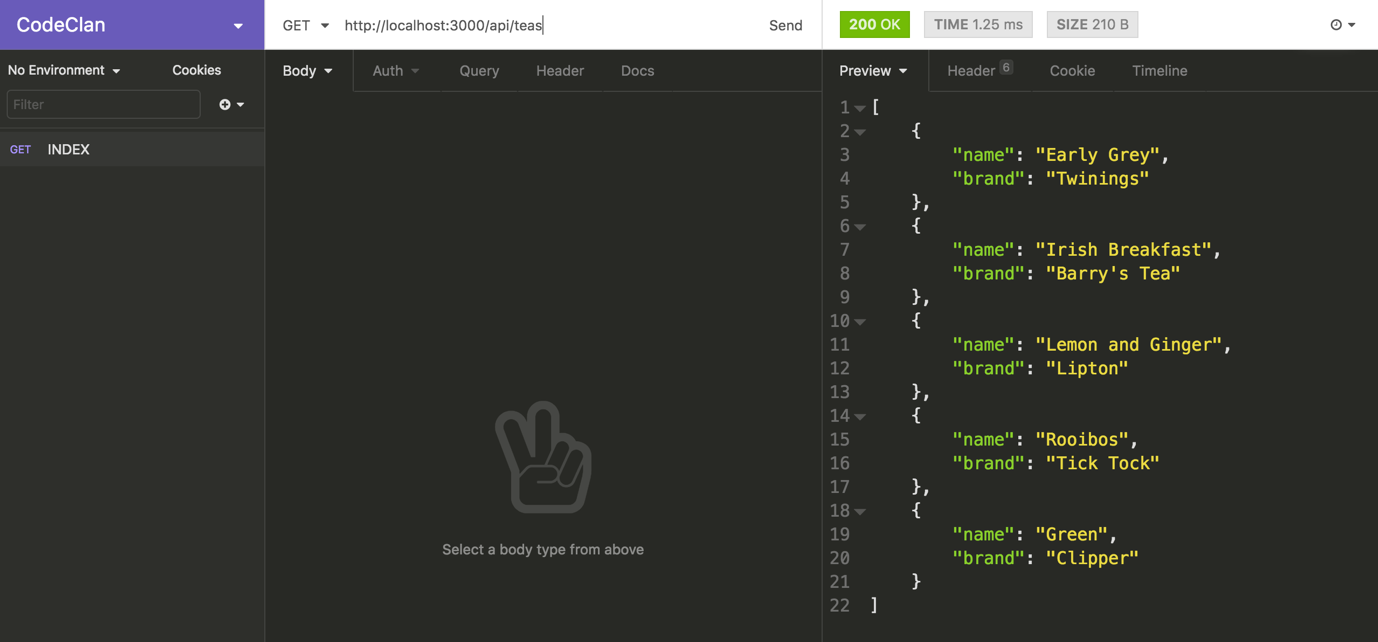
Now when we visit http://localhost:3000/api/teas in the browser, we see the JSON teas data displaying. We have just made the first endpoint of our API.

**Insomnia**

When we are building an API, we want a way of testing the endpoints as we create them. We can test the GET methods (index and show) in the browser, but we need a way of testing the other HTTP methods such as POST (create) and PUT (update) without having to build a front-end with a form to send the data. [Insomnia REST Client](https://insomnia.rest/) is a HTTP client that provides us with an convenient way to test our server with all the HTTP methods without requiring a browser front-end.

We are going to use Insomnia REST Client to the index route we have just created.

1. Open Insomnia REST Client.
2. Click the + symbol in the left menu (or use cmd + N) to create a new request.
3. Enter a name (‘Index’) and select the method (‘GET’) for the request. Click ‘Create’.
4. Add the URL (http://localhost:3000/api/teas) in the address bar at the top.
5. Click ‘Send’ to see the response data display in the right-pane.



*GET request to ‘http://localhost:3000/api/teas’ displaying the response using Insomnia REST Client*

**Show**

Let’s now implement the show action, where we respond to the request with one object. The request URL will contain a parameter, which we use to identify which object is being requested. For example, ‘http://localhost:3000/api/teas/1’, where ‘1’ relates to the id of the tea being requested.

Again, it will be a GET request, so we will use the get method, passing it the path. This time the path will include a parameter (indicated by the colon proceeding it). We will call the parameter id. Often we would be working with ids assigned by a database as we can ensure these are unique, but in this case we will use the index position of the tea object in the array.

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

res.json(teas);

});

app.**get**('/api/teas/:id', (req, res) **=>** { *// NEW*

});

On the request object, we can access the params (short for parameters) object. This allows us to use the parameter id to get back the value sent as part of the url. We can use that number to select the tea with that index position from the array and send it back on the response as JSON.

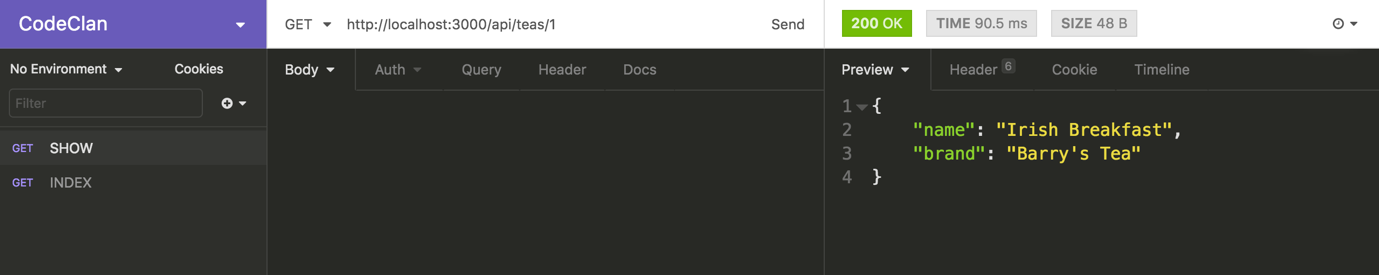
app.**get**('/api/teas/:id', (req, res) **=>** {

res.json(teas[req.params.id]); *// MODIFIED*

});

Let’s test our show route in Insomnia:

1. Click the + symbol in the left menu (or use cmd + N) to create a new request.
2. Enter a name (‘Show’) and select the method (‘GET’) for the request. Click ‘Create’.
3. Add the URL (http://localhost:3000/api/teas/1) in the address bar at the top.
4. Click ‘Send’ to see the response data display in the right-pane.



*GET request to ‘http://localhost:3000/api/teas/1’ displaying the tea object at position 1 in the array using Insomnia REST Client*

**Create**

To be able to handle a POST request, we have to learn about the request’s body object. When the client makes a POST request it can send data with the body of the request (for example, the values of a form when it is submitted), which we can then retrieve server-side.

In our case we want the client to be able to send a tea object with the request, and for our POST route to then add that object to the teas array. Our POST route is going to expect to receive an object with a name and brand.

{

"name": "One Cup",

"brand": "Tetley"

}

We will use Express’s post method, pass it the appropriate path, ‘/api/teas’ and a callback. To access the data sent from the client with the request, we access the request’s body object.

app.**get**('/api/teas/:id', (req, res) **=>** {

res.json(teas[req.params.id]);

});

app.post('/api/teas', (req, res) **=>** { *// NEW*

console.log(`req.body`, req.body);

});

Test our create route in insomnia:

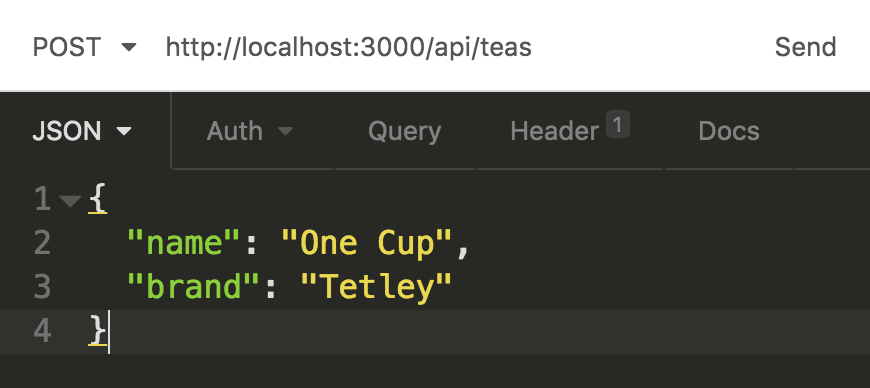
1. Click the + symbol in the left menu (or use cmd + N) to create a new request.
2. Enter a name (‘Create’) and select the method (‘POST’) for the request. Click ‘Create’.
3. Add the URL (http://localhost:3000/api/teas) in the address bar at the top.
4. This time we need to add the data we want to send with the request. Click ‘Body’, select ‘JSON’ and the add the following JSON to the left-hand pane.

{

"name": "One Cup",

"brand": "Tetley"

}



*Adding the JSON body to a POST request in Insomnia Rest Client*

Note: The keys must be enclosed with inverted commas because this is JSON (not JavaScript)

When we click send and look in the Terminal window where the server is running, we see req.body is undefined -

req.body undefined

To fix this error we need to use some Express middleware.

**body-parser**

We can see that the body, which we are trying to access on the request object, is undefined. Express is a light-weight framework and it doesn’t expose the body object with its built-in functionality. To access the request’s body we have to use some middleware called bodyParser. The role of body-parser is to extract the body from the POST request and make it accessible on req.body.

body-parser is an npm package so first we need to install it.

npm install body-parser

Then we require it and tell the server to use it.

**const** bodyParser **=** require('body-parser'); *// NEW*

app.use(bodyParser.json()); *// NEW*

bodyParser.json() provides the middleware that parses json, so our API will only parse the body of a request which has a header with ‘Content-Type’ specified as JSON. If you click on the ‘Header’ tab of the left hand section of Insomnia, you will see that there has been a header set with ‘Content-Type’ set to ‘application/json’. Insonmia does this for us when we add a JSON body to the request.

