Lesson plan section was successfully destroyed. ×

MongoDB/Mongoose Part 1

REVIEW: The Express App — 00:10

Author: Shaun McKinnon

QUIZ: Express App — 00:10

Author: Shaun McKinnon

SQL VS NoSQL — 00:30

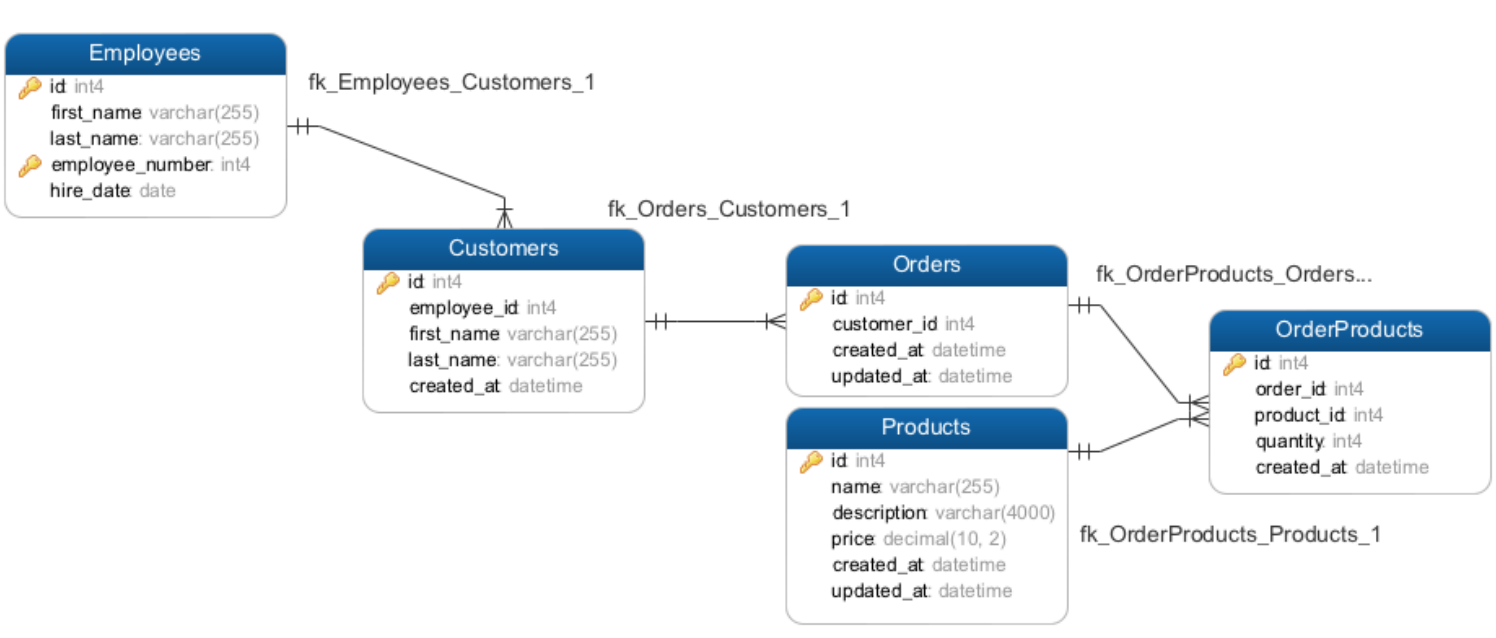
https://medium.com/chingu/an-overview-of-mongodb-mongoose-b980858a8994

What is a database?

There are several different types of databases, but they share one fundamental purpose: to store information. There are 3 fundamental types of databases that you will likely see in your quest for work: SQL, NoSQL, and Distributed File System. The latter one is less common, but if you're interested I would recommend Googling Hadoop. The other two are quite common and the subject of debates, as some appreciate the structure and relationships of SQL, while others like the flexibility and potential scalability of NoSQL.

What is so great about SQL (Relational DBs)

In the world of business, business people love to see structures that represent their business. **Customers**, **Employees**, **Orders**, **Products**, etc. In a SQL DB these would be likely tables, and they represent business data related to their names. In addition, we can visualize the relationships between these tables:



This structure will allow us to view all the customers under a specific employee. In addition we can see all the orders a customer has made and what products they had ordered. From that we can build analytical data on what products are more popular; what products are purchased in certain area; which employees have more active customers; which employees have signed the most customers.

