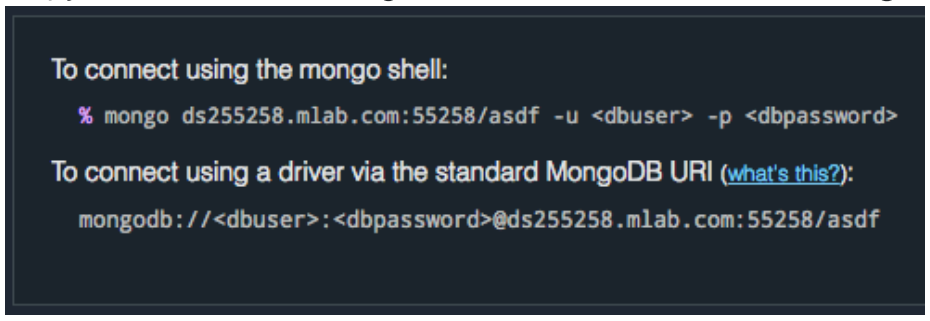
MongoDB/Mongoose Part 2

Setting up MLab — 00:20

1. Go to MLab and create an account (click the **GET STARTED INSTANTLY**) button
2. Go to your email and confirm
3. Login
4. Click on the **Create New** to create a new Mongo database
5. Choose **SANDBOX [FREE]**
6. Click **Continue**
7. Click **US East (Virginia) (us-east-1)** option
8. Click **Continue**
9. Type in **comp2068**
10. Click **Continue**
11. Click **Submit Order**
12. You will see a table with your database name listed in it: click on it
13. Copy the connection string (it's the line that starts with **mongodb://**)



14. Click on the tab **Users**
15. Click **Add database user**
16. Fill in the details to create a new user
17. The database string needs to be pasted into our **connect.js** file under the **/config** folder we created in our app

```
module.exports = {
  db: 'mongodb://my_user_name:my_password@ds153890.mlab.com:53890/comp2068'
}
```