Now when we test our create route, Insomnia Rest Client will hang because we haven’t told it what to do in response to the request, but if you look in the terminal window where the server is running, you will see the data outputted: { name: 'One Cup', brand: 'Tetley' }.

To complete the request, let’s push the posted data into the array of teas and send back the updated array.

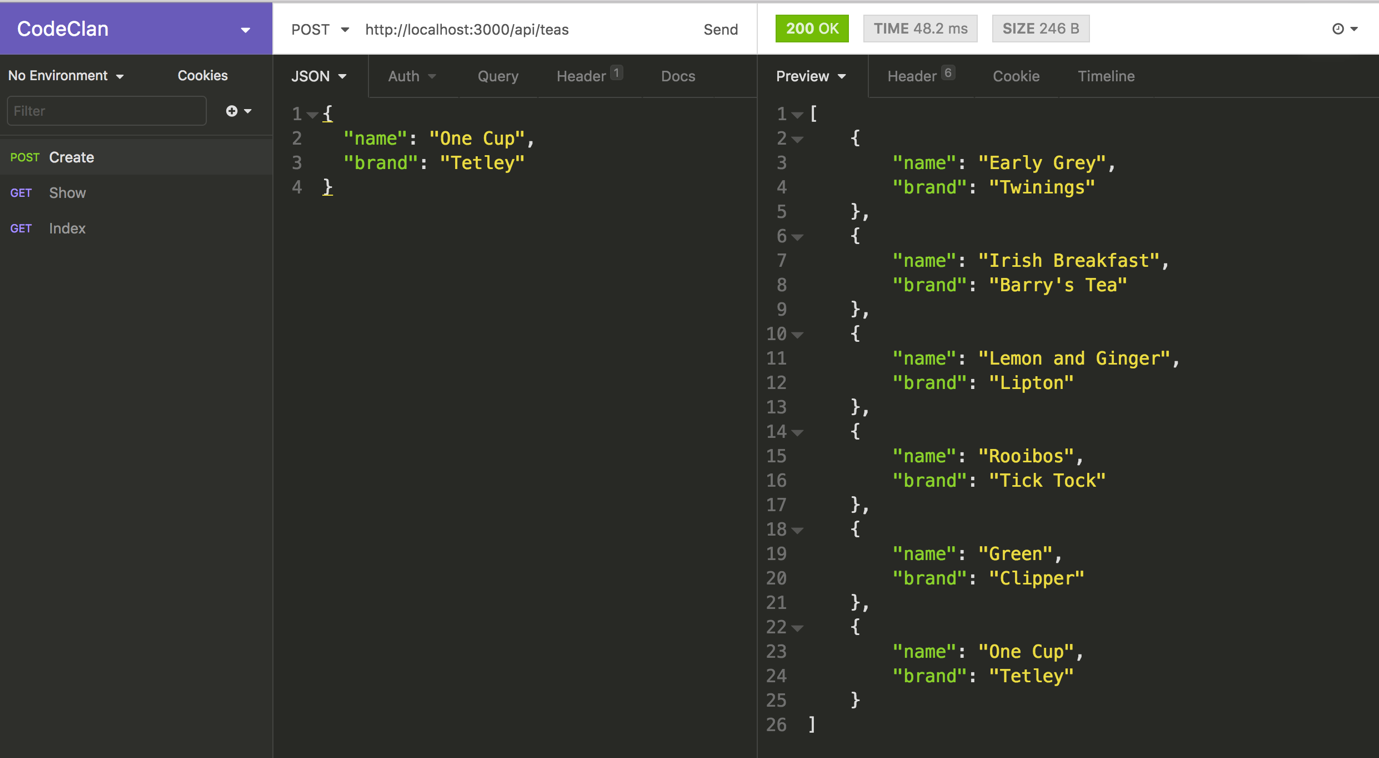
app.post('/api/teas/', (req, res) **=>** {

teas.push(req.body);

res.json(teas);

});

Now when we test the POST request in Insomnia REST Client, you should get a response of the teas data, with the new tea added.



*POST request to ‘http://localhost:3000/api/teas’ displaying the response using Insomnia REST Client*

Our create route is now complete.

**Task (15 minutes)**

Implement the following routes where the parameter id refers to tea’s index position in the array:

1. Destroy:
   * handle a delete request made to /api/teas/:id
   * delete the appropriate tea object in the array
   * send back all the teas data as JSON
2. Update:
   * handle a put request made to /api/teas/:id
   * update the appropriate tea object in the array with the new tea object sent on the request’s body object
   * send back all the teas data as JSON

Test each of the routes with Insomnia

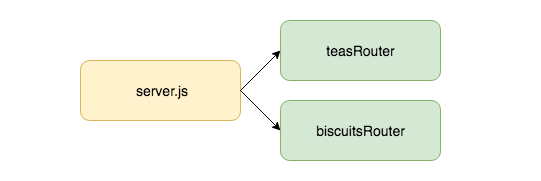
Example solution

We now have the CRUD (Create, Read, Update and Delete) operations for our teas resource.

**Express Router**

If we now want to add the second resource (biscuits) with all the corresponding routes, our server.js is going to get pretty unmanageable and not very DRY. As additional resources get added, the file will get increasingly difficult to maintain. Express provides a Router object which allows us to modularise our sets of routes, specifying a particular path (‘/api/teas’ and ‘/api/biscuits’) for each router.

We are going to refactor our back-end to implement modular routers and add the additional biscuits resource. The server will delegate the routing for each resource (teas and biscuits) to its own router. This is what our back-end architecture is going to look like:



*Server-side architecture with modular routers*

We are going to create a function that:

* takes in the teas array
* creates a Router
* defines a set of routes on the Router
* returns the Router

This function is going to be stored in a helper file called create\_router.js. Let’s create a helpers directory to store it in and then create the file.

mkdir server/helpers

touch server/helpers/create\_router.js

**createRouter Function**

createRouter will be a function that takes in the teas array (data). It is the function that we will export from the file.

*// create\_router.js*

**const** createRouter **=** **function** (data) {

};

module.exports **=** createRouter;

createRouter will create and export an Express Router. We will require Express and inside the function invoke express.Router which returns a Router object.

**const** express **=** require('express'); *// NEW*

**const** createRouter **=** **function** (data) {

**const** router **=** express.Router(); *// NEW*

};

Let’s now move the routes handling the teas resource requests, from server.js into create\_router.js. (Delete it from server.js and add it into create\_router.js).

*// create\_router.js*

**const** express **=** require('express');

**const** createRouter **=** **function** (data) {

**const** router **=** express.Router();

app.**get**('/api/teas/', (req, res) **=>** { *// NEW*

res.json(teas);

});

app.**get**('/api/teas/:id', (req, res) **=>** { *// NEW*

res.json(teas[req.params.id]);

});

app.post('/api/teas/', (req, res) **=>** { *// NEW*

teas.push(req.body);

res.json(teas);

});

app.put('/api/teas/:id', (req, res) **=>** { *// NEW*

teas[req.params.id] **=** req.body;

res.json(teas);

});

app.**delete**('/api/teas/:id', (req, res) **=>** { *// NEW*

teas.splice(req.params.id, 1);

res.json(teas);

});

};

module.exports **=** createRouter;

We need to make a couple of changes the routes we brought in:

1. We need to call the methods on the router object, router, rather than app
2. We need to change reference to teas to data
3. We need to remove /api/teas from the paths, as this will be specified by server.js

**const** express **=** require('express');

**const** createRouter **=** **function** (data) {

**const** router **=** express.Router();

router.**get**('/', (req, res) **=>** { *// MODIFIED*

res.json(data); *// MODIFIED*

});

router.**get**('/:id', (req, res) **=>** { *// MODIFIED*

res.json(data[req.params.id]); *// MODIFIED*

});

router.post('/', (req, res) **=>** { *// MODIFIED*

data.push(req.body); *// MODIFIED*

res.json(data); *// MODIFIED*

});

router.put('/:id', (req, res) **=>** { *// MODIFIED*

data[req.params.id] **=** req.body; *// MODIFIED*

res.json(data);

});

router.**delete**('/:id', (req, res) **=>** { *// MODIFIED*

data.splice(req.params.id, 1); *// MODIFIED*

res.json(data); *// MODIFIED*

});

**return** router;

};

module.exports **=** createRouter;

That is our teas router complete.

**Server**

server.js is going to create a teasRouter and delegate the routing to it for our teas resource. Let’s require create\_router.js in server.js and create the teasRouter.

*// server.js*

*// ...*

**const** bodyParser **=** require('body-parser');

**const** createRouter **=** require('./helpers/create\_router.js'); *// NEW*

app.use(bodyParser.json());

**const** teasRouter **=** createRouter(teas); *//NEW*

*// ...*

Now we need to tell the server to use the teasRouter by using the use method. The use method takes two arguments:

1. the path that we want it to use the teas router on, (this is why we removed /api/teas from the teas router routes), meaning we only have to state it here in server.js once.
2. The router object we want it to use

*// ...*

**const** teasRouter **=** createRouter(teas);

app.use('/api/teas', teasRouter); *//NEW*

*// ...*

Now we are set up nicely for when we want to add additional resources, as we will be able to create additional modular routers for each resource and require and add them here.

Great, now you can test your API in Insomnia Rest Client again to ensure it is still serving the JSON on the endpoints and visit http://localhost:8080/ to ensure the index.html is still being served up on the home route.