SQL gives us several queries that allow us to control how we interact with data. Because there are so many different SQL databases, we can't be sure what queries we have available to us, as well as what datatypes there are available. This leads into the need for an ORM (Object Relational Mapper) layer. This layer abstracts our database into objects with attributes, allowing us to disregard data-typing at the database level. For example, MySQL has Enum and PostgreSQL does not. The ORM, when

interacting with MySQL, will store enum values as enum strings. The ORM, when interacting with PostgreSQL, will instead store the values as integers. How you implement enum will be based on the ORM's requirements. This means that you learn the ORM and not the database.

What is so great about NoSQL DBs?

Because SQL relies on data relationships, this requires us to plan our database requirements in depth, in advance. Scalability becomes more difficult as new data requirements require us to migrate old data to new schemas. I'm sure you can see the potential problems here: missing data, corrupted data, improperly copied data, dropped relationships, etc... Scaling strict relational databases requires a lot of planning to avoid the above issues as well as money to pay for that planning. Often we will implement redundancies so we can rollback in case any errors occur. We will spin up entire redundant applications in order to ensure there is no downtime in case of an error.

NoSQL lowers the need for redundancy and planning. You can add new schema without affecting the old easily and in place without damaging your application. Because it's document store, your database can horizontally scale across new systems. SQL requires more resources in order to scale.

When should I use what?

Let's use some scenarios to determine the best choice:

I need an application to store an Organizational Hierarchy:

Requirements
A table for employees information
- name
- employee number
- position_id

A table for the organizations positions
- title
- parent position_id

My schema is fairly fixed. My employee information will be based on my business needs. The position table will allow me to build a hierarchy easily, and not require any flexibility. My data requires integrity to maintain its structure. A SQL database is made for this, and therefore the best choice for this scenario.

Let's look at a different scenario:

I need an application to store a Catalogue of Products:

Requirements
A table to store a product:
BUT WAIT! Some products have specifications. Some have configurations. Some have features. Some have benefits. Some have minimum purchase amounts. Some of maximum purchase amounts. Some become cheaper based on quantity.

A table to store a product:
- name
- description
- msrp

A table to store specifications
A table to store configurations
A table to store features
A table to store benefits
- type (spec, config, feat, or ben)
- product_id
- label
- description

That doesn't work:
For this specific spec I have images and tables for it. For this configuration, I want it to affect the price because it requires more material. This feature is tabular data, but this feature is bullet pointed.....F***!

In case you think this didn't happen: http://www.asbheat.com/products
We used a relational database for the mess described above. Our hierarchy was a nightmare and eventually we gave up. We

created a catch all table and coded out each section individually based on its requirements. We essentially did document storage in a relational database.

A NoSQL solution:

```
{
  name: '',
  msrp: '',
  details: {
    specifications: {
    },
    features: {
      tabbular: {
        tab1: '',
        tab2: '',
        tab3: ''
      },
      table: {
        header: [],
        body: []
      }
    }
  },
  price_affects: {
    quantity: {
      less: {
        amount: 15,
        mutation: {
          increment: 10,
          type: 'percent
        }
      },
      more: {
        amount: 100,
        mutation: {
          decrement: 20,
          type: 'percent
        }
      }
    }
  }
}
```

Many large cart systems utilize relational databases. Products are one of those beasts that will shift and change. Think of even product specific stores like a car dealership. Yes, every car will have tires, body type, and colour, but they will also have other details that affect specific types of cars. A utility van and a passenger car are not the same when we consider features and benefits. NoSQL allows us to use unstructured data and search that data efficiently.

One note: many companies are starting to see the need for non-hierarchical data structures. These same companies are having difficulty changing into systems like Mongo. Instead they're introducing JSON and JSONB columns into their DBs, or they're adding 'tagging' systems that allow them to group data dynamically. Always remember, **right tool for the job**.