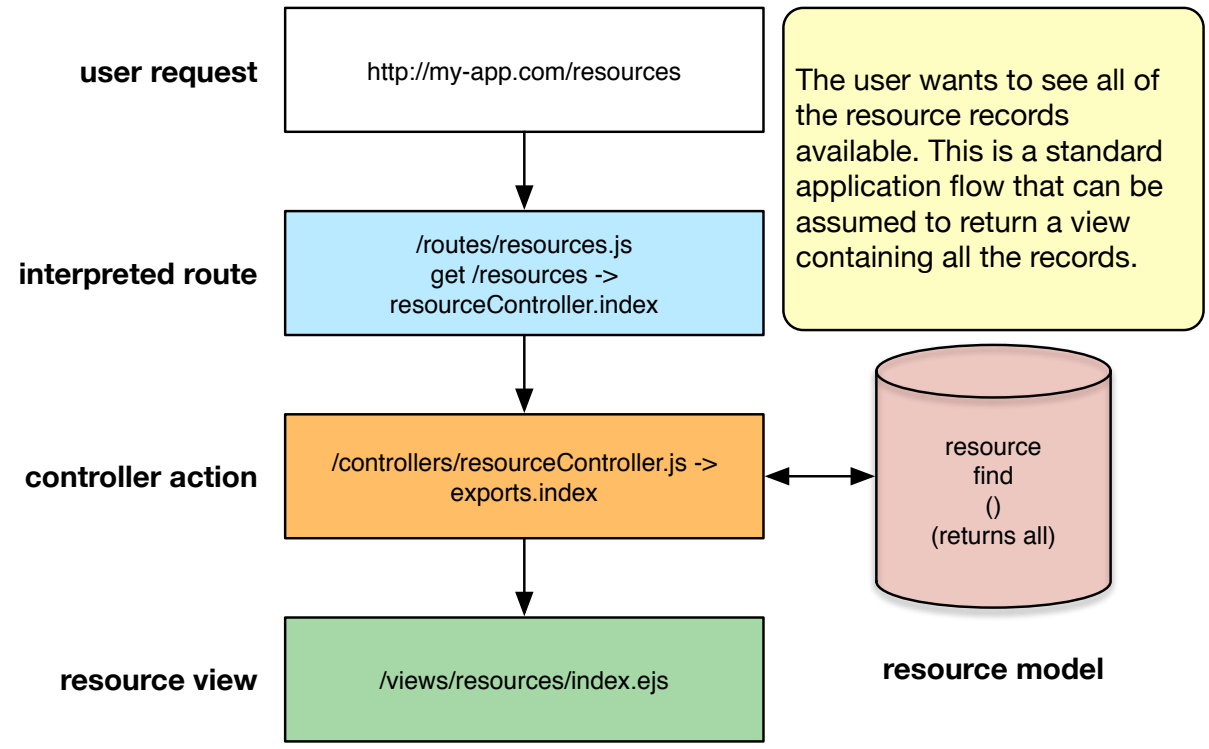
Author: Shaun McKinnon

Understanding the Express MVC App Flow — 00:20

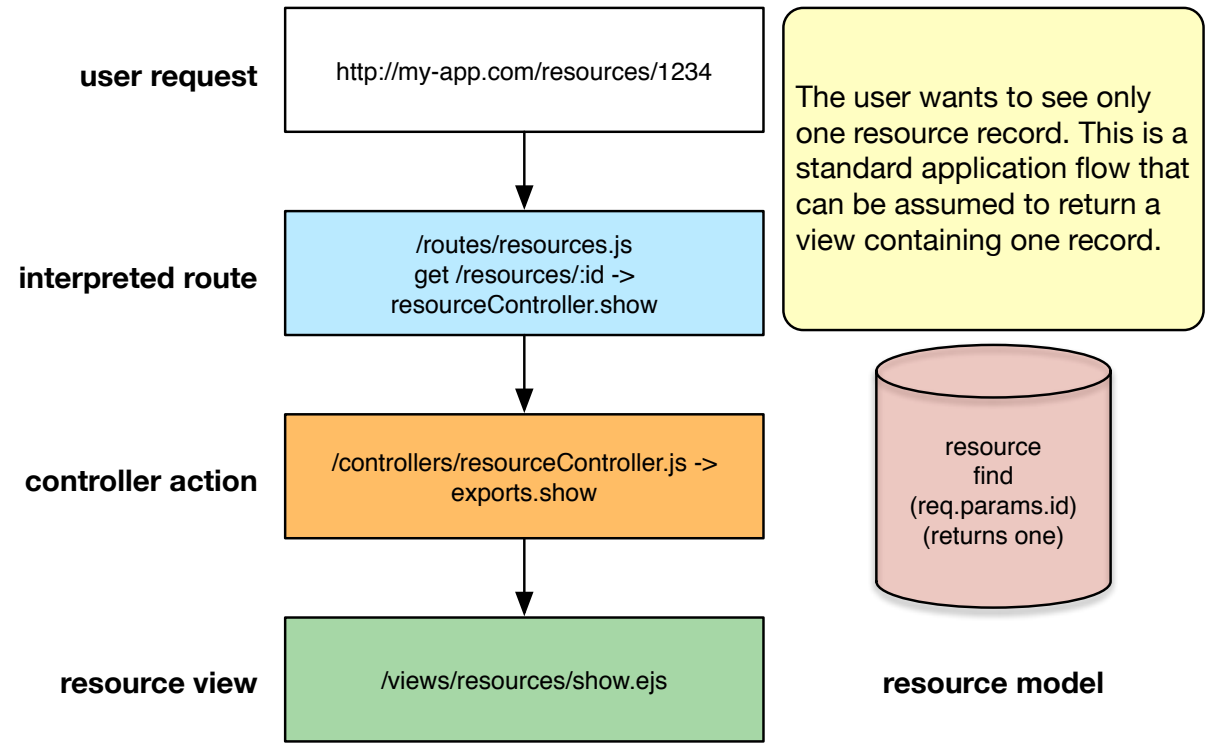
The Express (Resource Routes) Application Flow

Use Cases:

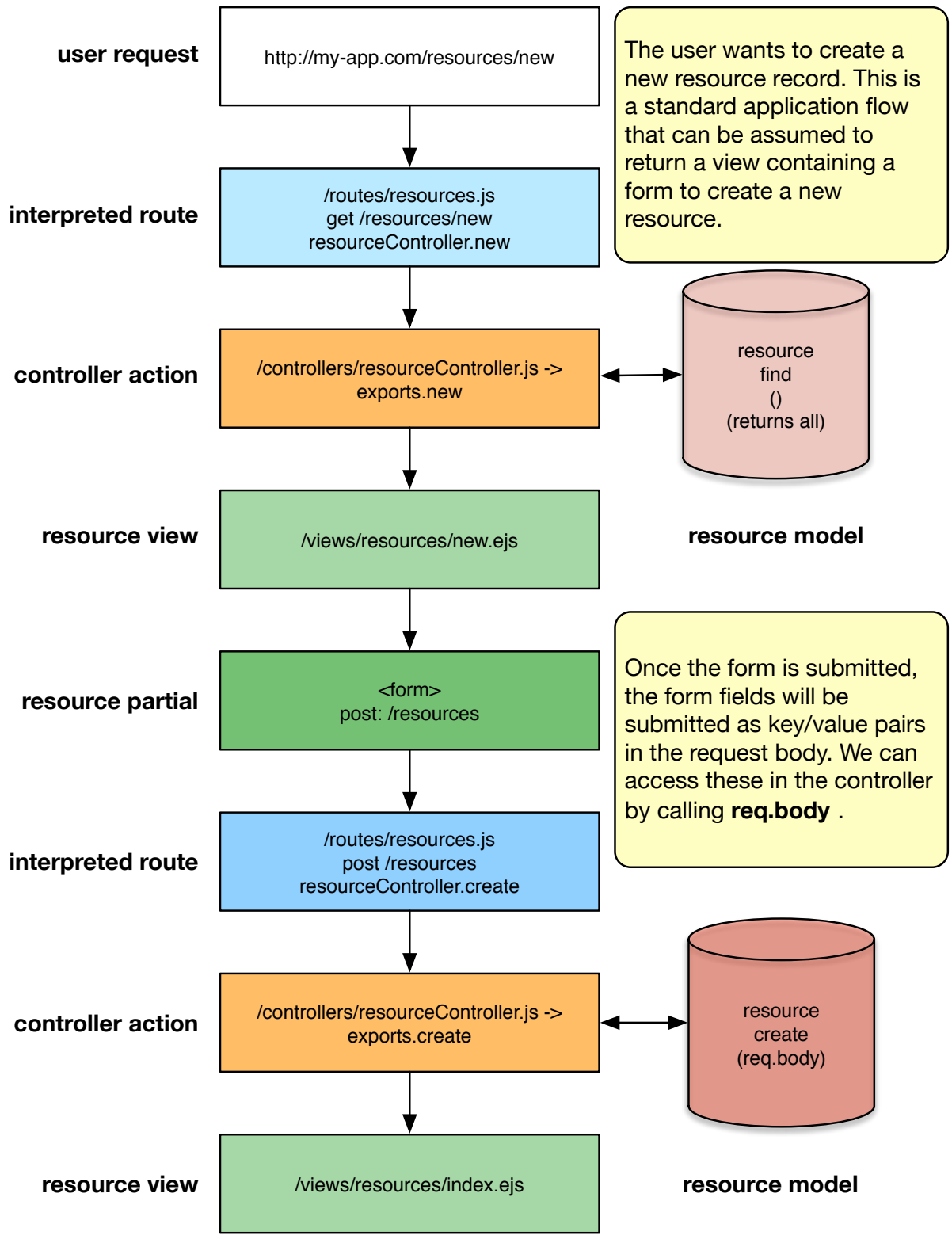
I want to see all records of a resource:



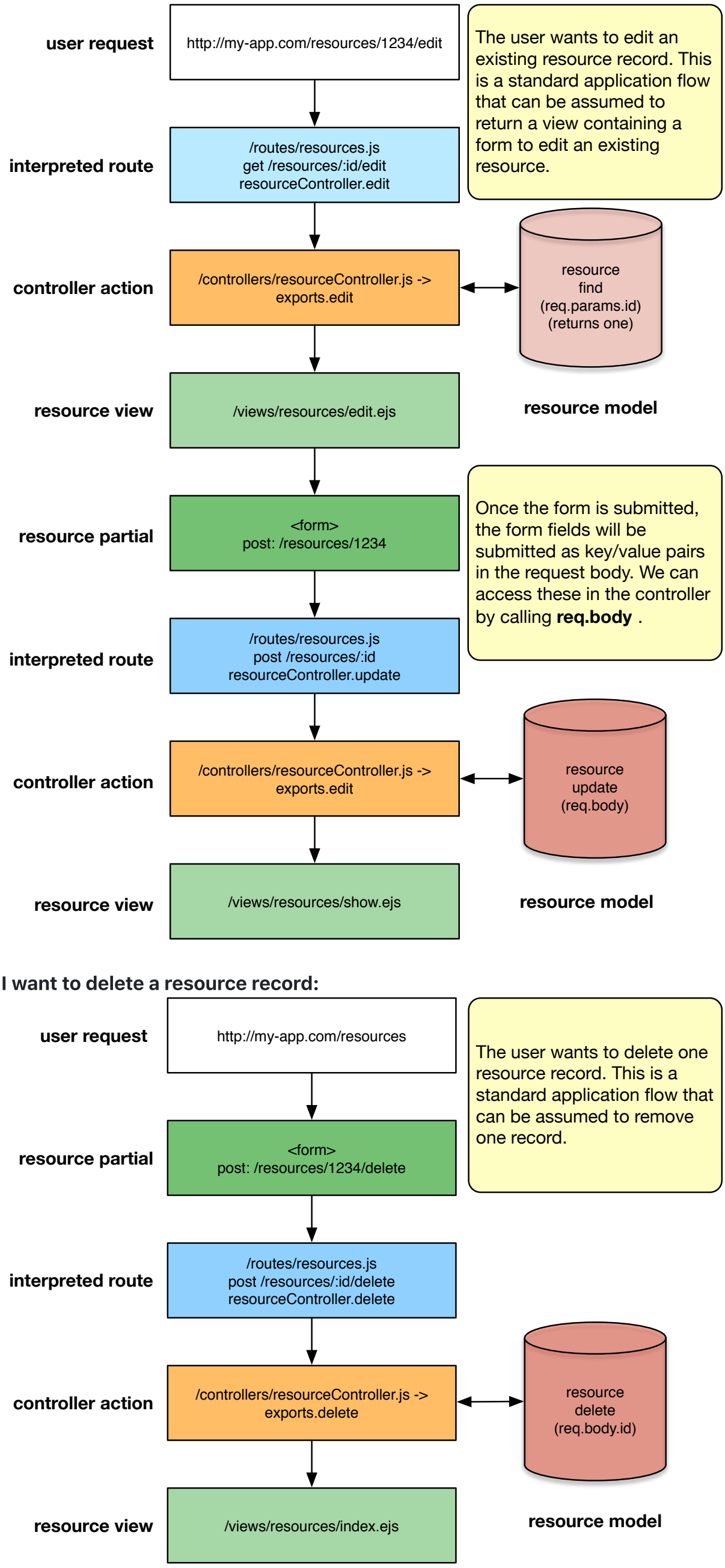
I want to see one resource record:



I want to create a new resource record:



I want to edit a resource record:



I want to delete a resource record:

Understanding Our Folder Structures

- **my-app**
  - **/models** *(our data **model** schemas go here)*
    - product.js *(our **product schema**)*
  - **/views** *(our resource **views** go here)*
    - **/products** *(we break each resource into its own folder)*

        - index.ejs *(this view will **show all** the **products** in the database)*
        - show.ejs *(this view will **show one product** chosen by the ID)*
        - new.ejs *(this view will **show a form** allowing for the user to **create a product**)*
        - edit.ejs *(this view will **show a form** allowing the user to **edit a product** by ID)*
- **/controllers** *(our resource **controllers** will go here)*
    - productsController.ejs *(this is our resource controller file that defines the actions each view will need)*
        - index *(this method will **fetch all** the products from the database and **render** the **index.ejs view** passing the products as a variable)*
        - show *(this method will **fetch one** product from the database and **render** the **show.ejs view** passing the product as a variable)*
        - new *(this method will **render** the new.ejs view)*
        - edit *(this method will **fetch one** product from the database and **render** the **edit.ejs view** passing the product as a variable)*
        - create *(this method will **create** a **new** resource using the req.body passed **from the new form**)*
        - update *(this method will **update** an **existing** resource using the req.body passed **from the edit form**)*
        - delete *(this method will **remove** an **existing** resource using the req.body.id passed **from the delete form**)*
- **/routes** *(our resource **routes** will go here)*
    - products.js *(this is the defined routes for our resource)*
        - **get** /products -> productsController.index *(this route will pass the **request** body to the **index** method in the **productsController**)*
        - **get** /products/:id -> productsController.show *(this route will pass the **request** body to the **show** method in the **productsController**)*
        - **get** /products/new -> productsController.new *(this route will pass the **request** body to the **new** method in the **productsController**)*
        - **get** /products/edit -> productsController.edit *(this route will pass the **request** body to the **edit** method in the **productsController**)*
        - **post** /products -> productsController.create *(this route will pass the **request** body to the **create** method in the **productsController**)*
        - **post** /products/:id -> productsController.update *(this route will pass the **request** body to the **update** method in the **productsController**)*
        - **post** /products/:id/delete -> productsController.delete *(this route will pass the **request** body to the **delete** method in the **productsController**)*