**Task (10 minutes)**

Create a biscuits resource with the following endpoints and corresponding actions:

* /api/biscuits - Index (GET)
* /api/biscuits/:id - Show (GET)
* /api/biscuits - Create (POST)
* /api/biscuits/:id - Destroy (DELETE)
* /api/biscuits/:id - Update (PUT)

The resource should serve up the following seed data:

**const** biscuits **=** [

{ name: "Digestives", brand: "McVitie's" },

{ name: "Hobnobs", brand: "McVitie's" },

{ name: "Shortbreads", brand: "Walkers" },

{ name: "Jammy Dodgers", brand: "Burton's" },

{ name: "Custard Creams", brand: "Crawford's" }

];

Example solution

Because the front-end is set up to consume the API on these routes, if you visit ‘http://localhost:8080/’ you are now able to use the form to submit a tea or a biscuit and see it render to the screen.

**Task (15 minutes)**

Take note of how the front-end application handles the submission of the form when a new tea is added, from the point the form is submitted, to the updated list being rendered on the screen.

Example solution

**Recap**

What design principles should we consider when creating and API? And why?

Answer

What is the function of the the Express middleware, body-parser?

Answer

What problem does the Express Router object solve?

Answer

**Conclusion**

Creating RESTful APIs allows us to create a back-end service that our front-end application can consume. Adhering to RESTful API design principles we can ensure our APIs are easy and intuitive for other developers to use.

Because Express is a light-weight framework, we have to use the additional middleware, body-parse, to handle POST requests and retrieve the data sent on the body of the request.

Using Express Router we can create modularised routes for each of the API’s resources, creating a server-side application which is maintainable and extensible.

# Intro To MongoDB

**Lesson Duration: 45 minutes**

## Learning Objectives

* Understand what a non-relational database is
* Be able to create a MongoDB database
* Understand what background processes are
* Be able to perform CRUD operations on a MongoDB database
* Know what MongoDB’s ObjectId is used for

## Introduction

## Relational Verses Non-relational

Databases allow us to organise and store collections of data. Relational databases (such as MySQL and PostgreSQL), have a schema that define tables with columns and rows. Relationships between the tables are created with the use of primary and foreign keys.

There are cases where our data does not fit well into a relational model, for example, if our collection contains objects with different properties, or the same property has values of different datatypes. In these cases a non-relational database might be more appropriate. Non-relational databases are also referred to as ‘NoSQL’ due to the fact they do not require SQL for queries.

We are going to be using an open-source, non-relational database called MongoDB. MongoDB takes its name from the word ‘humongous’ and it was designed for storing huge amounts of non-relational data. We will be using JavaScript to query it and as it stores data in collections of JSON-style, key-value pair documents, it is ideal for storing JavaScript objects.

## Non-relational Data

Let’s take a look at an example of non-relational data. If a shop stocks products, even within the specific category ‘kitchenware’, each product might have an entirely different set of properties.

**const** kitchenware **=** [

{

name: 'frying pan',

diameter: 28

},

{

name: 'chopping board',

size:

{

width: 20,

length: 18

},

material: 'wood'

},

{

name: 'whisk',

handleColour: 'red'

}

]

There is no obvious way of organising this data into separate tables, so if we were to try and store this data in a relational database, we would need a column for each property (name, diameter, size, material, handle colour). As each product would only have values for some of the columns, there would be a lot of nulls in the database.

## MongoDB

MongoDB is a document-oriented database. Instead of having tables with columns and rows, it stores data in collections of JSON documents.

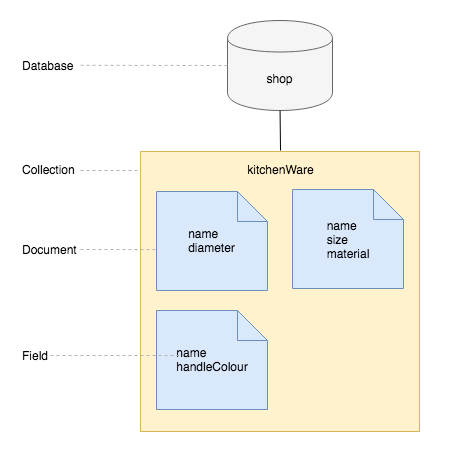
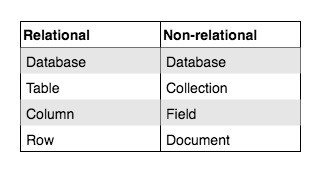


Diagram of non-relational database

We can see from this diagram that with a non-relational database, we don’t have the problem of every kitchenware product having a handleColour property, as each object is stored as a document and there are no tables or columns. Instead it has collections of documents and each document has the properties that it requires.



Terminology to describe the component parts of relational and non-relational databases

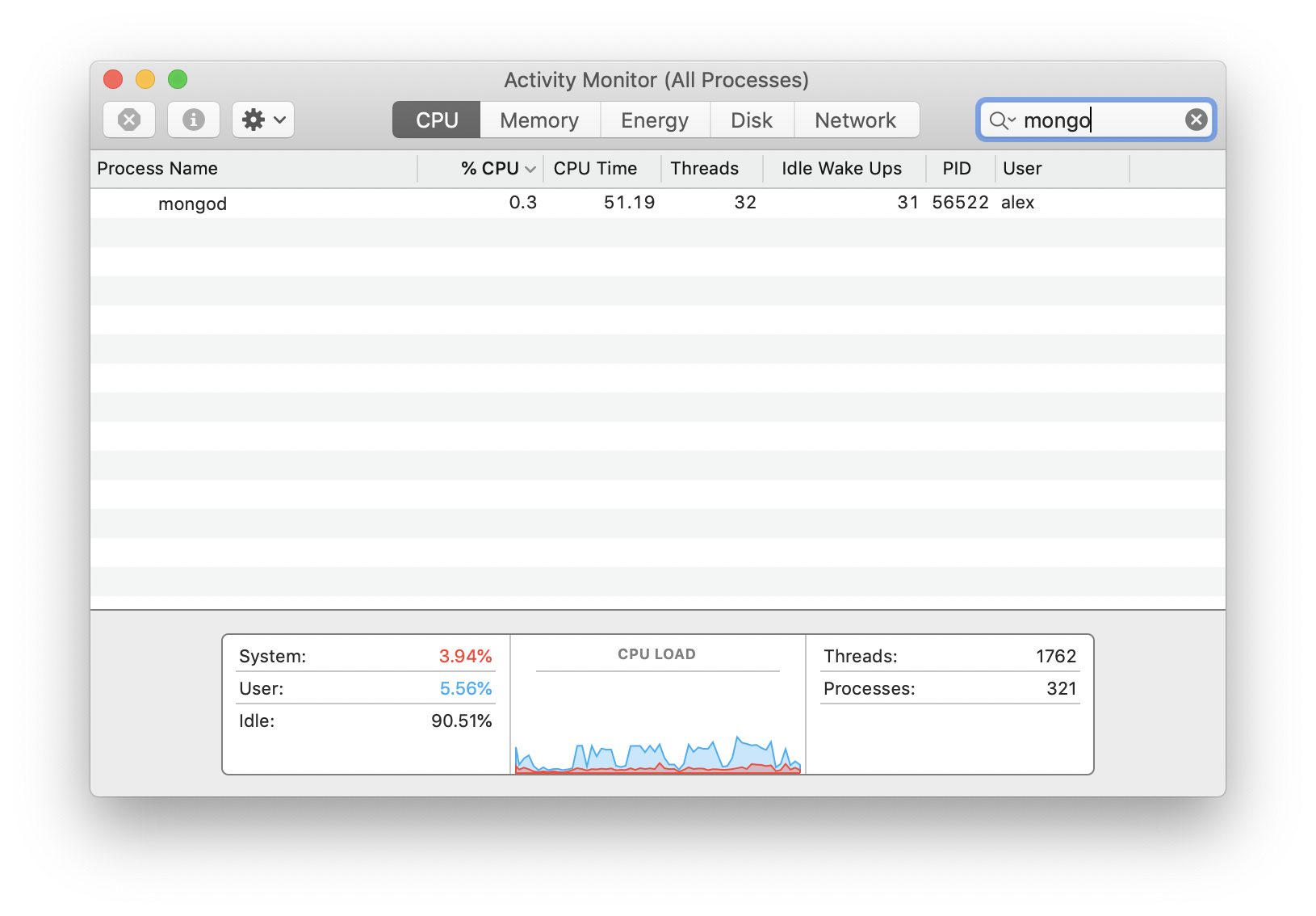
## Background Processes

In order to connect to and carry out CRUD operations on databases we need to have a server running on our Macs. Luckily the laptop script we ran before the start of the course set this up for us. It runs all the time in the background listening for connections. Much like a web server listens for HTTP requests.

Background processes do not have an icon that appears in our dock and we cannot switch to them like a regular application. To see them we can look in Activity Monitor.

1. Use Spotlight (command-space) to search for Activity Monitor
2. In the View menu make sure All Processes is ticked
3. Use the Search bar to look for mongod

We can also examine the resources a process is using with the tabs (CPU, Memory…)



If for some reason the mongod process froze we could force it to terminate here.

Sometimes we might need to look the logs this MongoDB server produces in order to debug errors. The following command, run in the Terminal will do this for us:

tail -f /usr/local/var/log/mongodb/mongo.log

*# ...*

*# - > 2019-10-30T15:36:22.232+0000 I NETWORK [initandlisten] waiting for connections on port 27017*

*# ...*

It shows us the end of the log (tail) and updates as new entries are added (-f).

## MongoDB from the Command-line

We will be using JavaScript to talk to MongoDB, so there is no need to learn additional querying languages. We are going to create a database with the following structure:

* Database name: ‘shop’
* Collection name: ‘kitchenware’
* Document: A product with the following properties:
  + name: knife
  + lengths: [15, 20, 25]

Often we will be interacting with MongoDB from our JavaScript applications, but we can also interact with it from the command line. In this lesson we will start by looking at the create and read operations from the command-line, before looking at MongoDB’s GUI and running the commands from a file.