Author: Shaun McKinnon

BREAK — 00:15

Installing and Running Mongo — 00:20

Introducing MongoDB

- Collections instead of tables
- Documents instead of records

- Documents can nest inside other documents (no joins)
- Documents are structured with Key/Value pairs
- Mongo comes from the word Hu**mongo**us, meaning huge

Installing MongoDB

1. Download MongoDB
   https://www.mongodb.com/download-center?jmp=nav#community
   - **NOTE** if you are on a Mac, download and install HomeBrew and use it to install Mongo. It will save you **MANY** steps!
   - https://brew.sh/
2. Install Mongo
   - Mongo is a self-contained application
3. To run Mongo from the command line, we'll need to add it to our PATH (if you're on Windows)
   1. Navigate to C:\Program Files\MongoDB\Server\3.0\bin
   2. Copy this path
   3. Press the **Windows key**, and type **cmd**
   4. Type **env** and choose the **environment variables** option
   5. Click **Environment Variables**
   6. Click on **PATH** and click **Edit**
   7. Click **New** and paste your path in the box
   8. Click ok until all the windows are closed
4. If you are on a Mac, do yourself a HUGE favour and install Brew (**see step 1**)
   1. Otherwise you will need to find the Mongo installation, copy the path, and add it to your .bash_profile (or .zshrc, or whatever shell profile you may have)
5. Next you will need to create the **/data/db** directory set so Mongo can create its db files
   1. Open terminal/bash/command line and type the following
   2. On **Windows**:
      1. **mkdir \data**
      2. **mkdir \data\db**
   3. On **Mac**:
      1. **sudo mkdir /data**
      2. **sudo mkdir /data/db**
      3. **sudo chown -R $USER /data/db**
         - The above command is due to ownership
         - We are changing the ownership of this directory to be for you
         - ***If you have more than one user and you want access to this directory for writing***, you will need to run a different command: **STOP (ONLY RUN IF YOU'VE UNDERSTAND WHAT THIS IS DOING)**
           - **sudo chmod -R go+w /data/db**
6. You are all set!

Running Mongo

In order run Mongo, we'll need to start the daemon. The daemon will start the MongoDB server and listen for connections. This is essentially the same as having a MySQL database server which we would connect to when we need to.

1. Let's start Mongo
   1. Type **mongod**
   2. If everything is working, you'll see the Mongo server startup and wait for us to connect
2. Open a new CLI window
   1. We need this to connect to our running Mongo Server
   2. The server **HAS** to be running in order for us to connect to
3. In the new CLI
   1. Type **mongo**
   2. This will connect to the running Mongo server
   3. It will then give us a prompt to run MongoDB queries
4. Let's test one query command to ensure everything is golden, Pony Boy (Rumble in the Bronx ref)
   1. **show dbs**
      - admin
      - local
   2. I have 2 databases, you may have more
5. That's it for now

Author: Shaun McKinnon

Understanding JSON — 00:15

JSON is a data structure that allows us to represent data in a key/value pairs. Originally the common format for working with this type of data was to use XML, however, XML has limitations and is much larger to move between endpoints. It also, in my opinion, isn't as human readable as JSON is.

JSON confuses newcomers due to two syntax operators: the **{}** and the **[]**. These operators represent the opening and closing of blocks. The curly braces, {}, represent **objects**, which are always in key/value pairs:

```
{
  key: 'value',
  key2: 43,
  key3: 19.56
}
```

Keys in an object must be unique. It is "illegal" to have 2 keys with the same name. You can think of keys as being similar to columns in a database table. The difference is that keys are not limited to symbol naming restrictions like columns are:

```
{
  'key one': 'value',
  'key 1': 43,
  key1: 19.56
}
```

All of the above are valid in JSON.

The value of a key can also be an object, allowing us to have **nested** objects:

```
{
  'key one': {
    nested_key: 'value',
    second_nested_key: 42
  },
  'key 1': 43,
  key1: 19.56
}
```

The key symbol names are scoped to their block, meaning that you can have 2 keys with the same name as long as they are nested:

```
{
  'key one': {
    'key one': 'value',
    key1: 42
  },
  'key 1': 43,
  key1: 19.56
}
```

If you name a key the same, within the same block, the second key will override the value of the first.