Author: Shaun McKinnon

BREAK — 00:15

Mongoose, Routes, Models, Controllers, and Views, OH MY! — 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://dbusername:dbpassword@ds153890.mlab.com:53890/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. Next, start your app by running **nodemon** (if it's installed) or **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**
6. If you navigate to **localhost:3000 (or perhaps 3001)** you will see your app
7. 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 router 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. We're going to separate this into its own controller
4. Add a new folder in the root called **/controllers**
   1. Add a new file called **productsController.js** to the **/controllers** directory
5. 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
6. Before we can access our route 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
7. 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.

/controllers/productsController.js

```
var Product = require( '../models/product' )

// our index function
exports.index = function( req, res ) {
  // create our locals parameter
  locals = {
    title: 'Products List'
  }

  Product.find().then( function (products) {
    // add the products to our locals
    locals.products = products

    // render our view
    res.render( 'products/index', locals )
  })
}
```

Controllers are responsible for passing our model data to our views. Here we're including the model that the **productsController** will work with: **Product**. Our **exports.index** function will create a locals object (these will be variables we want available to our view), find all the products from our Product model, then render our products/index view. You will see we're using promises instead of callbacks to handle the operation after the model as returned the data.

/routes/products.js

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

// create a link to our drink model
var productsController = require('../controllers/productsController')

router.get( '/', productsController.index )

// 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 contained our routes and controller actions. In the above file, we're splitting up the logic between 2 files instead. In order to make this work, we must

include our productsController file. We listen for the user's request. The above router will listen to requests for **/products/**. The route will then call the method **index** on the **productsController**. This convention is common amongst most MVC frameworks.

/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

Partials — 00:10

Why?

Partials are crucial to maintaining a modular layout. Much of the layout we will have will have a the same header and footer details. Partials allow us to separate the header and footer from the always changing content.

Partials are a big part of MVC frameworks. Occasionally they're called templates. It is important that regardless of the framework you're working with, you know where to find partials, and then make use of them.

Implementation

Add the following to each partial file:

/views/partials/header.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>
```

/views/partials/footer.ejs

```
    </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>
```

Add the partials to our existing files

Now that we have some partials, we can go replace the repeating blocks in each of our views. First, open **/views/index.ejs**. Change the file so it looks like below:

```
<% include partials/header %>

<article>
  <div class="row">

    <% for ( let card of cards ) { %>
      <div class="col-sm-6">
        <div class="card">
          <div class="card-body">
            <h5 class="card-title">
              <%= card.cardTitle %>
            </h5>
            <p>
              <%= card.cardBody %>
            </p>
            <a href="<%= card.cardLink.linkHREF %>" class="btn btn-primary">
              <%= card.cardLink.linkLabel %>
            </a>
          </div>
        </div>
      </div>
    <% } %>

  </div>
</article>

<% include partials/footer %>
```

Notice the **<% include ../partials/footer %>** syntax. These lines include a file relative to the **/views** directory.

We have one other view to fix. Open **/views/products/index.ejs** and change it to look like so:

```
<% include ../partials/header %>

<section>
  <div>
    <a href="/products/new">New Product</a>
  </div>

  <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>

<% include ../partials/footer %>
```

As we identify reusable pieces, we'll create more partial files. Partials are a great way to clean up our code.

Author: Shaun McKinnon

BREAK — 00:15

Routes & CRUD — 00:30

Resource Routing

Method	User Request	Intercepting Route	Controller Action
GET	/products/	/products	index
GET	/products/12345	/products/:id	show
GET	/products/new	/products/new	new
GET	/products/12345/edit	/products/:id/edit	edit
POST	/products/	/products/	create
POST	/products/12345	/products/:id/edit	update
POST	/products/12345/delete	/products/:id/delete	delete

Resource routing is a common convention for defining routes. Many MVC frameworks provide a way for setting a bunch of a routes at once by simply defining what the resource is. We are going to mimic resource routing by following the common conventions above.

Order of operations with routing is important. It is advisable to define your routes in this order: **index, new, show, edit, create, update, delete**
If you place **show** before **new**, the /products/new route will meet the /products/:id expression. Best to avoid that.

New/Create

The view below will allow us to collect information and use that to create our new product. Notice the action: this will be a **post** route that will need to be defined in our routes. The method will tell the router which route version we want to use. It is important to ensure you set the method to **post** or we will be sending our form to our **/index** route.

/views/products/new.ejs