### Create and Connect to a Database

Start by opening a new terminal window (the mongod server should still be running in a separate window) and enter the mongo shell.

mongo

We will use the mongo shell command use, followed by the name of the database, to connect to the database if it exists. If it doesn’t, this command will create the database with the specified name and connect to it.

use shop

We are now connected to the database, ‘shop’. We have access to a db object, which has all the methods that we need to create collections and to create, read, update and delete documents.

We are going to see how to perform the CRUD operations on our MongoDB ‘shop’ database.

### Creating a Collection and Inserting a Document

We currently don’t have any collections in the shop database. We can create a collection and insert a document into it with one command.

We will create a kitchenware collection and insert a document that represents a product (a knife) into it. We use the database object, db, access the collection name, kitchenware, and call insertOne, passing it the JSON object we want to insert. The [insertOne](https://docs.mongodb.com/manual/reference/method/db.collection.insertOne/#db.collection.insertOne) method allows us to insert one document into a collection. If the collection doesn’t exist yet, the insertOne method will first create it, then insert the document into it.

db.kitchenware.insertOne**({** name: "knife", lengths: **[**15, 20, 25] **})**;

Note: Because we are inserting a JSON document, we are able to insert an object with nested collections.

You’ll notice the following output:

{

"acknowledged" : **true**,

"insertedId" : ObjectId("5bab51cc6467abe6638d6ea5")

}

This is telling us that MongoDB has successfully inserted the document into the collection, and that it has given the document an id of ‘5bab51cc6467abe6638d6ea5’. This is the document’s unique identifier.

Let’s insert another.

db.kitchenware.insertOne**({** name: "pan", type: 'frying' **})**;

### Reading from the Database

Let’s now retrieve all the documents in the kitchenware collection using the find method. Again, we use the database object, db, access the collection, kitchenware, and call the method on the collection.

db.kitchenware.find**()**;

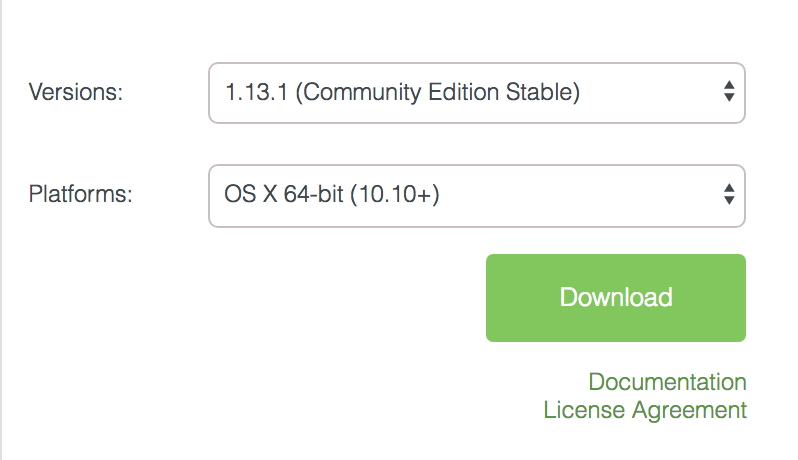
*# -> { "\_id" : ObjectId("5bab51cc6467abe6638d6ea5"), "name" : "knife", "lengths" : [ 15, 20, 25 ] }*

*# -> { "\_id" : ObjectId("5bab51d36467abe6638d6ea6"), "name" : "pan", "type" : "frying" }*

When we call find on the collection, passing it no arguments, it gives us back all the documents in the collection.

## MongoDB Compass GUI

We can use MongoDB’s GUI to more easily view the state of our database. You can download the free community edition from <https://www.mongodb.com/download-center/compass>



Download settings for MongoDB Compass

Once you have downloaded MongoDB Compass, open it and click “Connect”. Down the left-hand side you will see the list of databases. If you click on a database you will see any collections inside it, and inside the collection, any documents and their fields.

## MongoDB From a File

### Seeding a MongoDB

We can also define our database and seed it with data in a JavaScript file and run it using the mongo shell. Let’s do that now.

touch seeds.js

We will start by using MongoDB’s shell commands to create and switch to a database called ‘zoo’.

*// seeds.js*

use zoo;

And as this is a seeds file, we will drop the database so it is cleared each time. Just like when we were working from the command-line, now we are connected to the zoo database, we have the db object available to us.

use zoo;

db.dropDatabase(); *// NEW*

### CREATE - insertOne

Previously we used insertOne to add an entry to our ‘kitchenware’ collection in our ‘shop’ database. MongoDB also has an insertMany method that we call on our collection. We can pass insertMany an array of objects and it will insert all the objects into the database. Like insertOne, if there is no animals collection, insertMany will create the collection and insert into it.

use zoo;

db.dropDatabase();

db.animals.insertMany([ *// NEW*

{

name: "Janet",

type: "Polar Bear"

},

{

name: "Norman",

type: "Penguin",

age: 5

}

]);

Note: Because MongoDB is a non-relational database, we can insert entries with different fields without any problems.

To run the file we use the following command:

mongo < seeds.js

If you refresh MongoDB Compass’s database list (using the button at the top of the list of databases), you will see a database called zoo with two animals inside. We have now seeded our database.

### Querying a MongoDB

Next we are going to see how we can query our database. We are going to create a file to use to practice doing this.

touch mongodb\_play.js

Again, we need to specify which database we want to use.

*// mongodb\_play.js*

use zoo;

### READ

We have seen that we can use find, passing it no arguments, to get back all the entries in the collection. If we only want to get one entry back, we can use findOne. A common use case for this is when we are querying by a unique identifier, such as an ID.

To be able to find by id, we first need to learn about MongoDb’s ObjectID.

#### MongoDB ObjectId

When we insert an entry into a database, MongoDB assigns it an unique identifier, an [ObjectId](https://docs.mongodb.com/manual/reference/method/ObjectId/), with the key \_id. You can see these by looking at the entries with MongoDB Compass.

To query the database by ID, we need to create an instance of MongoDB’s ObjectId by using the constructor and passing it an existing ID as a string.

*// ...*

**const** id **=** ObjectId('5af17fe430e043c3e62149b8');

Note: Replace the above string ID with an ID of an entry currently in your database. You can access an existing ID from MongoDB Compass.

We pass findOne an object with the key \_id, with a value of our instance of the ObjectId. This will return the first entry it finds with the specified ObjectId. Because we know the IDs are unique, we know the first will be the only one that will match.

**const** id **=** ObjectId('5af17fe430e043c3e62149b8');

db.animals.findOne({ \_id: id }); *// NEW*

Run the file to see the found entry in the terminal window.

mongo < mongodb\_play.js

### UPDATE

We can update the fields of existing entries using the updateMany method if we want to update multiple entries. When we want to update one entry, we can use the updateOne method. We are going to update an entry with a specific ID, so we will use the updateOne method, calling it on the collection.

*// ...*

db.animals.updateOne();

We pass updateOne two arguments:

1. The query object for specifying which entry we want to update. Again, we will use the ID of the existing entry we want to update, which you can access from MongoDB Compass.
2. An object with the key $set. The value should be an object containing the field names and the new values. We will update the name of the animal to be ‘Pip’.

*// ...*

db.animals.updateOne( *// MODIFIED*

{ \_id: ObjectId('5af17fe430e043c3e62149b8') },

{ $set: { name: "Pip" } }

);

Note: Replace the above string ID with an ID of an entry currently in your database.

The first argument is used to identify the entry to update, the second argument is telling MongoDB to ‘set’ any fields in the existing entry that match a property in the object we pass it, to have the new value. I.e, for the specified entry, if the entry has a ‘name’ field, update its value to be ‘Pip’. (If the entry doesn’t have a field name, it will create one.)

When we run the file again, refresh MongoDB Compass’s entries list (using the button on the right side, just above the list of entries) we see that the name has been updated.

### DELETE

Lastly, if we want to delete entries from the database we can use the deleteMany method and pass it a query object. In the case we want to delete one entry we can use the deleteOne method that also takes a query object. Again, we call it on the collection, and it will delete the first document that matches the query (in our case, the first document with a \_id field with the value of ‘5af17fe430e043c3e62149b8’).

*// ...*

db.animals.deleteOne({ \_id: ObjectId('5af17fe430e043c3e62149b8') });

Note: Replace the above string ID with an ID of an entry currently in your database.

Now if you run the file again, refresh MongoDB Compass’s entries list (using the button on the right side, just above the list of entries) you will see that the entry you specified has been removed.

## Recap

What makes a database non-relational?

Answer

</br>

What command did we use to create and connect to a MongoDB database?

Answer

</br>

Once connected to a MongoDB database, how do we reference it when we want to interact with it?

Answer

</br>

What type of object does MongoDB use for entry IDs? And what key does it use to store them?

Answer

## Conclusion

Databases allow us to organise collections of data and access data efficiently. Non-relational data is data that doesn’t fit easily into a relational schema (tables with columns and rows). If your data model is non-relational, you may decide to use a non-relational database, such as MongoDB, which stores data as JSON documents in collections.

MongoDB gives each entry an ObjectId as a unique identifier. This is stored on the key \_id. We can create instances of ObjectIds to query our database, accessing entries by their ID.

## Further Resources

[A full list of MongoDB collection methods](https://docs.mongodb.com/manual/reference/method/js-collection/)

**Homework: Full Stack Games Hub App**

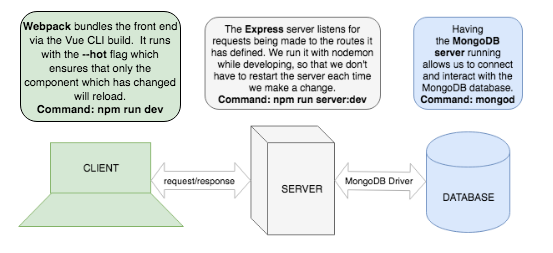
**Learning Objectives**

* Understand the relationship between client, server and database
* Be able to navigate a codebase that you haven’t written

**Brief**

Your boss has asked to you look over the codebase of a full-stack JavaScript application. The front-end is written in JavaScript using Vue, the back-end uses an Express server and a MongoDB database. Your task is to make yourself familiar with the codebase.