The second set of operators are the opening and closing of an array: **[]**. These allow us to past lists to our JSON structure. It is important to note, that this is not a valid JSON structure:

```
{
  [
    'value 1',
    'value 2',
    'value 3'
  ]
}
```

You must have a key:

```
{
  the_mighty_key: [
    'value 1',
    'value 2',
    'value 3'
  ]
}
```

This is also not a valid JSON structure:

```
[
  'value 1',
  'value 2',
  'value 3'
]
```

You must have the basic **{key: []}** structure.

Array structures allow for multidimensional arrays as well:

```
{
  the_mighty_key: [
    'value 1',
    'value 2',
    [
      'sub value 1',
      'sub value 2'
    ]
  ]
}
```

Your only limit to nesting is really your imagination. You can also nest JSON objects within arrays:

```
{
  the_mighty_key: [
    'value 1',
    'value 2',
    [
      'sub value 1',
      'sub value 2'
    ],
    {key: 'value 1', key2: 'value 2'},
    {key: 'value 1', key2: 'value 2'},
    {key: 'value 1', key2: 'value 2'}
  ]
}
```

JSON is the language of our NoSQL db, Mongo. The great thing about this is all the above structures are valid as Mongo documents as well as Javascript objects. Hence the name: **J**ava**S**cript **O**bject **N**otation.

Author: Shaun McKinnon

CRUD with Mongo CLI — 00:15

Simple CLI Crud Commands

OPERATION	STATEMENT	EXAMPLE
Creating/Using a Database	use DATABASE_NAME	use test_db
Dropping a Database	db.dropDatabase()	use test_db db.dropDatabase()
Creating a Collection	db.createCollection(name)	db.createCollection("test_collection")
Dropping a Collection	db.COLLECTION_NAME.drop()	db.test_collection.drop()

Inserting a Document	db.COLLECTION_NAME.insert(document)	db.test_collection.insert({ key1: 'value', key2: 43, key3: [13.5, 12.6, 17.8] })
Find all the documents of a collection	db.COLLECTION_NAME.find()	db.test_collection.find() or db.test_collection.find().pretty()
Find all by exact key/value (key == key, val == val)	db.COLLECTION_NAME.find({ <key>: <value> })	db.test_collection.find({ key1: 'my search value' })

Find all by less-than, less-than & equals, greater-than, greater-than & equals:
$lt, $lte, $gt, $gte
	db.COLLECTION_NAME.find({
	<key>: {
	$lt: <value>
	}
	})
		db.test_collection.find({
		key1: {
		$lt: 20.00
		}
		})

Find all by not equal to	db.COLLECTION_NAME.find({
	<key>: {
	$ne: <value>
	}
	})
		db.test_collection.find({
		key1: {
		$ne: 'Bob'
		}
		})

Find all by a and b	db.COLLECTION_NAME.find({ $and: [{<key1>: <value>}, {<key2>: <value>}] })	db.test_collection.find({ $and: [{key1: 20}, {'key two': 'Bob'}] })
Find all by a or b	db.COLLECTION_NAME.find({ $or: [{<key1>: <value>}, {<key2>: <value>}] })	db.test_collection.find({ $or: [{key1: 20}, {'key two': 'Bob'}] })

Update first document found by criteria	db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA)	db.test_collection.update({
		{
		key1: 20
		}, {
		$set: {
		key1: 35
		}
		})

Update all documents found by criteria	db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA, OPTIONS)	db.test_collection.update({
		{
		key1: 20
		}, {
		$set: {
		key1: 35
		},
		{multi: true}
		})

Nuke & Pave a document:
The **save** method will completely save over a document
	db.COLLECTION_NAME.save({_id:ObjectID, NEW_DATA})	db.test_collection.save({
		{
		_id:
		ObjectId("5b141942e795d17231916472"),
		key1: 35,
		'key two': "bob",
		values: [1, 2, 3, 4]
		})

Deleting (Removing) all Documents that match critera	db.COLLECTION_NAME.remove(DELETION_CRITER)	db.test_collection.remove({
		{
		key1: 35
		})

Delete just one Document	db.COLLECTION_NAME.remove(DELETION_CRITER, 1)	db.test_collection.remove({
		{
		key1: 35
		}, 1)

Delete every Document in a collection	db.COLLECTION_NAME.remove(DELETION_CRITER, 1)	db.test_collection.remove()

If you want to delve deeper into the Mongo CLI, you can do so by exploring all the commands at your disposal here: https://www.tutorialspoint.com/mongodb/index.htm