```
<% include ../partials/header %>

<section>
  <form action="/products" method="post">
    <fieldset>
      <legend>Product Information</legend>

      <div class="form-group">
        <label for="name">Name</label>
        <input type="text" class="form-control" id="product_name" name="name" placeholder="Creepy Leeches"
required>
      </div>

      <div class="form-group">
        <label for="description">Description</label>
        <input type="text" class="form-control" id="product_description" name="description"
placeholder="Greasy, black, vampires of the deep." required>
      </div>

      <div class="form-group">
        <label for="price">Price</label>
        <div class="input-group">
          <div class="input-group-prepend">
            <div class="input-group-text">$</div>
          </div>

          <input type="number" class="form-control" id="product_price" name="price" placeholder="4.99"
step="0.01" min="0" required>
        </div>
      </div>
    </fieldset>

    <div class="form-group">
      <button type="submit" class="btn btn-primary">Create Product</button>
    </div>
  </form>
</section>

<% include ../partials/footer %>
```

In the above I am taking advantage of the HTML5 type **number** and the validation helpers **step** and **min**.

We will require 2 methods defined in our controller: The first will be for the view, the second for the actual creation of the new document in Mongo. Our view is a simple render of the file we created above, along with some locals. I have made locals a variable that contains an object. this will clean up our code and improve maintainability.

/controllers/productsController.ejs

```
// New
exports.new = function ( req, res ) {
  // locals
  locals = {
    title: 'New Product'
  }

  res.render( 'products/new', locals )
}

// Create
exports.create = function ( req, res, next ) {
  Product.create({
    name: req.body.name,
    description: req.body.description,
    price: req.body.price
  })
  .then( function () {
    res.redirect( '/products' )
  })
  .catch( function ( err ) {
    next( err )
  })
}
```

In the above code, you will notice that we first **create** our product. Mongoose uses promises to wrap these methods. This means we have the handy **then()** and **catch()** methods available to us. **then()** will execute regardless if our code resolves or rejects. However, if our code rejects, **catch()** will be called first. We can use this order of operations to define a success and error pattern. If our product is successfully created, we will redirect to our **/products** index route. Otherwise, we will utilize the **next** object to call Express' internal error handler and respond with an error page and some helpful information.

Last we will require 2 routes to direct traffic to the **new** and **create** actions that we've defined in our controller. Notice that **post** is pointing to **/** which is the same as our index action. This is a common resource route convention. Hence the importance of not messing up the verbs get and post.

/routes/products.js

```
// new
router.get( '/new', productsController.new )

// create
router.post( '/', productsController.create )
```

Once you have finished, create 4 new products using the form at /products/new. We will use these products in our upcoming actions.

Show

/views/products/show.ejs

```
<% include ../partials/header %>

<section>
  <div>
    <a href="/products/new">New Product</a>
  </div>

  <div class="row">

    <div class="col-sm-6">
      <div class="card">
          <div class="card-body">
            <h5 class="card-title">
              <%= product.name %>
            </h5>

            <p>
              <%= product.description %>
            </p>

            <p>
              <%= product.price %>
            </p>

            <div class="actions">
              <a href="/products/<%= product.id %>" class="btn btn-primary">
                Show
              </a>

              <a href="/products/<%= product.id %>/edit" class="btn btn-primary">
                Edit
              </a>

              <form action="/products/<%= product.id %>/delete" method="post"  onsubmit="return
confirm('Are you sure you want to delete this product?');">
                <input type="hidden" name="id" value="<%= product.id %>">
                <button type="submit" class="btn btn-danger">DELETE</button>
              </form>
            </div>
          </div>
        </div>
    </div>