The application includes a README.md with instructions on running the application.



*Overview of the tech stack and tooling with commands*

**MVP**

**Task**

Draw a diagram showing the dataflow through the application starting with a form submission, ending with the re-rendering of the page. This will involve a multi-direction data-flow with the client posting data to the server and the server sending data back to the client with the response. Detail the client, server and database in the diagram and include the names of the files involved in the process.

**Questions**

1. What is responsible for defining the routes of the games resource?
2. What do you notice about the folder structure? Whats the client responsible for? Whats the server responsible for?
3. What are the the responsibilities of server.js?
4. What are the responsibilities of the gamesRouter?
5. What process does the the client (front-end) use to communicate with the server?
6. What optional second argument does the fetch method take? And what is it used for in this application? Hint: See [Using Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch) on the MDN docs
7. Which of the games API routes does the front-end application consume (i.e. make requests to)?
8. What are we using the [MongoDB Driver](http://mongodb.github.io/node-mongodb-native/) for?

**Extension**

Why do we need to use [ObjectId](https://mongodb.github.io/node-mongodb-native/api-bson-generated/objectid.html) from the MongoDB driver?

Add to your diagram the dataflow for removing a game.

**Homework: Full Stack Games Hub App**

**MVP**

**Task**

Draw a diagram showing the dataflow through the application starting with a form submission, ending with the re-rendering of the page. This will involve a multi-direction data-flow with the client posting data to the server and the server sending data back to the client with the response. Detail the client, server and database in the diagram and include the names of the files involved in the process.

*Diagram of the dataflow through the application when a user submits the form*

**Questions**

What is responsible for defining the routes of the games resource?

**Answer**

What do you notice about the folder structure? Whats the client responsible for? Whats the server responsible for?

**Answer**

What are the the responsibilities of server.js?

**Answer**

What are the responsibilities of the gamesRouter?

**Answer**

What process does the the client (front-end) use to communicate with the server?

**Answer**

What optional second argument does the fetch method take? And what is it used for in this application?

**Answer**

Which of the games API routes does the front-end application consume (i.e make requests to)?

**Answer**

* What are we using the [MongoDB Driver](http://mongodb.github.io/node-mongodb-native/) for?

**Answer**

**Extension**

* Why do we need to use [ObjectId](https://mongodb.github.io/node-mongodb-native/api-bson-generated/objectid.html) from the MongoDB driver?

**Answer**

# MongoDB With a Server

**Lesson Duration: 120 minutes**

### Learning Objectives

* Understand the relationship between a client, server and database
* Be able to connect to a MongoDB database from a server
* Be able to create a JSON API using MongoDB and Express

## Introduction

By creating a RESTful API we are able to create a back-end service for our front-end applications to consume. Following API design principles to implement the CRUD operations, we can ensure that the API is intuitive to use. We have seen how to create a RESTful API using an Express server.

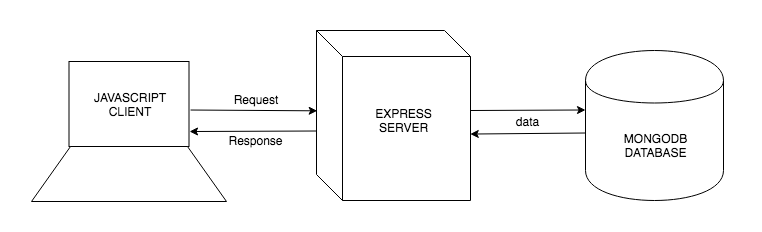
We have also seen how to use a MongoDB database to persist and retrieve non-relational data, using its shell to find and insert JSON documents into collections.

In this lesson we are going to create a RESTful JSON API for a games resource, using an Express server and a MongoDB database. We are going to define a set of routes that implement CRUD functionality and the data being served on the end-points will be retrieved by the server from the database.

## Games Hub Application

Instructor note: Hand out start code.

This is how the full stack JavaScript application, Games Hub, is structured.



Games Hub: A full stack JavaScript application

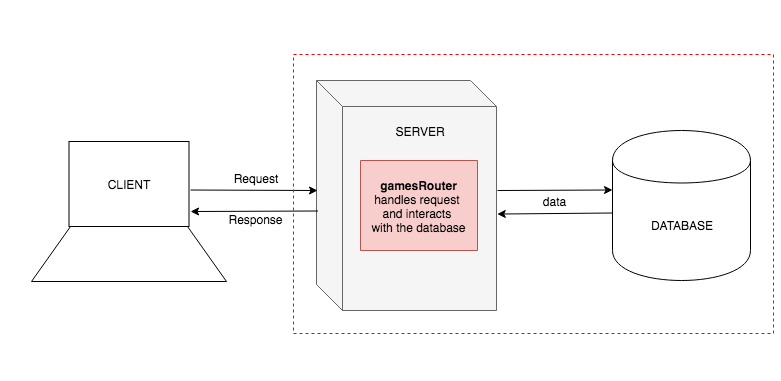
### Task: (10 minutes)

Run the start code using the instructions in the README.md. From reading any errors that appear in the browser console and from looking at the codebase, identify which functionality of the application hasn’t been written yet.

Answer

In this lesson we are going to be working server-side to:

* connect to the games\_hub database.
* access the games collection.
* create the routes for the games resource persisting and retrieving data from the database. We will be using Insomnia REST Client to test each route as we write it.



We will be working server-side to create the API that the front-end consumes

## MongoDB Driver

Until now we have been interacting with our MongoDB databases using the MongoDB shell, running a file with the command, mongo < file\_name.js. In order to interact with databases from inside our JavaScript applications, we are going to use the MongoDB Driver’s API. The MongoDB Driver is an npm package, so let’s start by installing it.

cd server

npm i mongodb

## Connecting to the Database

We want the routing for the games resource to be handled by a games router. Inside server.js we are going to connect to the games\_hub database, access the games collection, and create a games router, passing it an object representing the games collection, so that the games router can interact with the games collection in each of its routes.

Let’s start by requiring the MongoDB Driver that we previously installed in server.js, and accessing the MongoClient object.

*// server.js*

**const** express **=** require('express');

**const** app **=** express();

**const** path **=** require('path');

**const** MongoClient **=** require('mongodb').MongoClient; *// NEW*

We will use MongoClient’s connect method to connect to the MongoDB server. connect takes one argument: A URL. The URL must consist of MongoDB’s proprietary access mechanism (rather than HTTP which you will have more commonly seen), the location and the port number. MongoDB server runs on port 27017 by default.

app.use(parser.json());

MongoClient.connect('mongodb://localhost:27017') *// NEW*

Connecting to the MongoDB server is an asynchronous process and returns a promise. Once the promise has resolved, it will give us a client object that we can use to connect to the database. Because we getting a promise back from the connect method we will handle it with then. I.e, “When the connection to the server has been achieved, then give me the client object that I can use to connect to the database”. If the connection fails, we can use catch to log the error.

MongoClient.connect('mongodb://localhost:27017')

.then((client) **=>** { *// NEW*

})

.**catch**(console.error); *// NEW*

Note: There is only one semi-colon at the end of the expression.

We are going to use the client object to connect to the games\_hub database using the db method. We pass db the name of the database. Then we can access the games collection from the database using the collection method passing in the name of the collection.

MongoClient.connect('mongodb://localhost:27017')

.then((client) **=>** {

**const** db **=** client.db('games\_hub'); *// NEW*

**const** gamesCollection **=** db.collection('games'); *// NEW*

})

.**catch**(console.error);

We want to delegate the routing for the games resource to a games router. Let’s create a requiring createRouter.js.

**const** MongoClient **=** require('mongodb').MongoClient;

**const** createRouter **=** require('./helpers/create\_router.js'); *// NEW*

We are going to create a games router to handle the routing of the games resource. We will pass it the database’s gamesCollection because (although we haven’t written it yet) we know that the games router will need access the games collection in its route definitions.

MongoClient.connect('mongodb://localhost:27017')

.then((client) **=>** {

**const** db **=** client.db('games\_hub');

**const** gamesCollection **=** db.collection('games');

**const** gamesRouter **=** createRouter(gamesCollection); *// NEW*

})

.**catch**(console.error);

We tell the server we want to delegate the routing to it by the use method which takes a path and a router.

MongoClient.connect('mongodb://localhost:27017')

.then((client) **=>** {

**const** db **=** client.db('games\_hub');

**const** gamesCollection **=** db.collection('games');

**const** gamesRouter **=** createRouter(gamesCollection);

app.use('/api/games', gamesRouter); *// NEW*

})

.**catch**(console.error);

## Games Router

create\_router.js is going to be responsible for handling requests, and interacting with the database’s collection to persist and retrieve data. The createRouter function already takes in a database collection and creates and exports a router object. Now we can define each of the routes on the router object.

### Index Route

Let’s start with the index route. As before we will use the get method on the router, pass it a path and a callback that gets passed the request and response objects.

*// create\_router.js*

**const** createRouter **=** **function** (collection) {

router.**get**('/', (req, res) **=>** { *// NEW*

});

**return** router;

};

Let’s start by checking our route is working by sending back a string.

router.**get**('/', (req, res) **=>** { *// NEW*

res.send('Hello World!');

});

Now when we visit ‘http://localhost:3000/api/games’ we see ‘Hello World!’ displaying on the page.

When there is a request made to this route, we want the all the documents to be retrieved from the games collection in the database and sent back, as JSON, with the response. We have access to the games collection (collection), so we can call the find method on it to get all the documents back.