Running Mongo CLI Commands

1. In one terminal window run mongod
2. In a second terminal window run mongo

```
// creates a uses a test database (immediately assigns it to the variable db)
use test

// create a collection called students and a new document (record) to it
db.students.insert({name: 'Shaun McKinnon', age: 39})

// bulk insert records
db.students.insert([{name: 'Arsh Patel', age: 22}, {name: 'Will Will', age: 23}])

// finds all records in our collection
db.students.find()

// pretty print our records
db.students.find().pretty()

// update a record
db.students.update({name: 'Arsh Patel'}, {$set: {dob: new Date("1996-01-01")}})

// delete a record
db.students.remove({name: 'Shaun McKinnon'})

// drop a collection (will remove all documents and drop the collection)
db.students.drop()

// drop a database (will drop all collections and then the db)
db.dropDatabase()
```

Just a note: The Mongo CLI is a working JavaScript console that you can use to enter raw JS commands in. You can even use these commands to stub records if you want:

```
for (let i = 0; i < 10; i++) {
  db.students.insert({student_id: i})
}
```

Author: Shaun McKinnon

BREAK — 00:15

Models and Mongoose — 00:40

What is Mongoose?

Mongoose is an ODM, which stands for **O**bject **D**ocument **M**apper. Basically the idea is to make writing validation, sanitization, business logic, and other boilerplate code simpler and faster. Mongoose will become our application's models, which will allow us to abstract our data logic from our view.

Adding Mongoose to Our App

In order to use Mongoose, you **MUST** have MongoDB and NodeJS installed. Otherwise, we add Mongoose the same way we do any package:

1. Navigate to the app we created in our Express lesson in your CLI
2. Run **npm install mongoose** to install the Mongoose package locally
3. Next, open your Express app in your IDE
4. Open **app.js**
    1. Add the line **var mongoose = require('mongoose');** after the **var app = express();** line (around line 11)
    2. Create a new directory called **config** at the root of the app
        1. Create a file called **connect.js**
        2. In the file, publish the following

```
module.exports = {
  // db type/server/database
  db: 'mongodb://localhost/comp2068'
}
```

    3. Back in the **app.js** file, add the following lines under the **var mongoose = require('mongoose');**

```
var mongoose = require( 'mongoose' );
var config = require( './config/connect' );
mongoose.connect( config.db );
```

5. In order to test, we'll first have to start mongod, if it isn't already running:
   Type **mongod** in your CLI
6. Next, start your app by running **npm start** in your CLI
    - You should see the following message

```
2018-06-03T14:52:01.996-0400 I NETWORK  [listener] connection accepted from 127.0.0.1:51067 #1 (1 connection now open)
2018-06-03T14:52:02.001-0400 I NETWORK  [conn1] received client metadata from 127.0.0.1:51067 conn1: { driver: { name: "nodejs", version: "3.0.8" }, os: { type: "Darwin", name: "darwin", architecture: "x64", version: "17.5.0" }, platform: "Node.js v9.9.0, LE, mongodb-core: 3.0.8" }
```

    - The above message tells us that we have received a connection from our application through the port 51067. It also tells us the platform our application is use, **Node.js**
7. If you navigate to **localhost:3000 (or perhaps 3001)** you will see your app
8. Congrats, you have connect the app to the database!

Creating a Model

MVC relies on separating data from presentation from logic. We use Models for our data, Views for our presentation, and Controllers for our logic. The benefit of this structure is it lends to better scalability, maintainability, and stability.

Models are our data piece, and they're great in that they encapsulate sanitization, validation, and mutation all in one area. The operations we perform on data should happen here, as the controller should simple facilitate the transfer of the data to an endpoint (perhaps the view). In our application, we're using Mongoose to help us create models and interact with our database:

1. Create a **models** directory at the root of our app
2. Create a file called **product.js** in the **/models** directory
    - Collections are plural, single documents are singular
    - This correlates to tables are plural, and records are singular
    - This is a common naming convention in MVC for models
    - Many MVC frameworks will have an inflection class that will deal with looking up singular and plural variations of your symbol names
    - The Mongo collection will be called **products**
3. In our **/routes** folder, create a new controller file called **products.js**
    - Express is a little different than standard MVCs where the routes folder contains controller files that handle the incoming route as well as the action that will be called