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

collection.find() *// MODIFIED*

});

The find method returns a cursor object containing the documents. We want an array of documents, so we can convert the cursor into an array, using the toArray method.

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

collection.find().toArray() *// MODIFIED*

});

Lastly, because the cursor’s toArray is asynchronous, it returns a promise. This means we can chain a then passing it a callback. When the promise resolves and the array of documents is ready, then will pass the array of documents to the callback (docs). We serialise the array into JSON and send it back on the response.

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

collection.find().toArray()

.then((docs) **=>** res.json(docs))

});

Now you can test the index route in Insomnia REST Client, by creating a new GET request, and making the request to http://localhost:3000/api/games. You will see the JSON response of the game objects you seeded your database with.

Because we are chaining a number of methods in this route, we can format it to make it more readable.

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

collection *// MODIFIED*

.find()

.toArray()

.then((docs) **=>** res.json(docs))

});

Now we can see more easily that we are:

1. calling find on the collection to get back a cursor object of all the documents
2. converting the cursor into an array with toArray
3. and then once that process has completed, we are sending the array of documents (docs), as JSON, back with the response.

We should also pass this a catch block in case of any errors.

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

collection *// MODIFIED*

.find()

.toArray()

.then((docs) **=>** res.json(docs))

.**catch**((err) **=>** {

console.error(err);

res.status(500);

res.json({ status: 500, error: err });

})

});

If there is a problem the server will respond with a 500 error code which means there was an internal server error.

We have now completed the index route, and you can test it using Insomnia Rest Client with a GET request to http://localhost:3000/api/games.

### Show Route

The show route, by convention, returns one object. It will the get method and the route path will need a parameter in the URL (for example, :id), so that the client can specify which game they want to receive when they make a request.

**const** gamesRouter **=** **function** (collection) {

*// ...*

router.**get**('/:id', (req, res) **=>** { *// NEW*

});

**return** router;

};

We are going to use the ID that the client specifies in the request to find the corresponding game object from the database. Let’s access the ID of the request’s params object, using our parameter name.

router.**get**('/:id', (req, res) **=>** { *// NEW*

**const** id **=** req.params.id;

});

Now we have the ID, we can use MongoDB’s findOne method, which takes a query object. If we were to search by name, we would do the following, findOne({name: 'Love Letter'}), but we can’t be sure every game has a unique name.

#### MongoDB’s ObjectID

When inserting a document into the database, MongoDB assigns it unique identifier with the key \_id. To query the database for an object with a particular ID, we want to do the following, findOne({\_id: '5af2d6f93776ded87a62a4ec'}). However, the ID will never match if the ID is passed as a string because MongoDB stores IDs as ObjectIDs](https://docs.mongodb.com/manual/reference/method/ObjectId/). Therefore to find a document, we have to pass the ID as an ObjectID. ObjectID comes from MongoDB so we need to require it.

**const** express **=** require('express');

**const** ObjectID **=** require('mongodb').ObjectID; *// NEW*

Now when we want to query the database for a particular ID, we can pass the findOne method the object { \_id: ObjectID(5af2d6f93776ded87a62a4ec) }.

router.**get**('/:id', (req, res) **=>** { *// NEW*

**const** id **=** req.params.id;

collection

.findOne({ \_id: ObjectID(id) })

});

Lastly, we want to send the found game object back, as JSON, with the response. As findOne is asynchronous (it takes time) and returns a promise, we will use then to receive the found game once the promise has been resolved. We can then convert the document to JSON and send it back with the response.

router.**get**('/:id', (req, res) **=>** {

**const** id **=** req.params.id;

collection

.findOne({ \_id: ObjectID(id) })

.then((doc) **=>** res.json(doc)) *//NEW*

.**catch**((err) **=>** {

console.error(err);

res.status(500);

res.json({ status: 500, error: err });

});

});

});

We have now completed the show route, and you can test it using Insomnia Rest Client with a GET request to http://localhost:3000/api/games/[existing ID].

### Create Route

### Task: (10 minutes)

Your task is to create the “create” route.

You will need to:

* check that the server is configured to use body-parser
* define a new route that handles POST requests
* access the new game object from the request’s body
* insert the game into the games collection using the insertOne method
* send back all the documents from the collection with the response. Note: insertOne is asynchronous and returns a promise, so use a then to access all the games from the collection once the promise has resolved and convert the documents into to an array
* finally, we know that the cursor method, toArray, is asynchronous and returns a promise, so use another .then to convert the array into JSON and send it back with the response.

Test the create route in Insomnia REST Client, by creating a new POST request and adding the following JSON object to the body:

{

"name": "Chess",

"playingTime": 60,

"players": {

"min": 2,

"max": 2

}

}

Make the request to http://localhost:3000/api/games/ and you will see the JSON response of the all the game objects including the one you added.

Example solution

This works, however its not very efficient. Imagine we had millions of entries in our database, when we add one, do we really want to get them all back?

We can make this more efficient.

router.post('/', (req, res) **=>** {

**const** newData **=** req.body;

collection

.insertOne(newData)

.then((result) **=>** { *//UPDATED*

res.json(result.ops[0]) *//UPDATED*

})

.**catch**((err) **=>** {

console.error(err);

res.status(500);

res.json({ status: 500, error: err });

});

});

After we’ve inserted the document we’re accessing the result of this function. Result contains the result document from MongoDB and ops contains the documents inserted with added \_id fields.

Now we’re just returning the object we’ve inserted rather than all the objects.

We have now completed the create route.

### Destroy Route

### Task: (10 minutes)

Your task is to create the destroy route.

You will need to:

* Define a new route that handles DELETE requests with an id parameter in the path
* Access the ID from the request’s params object
* use deleteOne to delete the document in the games collection that has an ID that matches the ID specified in the request. To do this, pass deleteOne an object with the key of the property you want to search by, and the value you want to search with: { \_id: ObjectID(id) }
* Send the result as JSON. Unlike the insertOne function we don’t want the object back, we just want the result in JSON format. Note: deleteOne is asynchronous and returns a promise, so use then to access all the results.

Example solution

We have now completed the destroy route, and you can test it using Insomnia Rest Client with a DELETE request to http://localhost:3000/api/games/[existing ID].

The result that we get back is a confirmation that the document has been removed.

{

"n": 1,

"ok": 1

}

### Update Route

Now, let’s write the update route. We will use the put method, and pass it a path with an id parameter, and a callback that will receive the request and response.

router.put('/:id', (req, res) **=>** {

});

To update a document in the database we need find the correct document from the database, and update it with the new values from the request.

Let’s start by getting the ID of the object we want to update from the request’s params object, and the new game object with the updated values from the request’s body object.

router.put('/:id', (req, res) **=>** {

**const** id **=** req.params.id;

**const** updatedData **=** req.body;

});

Now we are going to use those two piece of information to update the document in the database’s collection, using the collection’s findOneAndUpdate method. findOneAndUpdate takes two arguments, both objects. It also takes a third optional options object:

1. The first is used to query the collection to find the document that matches the filter (in our case we are searching by ID)
2. The second is an object that has the property $set. This is telling MongoDB to set each of the fields of the found document, with the values of the corresponding properties of findOneAndUpdate.
3. In the options object we’re passing the key of returnOriginal and setting it to false. By default findOneAndUpdate returns the object before its been updated. Setting this value to false will return us the new object.

router.put('/:id', (req, res) **=>** {

**const** id **=** req.params.id;

**const** updatedData **=** req.body;

collection

.findOneAndUpdate(

{ \_id: ObjectID(id)},

{ $set: updatedData },

{returnOriginal: **false**}

)

});

Once that is complete we want to get that document back. findOneAndUpdate is asynchronous and returns a promise, so we will use then to access the result and res.json() the result.value. The result object contains conformation of the update. To access just the updated object we need to access .value.

router.put('/:id', (req, res) **=>** {

**const** id **=** req.params.id;

**const** updatedData **=** req.body;

collection

.findOneAndUpdate(

{ \_id: ObjectID(id)},

{ $set: updatedData },

{returnOriginal: **false**}

)

.then((result) **=>** {

res.json(result.value)

})

.**catch**((err) **=>** {

console.error(err);

res.status(500);

res.json({ status: 500, error: err });

});

})

Test the update route in Insomnia REST Client by creating a new PUT request and adding an updated JSON game object into the body. Make the request to http://localhost:3000/api/games/[existing ID] and you will see the JSON response of the game object you updated.

Important: Do not include the \_id property in the updated game object when you send the put request. MongoDB will give an error if you try to update a document by giving it an object with a key, \_id.

We have now completed the update route.

We have now created a RESTful JSON API for our games resource, with data being persisted in a MongoDB database. We have tested all of the routes in Insomnia Rest Client to ensure they work.

## Cors

We have one more thing to do to complete our application. With the server still running, navigate to your client folder and run npm run serve to get the Vue development environment running.

npm run serve

No games are rendering. They should be as there’s a fetch request in GamesGrid to bring all of the games back from the database. The terminal is telling us what the error is.

Access to fetch at 'http://localhost:3000/api/games' from origin 'http://localhost:8080'

has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header

is present on the requested resource.

We need to install the npm cors package for the server. From the server directory:

npm i cors

And then all we have to do is tell Express to use it.

*//server.js*

**const** cors **=** require('cors') *//NEW*

app.use(parser.json());

app.use(cors()); *//NEW*

Now when you refresh your browser you should see all the games from the database rendering on screen, with the ability to add and delete these games.

## Recap

Why do we need to use MongoDB Driver?

Answer

Why do we need to use promises in the games resource routes?

Answer

## Conclusion

To create a RESTful JSON API we used an Express server to define a set of routes, and the MongoDB Driver to interact with the data stored in a MongoDB database. We have completed the back-end of a full stack JavaScript application.