4. In our **/views** folder, create a new folder called /views/**products**
    1. In this folder, add a view called **index.js**
    2. Keeping our views in subfolders will make it easier to maintain naming conventions
5. Before we can access our controller we must register it with our application
    1. In the app.js file add the following lines:
       Under the line var usersRouter = require('./routes/users');
       Add: **var productsRouter = require('./routes/products');**
    2. This is our router/controller file that we want to include in our application
    3. Next, under the line app.use('/users', usersRouter);
       Add: **app.use('/products', productsRouter);**
    4. By this point, you should recognize when we're registering middleware with our application. Here we are registering the middleware **productsRouter** to our path **/products**. This will result in a user going to **/products** pointing to '**/**' in our router/controller file, products.js and triggering that action
6. Populate the pages we created with the following code

/models/product.js

```
var mongoose = require( 'mongoose' )

// all model classes will inherit from
// the mongoose.Schema class
var productSchema = new mongoose.Schema({
  name: {
    type: String,
    required: 'Please enter a product name.'
  },
  description: {
    type: String,
    required: 'Please enter a product description.'
  },
  price: {
    type: Number,
    required: 'Please enter an MSRP value.'
  }
})

// make this class public
module.exports = mongoose.model( 'Product', productSchema )
```

In the above file, we include the Mongoose module. We then build our schema which you will notice is a JSON structure. Each main key is an attribute/property/field/column of our collection/record. Each attribute has an object as a value that has various properties. **type** set's the data type of the object. **required** defines the attribute as being required in order to create it.

Next we export our model class so it is publicly available to our router/controller that we will define next.

/routes/products.js

```
var express = require('express')
var router = express.Router()

// create a link to our drink model
var Product = require( '../models/product' )

// our controller action for products
router.get('/', function(req, res, next) {
  // get our model data
  Product.find( function (err, products) {
    // if we get an error
    if ( err ) {
      console.log( err )
      res.render( 'error' )
    } else {
      // load the view
      res.render( 'products/index', {
        title: 'Our Products List',
        products: products
      })
    }
  })
})

// makes our file public to the application
module.exports = router;
```

Last week we looked at the routes/index.js file that was generated by the Express Generator. These files are our controller files. Using a route we can specify an action we want to occur. The controller action then pulls in the data from the model and delivers it to the view. We use a callback to define when want to render the view, because retrieving data from the model is an asynchronous process.

Once we get our data, we need to define what we'll do if there is an error. We have a default 404 error page (/views/error.ejs) that we can render if there is an error getting our data.

Otherwise, we send the products to the view as a localized variable. In the view we will iterate over the products array and display each product.

/views/products/index.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>

    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-WskhaSGFgHYWDcbwN70/dfYBj47jz9qbsMId/iRN3ewGhXQFZCSftd1LZCfmhktB"
crossorigin="anonymous">

    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>

  <body>
    <div class="container">
      <header>
        <div class="jumbotron">
          <h1><%= title %></h1>
        </div>
      </header>

      <section>
        <div class="row">

          <% for ( let product of products ) { %>
            <div class="col-sm-6">
              <div class="card">
                <div class="card-body">
                  <h5 class="card-title">
                    <%= product.name %>
                  </h5>
                  <p>
                    <%= product.description %>
                  </p>
                  <a href="#" class="btn btn-primary">
                    <%= product.price %>
                  </a>
                </div>
              </div>
            </div>
          <% } %>

        </div>
      </section>
    </div>

    <script src="https://code.jquery.com/jquery-3.3.1.min.js" integrity="sha256-
FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=" crossorigin="anonymous"></script>

    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.1/js/bootstrap.min.js"
integrity="sha384-smHYKdLADwkXOn1EmN1qk/HfnUcbVRZyYmZ4qpPea6sjB/pTJ0euyQp0Mk8ck+5T"
crossorigin="anonymous"></script>
  </body>
</html>
```

Here we've swapped out the **card** data with properties from our **products** array. A card will be generated for each product within our array, which we then can display to our user. Currently we're using the button to display the price. Next week we will create some more routes and use the button to take us to a single product view.

Author: Shaun McKinnon