# Lab: Full Stack Application POST Request

**Lab Duration: 90 minutes**

### Learning Objectives

* Understand the data flow through a full stack application
* Be able to make a post request from a front-end application and persist the data in a database

## Introduction

You have been given an existing application that allows bird watchers to keep a record the species, dates and locations of bird sightings. The functionality that allows the user to save their sighting is not yet complete. Your task is to complete it.

## Task

### MVP

* Implement the functionality that allows a users to submit the form and for their bird sighting to be persisted so they can view their sightings at a late date. The webpage should display a list of all the saved bird sightings.

### Extension

* Implement the functionality that allows the user to delete a sighting.

## Planning

Identify which files in the application you are going to need to modify and how you are going to test each modification as you go. As a general rule of thumb, read your error messages - it may not be the case that everything you need to begin is actually there.

# Full-stack Lab: Hotel Bookings

**Duration: All Day**

## Learning Objectives

* Be able to create a full-stack app
* Practice building a server with a MongoDB for persistence
* Understand the relationship between a client, server and database

## Brief

Build a system to allow a hotel to manage bookings. There should be a form to allow the hotel to add the following information for each booking:

* Guest Name
* Guest Email Address
* Checked in status

Bookings should be stored in a MongoDB via an JSON API in Express.

## MVP

* Hotel managers should be able to view a list of all bookings
* Hotel managers should be able to add a new booking, which should update the page without the page being refreshed
* Hotel managers should be able to delete bookings

## Extensions

* Don’t allow a booking to be submitted to the API unless both the name and email address are present

## Advanced Extensions

* Allow the hotel manager to update the “checked in” status of the booking

## Considerations

Test your code often. Try not to write very much code before you run it again. We value working software, even if it is not feature-complete and want to isolate problems so we can fix them more quickly. You can use Insomnia REST Client to test the Express API.

## Planning

Wireframe the client application before you build anything; what components will you need? What state (data) will each component hold? What will the hierarchy of components be?

# Advanced Git

## Learning Objectives

* Know Git naming conventions
* Be able to create and move between branches
* Understand the difference between implicit and explicit merges
* Be able to merge branches and resolve merge conflicts
* Be able to use pull requests to merge branches
* Be able to apply Git hygiene
* Be able to use Git as part of a team

## Introduction

We have been using GitHub as a versioning tool to enable us to go back to previous version of our code. Git has a feature called branching that allow us to create parallel versions of our projects, so as to ensure we always have a stable version readily available. GitHub also has a whole host of features that allows developers to collaborate on projects, working on the code simultaneously.

## Git’s Master Branch

Until now we have been working on one branch, the ‘master’ branch. In a Git initialised directory, you will see the word ‘master’ after the directory name in the terminal. This is the default branch that is created when we create a Git repository. (whether local or remote)

Until now, every time you have been committing your changes, you have been committing to master. You have then been able to use git commands to go back to a previous version of your code. One problem with this process is that it can be difficult to find the commit relating to the version you want to go back to. Branches help with this problem.

## Git Branches

With branches, once we have a stable version of a program committed to git and we are about to add a new piece of functionality or refactor the code in some way, we can create a new branch. This creates a copy of the code. The new branch will be in the same state as the branch you branched off. Now you can work in the new branch, adding your new feature or refactoring, knowing that you have a stable version back on the original branch should you decide to abandon the feature or you decide the refactor wasn’t an improvement.

### Branch Naming Conventions

Permanent Branches:

* “master” is the name of the default branch. This should always contain stable working code.
* “develop” is often used as the branch for a stable version of code, that is being added to regularly during the development process. Like the master branch, it should always contain working stable code.

Temporary Branches:

* Feature branches are used to develop new functionality. The naming convention is ‘feature/name\_of\_feature’. These branches are deleted after the feature is integrated back into develop.
* Fix branches are used to fix a bug. There are different conventions, but you can use a prefix to indicate its purpose like, ‘fix/name\_of\_fix’. These branches are deleted after the fix is integrated back into develop.

## Git Merging and Conflicts

When we merge two branches together, we integrate the code from one into the other. For example, when we have completed the feature on the feature branch we want and integrate the code into the main branch. Sometimes Git can do the integration process automatically for us, but when there have been changes made on both branches, Git may not know which lines to keep and which to discard, and we have to do it manually. This is called a merge conflict. This is a common problem when working collaboratively, where different people are working on different branches simultaneously and learning how to handle them is an important part of learning how to use Git.

The principle of merging branches is to first merge the stable branch (e.g. develop) into the potentially unstable branch (e.g. feature) so that any conflicts can be dealt with and fixes made there. Once you are sure your feature branch is stable, you then merge it into develop. This ensures the stable branch remains stable.

## Task (45 mins)

In pairs work through the following steps to create and merge branches and deal with any git merge conflicts. Work through the steps together, so you understand each part of the process.

You are going to start with a master and a develop branch. Then you are going to be making feature branches and merging them into develop.

## Person One

Person One is going to start by creating the directory structure and a develop branch.

### Master branch

1. Create a local directory called git\_lab and initialise git inside it - git init
2. Create a new repo called ‘git\_lab’ on GitHub and add it as the remote of your local directory git remote add origin git@github.com:your\_github\_username/git\_lab.git
3. Create a file called greet.js inside the directory
4. Commit and push the changes

We now know that the local and remote master branch are the same.

### Develop branch

1. Create a develop branch off of master - git checkout -b develop. (The -b flag indicates you are making a new branch, and it is followed by the name of the branch. This command will move you into develop, so you will see the word ‘develop’ in your terminal, next to the directory name.)
2. Add the following to greet.js:
3. **const** helloWorld **=** () **=>** {
4. **return** "Hello World!";
5. }
6. Add and commit your changes.
7. Push to develop branch - git push origin develop. We now know the local and remote develop branches are the same. These changes have **only** been committed on the develop branch. The master branch has not been not affected.
8. Add your partner as a collaborator to the repository on GitHub by going to the repository page. Click settings then collaborators and add a new collaborator. This will enable your partner to push and pull to the repository directly.

When you move between branches, your editor will automatically update to display the state of the current branch. If you now checkout to master, those changes you have just made will not be displayed in the editor. Then when moving back to develop, they will appear again.

## Person Two

Person Two is going to work on a feature and merge it into develop.

1. Accept the invitation (you will have been sent an email).
2. Clone the repo. You will now have all the existing branches from the remote in your local cloned version.
3. Checkout to the develop branch - git checkout develop. (Notice there is no -b flag in the command because we are not creating a new branch, we are checking out to an exiting branch).
4. Create a new new feature branch - git checkout -b feature/add\_default\_param - notice we are branching off of the develop branch, so our new branch will be the same as develop.
5. Refactor the function to have a name parameter with a default value:
6. **const** helloWorld **=** (name **=** "World") **=>** {
7. **return** `Hello ${name}!`;
8. }
9. Add and commit the changes
10. Push to the feature branch - git push origin feature/add\_default\_param.

We know that ‘feature/add\_default\_param’ local and remote branches are the same.

### Merging

Person two is happy that their feature branch is finished, and now wants to integrate their changes into develop. However, person two doesn’t know if someone else has made changes to develop while they have been working on their feature branch, so they need to start by getting the latest version of develop.

1. Checkout develop - git checkout develop.
2. Pull develop - git pull origin develop - we now know that develop remote and local are the same so we can go ahead with the merge.

The end goal is having all our changes integrated into develop.

The principle of merging is that we first merge the stable branch (develop) into the feature branch, so we can fix any conflicts in the feature branch, before merging into develop. This means that develop is always stable.

1. Checkout to the feature branch.
2. Merge develop into feature/add\_default\_param - git merge develop. You can think of the merge command as “pulling” the branch specified in the command into your current branch. Make sure you are in your feature branch. We want to deal with any conflicts in the feature branch before merging into develop. (**If git tells you that your branch is ‘already up-to-date’, skip to the next section: ‘Merging Feature into Develop’**)
3. Resolve any conflicts. When merging, sometimes there are conflicts. This means that git can’t automatically merge the changes. Git will list the files that contain the conflicts in the terminal window. For exmaple:

CONFLICT (content): Merge conflict in greet.js

Automatic merge failed; fix conflicts and then commit the result.

If you open the file in Atom you will see something like the following:



Unresolved Git Conflict in the Editor

The ‘head’ section shows your branches state, and the ‘develop’ section shows the state of the develop branch. You need to edit the file to be how you want it, check your code still runs, then add, commit and push the changes to your feature branch.

The “Use me” buttons are added by the Atom text editor, but the symbols (<<<<<<<, ======= and >>>>>>>) are just text that has been added to your file. You must remove the symbols, and combine the two versions of the file to create the finished. Atom’s “Use me” feature tries to make this easier, allowing you to choose one version and removing the symbols for you. However, you do not have to use this, you can just edit the file yourself.

Now you know your feature branch has all the latest changes from develop integrated, so you can merge it back into develop without any issues.

1. Add & commit your changes
2. Push to the feature branch. We now know local and remote of our feature are the same and that we have the latest version of develop integrated with our changes.

#### Merging Feature into Develop

Now want to merge our feature into develop:

1. Checkout develop - git checkout develop.
2. Do a pull on develop just to double check no new changes have been made on develop in the meantime - git pull origin develop. Note: If changes had been made to develop, you would revert back to step three of previous section.
3. Merge feature/add\_default\_param into develop - git merge feature/add\_default\_param. There should be no conflicts at this stage, as we already dealt with any conflicts on the feature branch.
4. Push to develop - we now know that the local and remote are the same in develop and our feature branch has been merged.

## Merging with conflicts

Each person is now going to work on a branch simultaneously, and a conflict is going to be created that you will need to handle.

Each person should:

1. Double check they have all the latest changes on develop - git pull origin develop
2. Make their own fix branch (Ensure you branch off develop):
   * Person 1: ‘fix/implicit\_return’,
   * Person 2: ‘fix/rename\_function’.
3. Complete the fixes:
   * Person 1: Refactor the function to use the arrow function’s implicit return
   * **const** greet **=** (name **=** "World") **=>** `Hello ${name}!`;
   * Person 2: Rename the function from helloWorld to greet.
4. Both should add and push their changes to their respective branches.
5. One after the other, each person should follow the above merging process, finishing with all changes merged into develop.

## Merge into Master

Finally merge develop into master following the same merging process as above. First merging master into develop, dealing with any merge conflicts on develop, (there shouldn’t be any in this case) and then merging develop into master.

## Conclusion

You have now followed a work-flow that enabled you to simultaneously have multiple people working on the same project using branches, ensuring that you always have two stable versions of the code (on the master and develop) branch. This means that at any point in time during development you always have a working application.

**Agile and Scrum**

**Learning Objectives**

* Know what Agile is and how Scrum implements it
* Understand the benefits of Agile and Scrum in web/software development
* Know the roles that make up Scrum teams

**Introduction**

When developing software there are often many constraints involved, such as user requirements, deadlines and budgets. When working as a team to deliver a project, it is important to agree on the type of project management strategy being used. Each member of the team must be clear on their role and responsibilities to ensure the final project meets the requirements and is delivered on time and within budget.

Agile is a popular methodology used in software development and has been developed in response to some of the problems that arise from other project management processes. This lesson is going to cover Scrum, which is an implementation of Agile, and the benefits of using its practices when developing software in teams.

We will start by looking at another project management methodology, Waterfall, which produces some of the problems that Agile aims to solve.

**The Waterfall Methodology**

The Waterfall method takes a linear approach to software development. In this methodology, the sequence of events could happen in the following order:

* Gather and document requirements
* Design
* Build with unit and integration testing
* Perform system testing
* Perform user acceptance testing (UAT)
* Fix any issues
* Deliver the finished product

**Drawbacks of Waterfall**

Gathering and documenting requirements in a way that is meaningful to a customer is often the most difficult part of software development; clients may not be completely sure what they need, or their requirements might not be documented accurately.

With Waterfall, the full product is built inline with the set of user requirements gathered from the customer at the start of the process. However, as the user’s requirements might change in the time it takes to build the product, or they might look at the final product and feel that it is not representative of what was initially discussed, this process can lead to large losses in time. Making changes to the product once it has been fully built can be difficult and in the worst case, it can lead to the whole product being scrapped and the process started again from the beginning.

It was realised that it would be better to catch the inaccuracies of requirements and any required changes earlier in the development process.

**Agile**

The Agile Method is an approach to project management that aims to respond to the unpredictability of developing software products, by focussing on incremental releases with frequent feedback cycles.

The [Manifesto for Agile Software Development](http://agilemanifesto.org/) written in 2001 states that agile software development aims for:

* Individuals and interactions over processes and tools
* Working software over comprehensive documentation
* Customer collaboration over contract negotiation
* Responding to change over following a plan

**Scrum**

Scrum is an implementation of agile and is one of agile’s most popular frameworks, defining a set of practices that enable the above aims.

With scrum, the product is built in a series of fixed-length iterations called Sprints that give teams a framework for shipping software on a regular basis. Milestones (the end of a sprint) come frequently bringing with them a feeling of tangible progress with each cycle that focuses and energises everyone. Short iterations also reinforce the importance of good estimation and fast feedback from tests and the client – both recurring struggles in waterfall projects.

Scrum done well can massively improve team productivity and morale, and the product development process as a whole.

**Scrum Team**

A Scrum team has a minimum of three specific roles:

**1. The Product Owner**

Product owners are the champions for their product. They are focused on understanding business and market requirements, and prioritising the work to be done by the engineering team accordingly. Effective product owners:

* closely partner with the business and the team to ensure everyone understands the work items in the product backlog.
* give the team clear guidance on which features to deliver next.
* decide when to ship the product with the predisposition towards more frequent delivery.
* keep in mind that a product owner is not a project manager.

Product owners do not manage the status of the program. They focus on ensuring the development team delivers the most value to the business. Also, it’s important that the product owner is an individual. No development team wants mixed guidance from multiple product owners.

**2. Scrum Master**

Scrum masters are the champions for scrum within their team. This could be a job role in itself, or the product owner or a developer may take this role. The scrum master coaches the team, the product owner, and the business on the scrum process and look for ways to improve their practice of it. An effective scrum master deeply understands the work being done by the team and can help the team optimise their delivery flow. As the facilitator-in-chief, they schedule the needed resources (both human and logistical) for sprint planning, stand-up, sprint review, and the sprint retrospective.

Scrum masters also look to resolve impediments and distractions for the development team, insulating them from external disruptions whenever possible. Part of the scrum master’s job is to defend against an anti-pattern common among teams new to scrum: changing the sprint’s scope after it has already begun. Product owners will sometimes ask, “Can’t we get this one more super-important little thing into this sprint?”, but keeping scope air tight reinforces good estimation and product planning–not to mention fends off a source of disruption to the development team.

**3. Development Team (Developers, Testers, Designers, UX, etc.)**

The development team drives the plan for each sprint. They forecast how much work they believe they can complete over the iteration using their experience as a guide.

Members of the development team are the champions for sustainable development practices. The most effective scrum teams are tight-knit, co-located, and usually 5 to 7 members. Team members have differing skill sets, and cross-train each other so no one person becomes a bottleneck in the delivery of work. Strong scrum teams approach their project with a clear “we” attitude. All members of the team help one another to ensure a successful sprint completion. The team might include testers, designers, and ops engineers in addition to developers.

A scrum team controls its own workflow and self-organises around their defined tasks. Agile teams use pull models where team member’s pulls a certain amount of work off the backlog and commits to completing it that sprint, which is very effective in maintaining quality and ensuring optimum performance of the team over the long-term. Neither scrum masters nor product owners push work to the team (which, by contrast, tends to erode both quality and morale).

**Sprints**

Scrum calls for four ceremonies that bring structure to each sprint:

**1. Sprint Planning**

A team planning meeting that determines what to complete in the coming sprint.

**Requires**: Development Team, Scrum Master **Optional**: Product Owner **When**: At the start of a sprint. **Purpose**: The Sprint Planning meeting is where the team defines the the work that will be completed in the sprint . Depending on the process being used by the Scrum team, user stories and a backlog are created, which structure the workflow for the entire sprint. A popular system for managing the sprint workflow is the Kanban board (see section ‘Sprint Planning: Kanban Board’). The Sprint Planning meeting is where the Kanban board would be created.

**2. Daily Stand-up**

Daily stand-up (also known as a daily scrum) is a 15 minute mini-meeting for the development team to synchronise.

* **Requires**: Development Team, Scrum Master
* **When**: Once per day
* **Purpose**: Stand-up is designed to quickly inform everyone of what’s going on across the team. It’s not a detailed status meeting. Each team member answers the following questions:
  1. What work did I complete yesterday?
  2. What will I work on today?
  3. Am I blocked by anything?

**3. Iteration Review**

Work completed during the sprint is demonstrated and feedback is gathered.

* **Required**: Development Team, Scrum Master, Product Owner
* **Optional**: Project stakeholders
* **When**: At the end of a sprint
* **Purpose**: Iteration review is a time to showcase the work of the team. They can be in a casual format like “demo Fridays”, or in a more formal meeting structure. This is the time for the team to celebrate their accomplishments, demonstrate work finished within the iteration, and get immediate feedback from project stakeholders. Remember, work should be fully demonstrable and meet the team’s quality bar to be considered complete and ready to showcase in the review. Incremental feedback gathered from stakeholders at each demo ensures the product is meeting requirements throughout the development process.

**4. Retrospective**

A meeting where the sprint is evaluated.

* **Required**: Development Team, Scrum Master, Product Owner
* **When**: At the end of the sprint
* **Purpose**: Agile is about getting rapid feedback to make the product and development culture better. Retrospectives help the team understand what worked well–and what didn’t and is the opportunity to identify actions to make the next sprint better.

**Sprint Planning: Kanban Board**

Sprints will often implement a system for defining tasks and keeping track of who is doing what. Scrum encourages using a pull model, where team members assign themselves to the approriate tasks, rather than tasks being delegated by a project manager. The Kanban board is a popular example of such a system where tasks are defined in a backlog, and moved from ‘doing’ to ‘done’ as they get completed. As a team member takes a task from the backlog, they can assign themselves to it, so everyone can see who is working on what.

A Kanban board is developed and used in the following way:

* The product owner creates a list of user stories for the sprint. A story or user story is the smallest unit of work in an agile framework. It is a software system requirement that is expressed in a few short sentences, ideally using non-technical language. For example: “A user can input departure and return dates and view a list of available flights”.
* The full Scrum team collectively decide how the divide these user stories into tasks. These are the granular pieces of work that help define the implementation items for the story and the upcoming sprint.
* The backlog is created. A product backlog is a prioritised list of work for the development team that is derived from the roadmap and its requirements. All work items should be included in the backlog: user stories, bugs, design changes, technical debt, customer requests, action items from any previous retrospective.

The Kanban board can be a powerful motivators. They drive a spirit of “we’re doing this!”. They also allow the development team to self-organise the division of tasks, as team members can take tasks of the backlog themselves, replacing the more traditional project manager role.

**Conclusion**

The Agile methodology was developed to solve common problems that occur when developing software with other project management strategies. Agile aims are:

* Individuals and interactions over processes and tools
* Working software over comprehensive documentation
* Customer collaboration over contract negotiation
* Responding to change over following a plan

Scrum is an implementation of Agile which provides a set of practices that enable the Scrum team to define, manage and track the product development process with short iterations of feature development and frequent feedback cycles.