  </div>
</section>

<% include ../partials/footer %>
```

Show will show only one product. Usually this would be a more detailed view of that product.

Our show controller is pretty bare bones, but points out one gigantic gotchya compared to most MVCs. Usually **params** would hold our **get** and **post** request bodies. This is where we would find our query strings. However, Express is intelligent in forcing the convention that these things are different, and a **get** request shouldn't have a request body. Therefore, when getting query strings, follow this convention:
get = req.params.my_attribute
post = req.body.my_attribute

/controllers/productsController.ejs

```
// Show
exports.show = function ( req, res ) {
  // locals
  locals = {
    title: 'Product'
  }

  Product.findById({
    _id: req.params.id
  })
  .then( function ( product ) {
    // add the products to our locals
    locals.product = product

    // render our view
    res.render( 'products/show', locals )
  })
  .catch( function ( err ) {
    next( err )
  })
}
```

/routes/products.js

```
// show
router.get( '/:id', productsController.show )
```

Edit/Update

Edit is the same as **New**, however, for a better UX, we pass in the product the user is editing so they can see what values the current attributes hold.

/views/products/edit.ejs

```
<% include ../partials/header %>

<section>
  <form action="/products/<%= product.id %>/edit" method="post">
    <fieldset>
      <legend>Product Information</legend>

      <input type="hidden" value="<%= product.id %>" name="id">

      <div class="form-group">
        <label for="name">Name</label>
        <input type="text" class="form-control" id="product_name" name="name" placeholder="Creepy Leeches"
required value="<%= product.name %>">
      </div>

      <div class="form-group">
        <label for="description">Description</label>
        <input type="text" class="form-control" id="product_description" name="description"
placeholder="Greasy, black, vampires of the deep." required value="<%= product.description %>">
      </div>

      <div class="form-group">
        <label for="price">Price</label>
        <div class="input-group">
          <div class="input-group-prepend">
            <div class="input-group-text">$</div>
          </div>

          <input type="number" class="form-control" id="product_price" name="price" placeholder="4.99"
step="0.01" min="0" required value="<%= product.price %>">
        </div>
      </div>
    </fieldset>

    <div class="form-group">
      <button type="submit" class="btn btn-primary">Update Product</button>
    </div>
  </form>
</section>

<% include ../partials/footer %>
```

In **exports.edit** we find the document and return it to our **edit** view as a local variable. This is similar to what we did in **index**. In the **exports.update** action, we find our product then change its parameters. Once we're finished we save it. Express has an **update** method similar to the **create** method we used in **new**. However, it doesn't fire the validations we've set in our model, resulting in now validation check. This means we would need to perform the validation checks in our controller, which is a gross, wrong, bad, terrible, shit idea. By utilizing the way we have, we essentially force validation to occur, as validation will always happen on **save()**.

/controllers/productsController.ejs

```
// Edit
exports.edit = function ( req, res, next ) {
  // locals
  locals = {
    title: 'Edit Product'
  }

  Product.findById({
    _id: req.params.id
  })
  .then( function ( product ) {
    // add the products to our locals
    locals.product = product

    // render our view
    res.render( 'products/edit', locals )
  })
  .catch( function ( err ) {
    next( err )
  })
}

// Update
exports.update = function ( req, res, next ) {
  Product.findById( req.params.id )
  .then(function ( product ) {
    product.name = req.body.name
    product.description = req.body.description
    product.price = req.body.price

    product.save()
    .then(  function () {
      res.redirect( '/products' )
    })
    .catch( function ( err ) {
      next( err )
    })
  })
  .catch(function ( err ) {
    next( err )
  })
}
```

/routes/products.js

```
// edit
router.get( '/:id/edit', productsController.edit )

// update
router.post( '/:id/edit', productsController.update )
```

Delete

Very easy. Just a simple pass of the id and we have deleted the record.

/controllers/productsController.ejs

```
// Delete
exports.delete = function ( req, res ) {
  Product.remove({
    _id: req.body.id
  })
  .then( function () {
    res.redirect( '/products' )
  })
  .catch( function ( err ) {
    next( err )
  })
}
```

/routes/products.js

```
// delete
router.post( '/:id', productsController.delete )
```

Updating the Products Index Page

Because we have made so many changes, we want to update our **/products/index** page so it has some of the new links in the cards:

```
<% include ../partials/header %>

<section>
  <div>
    <a href="/products/new">New Product</a>
  </div>

  <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>

            <p>
              <%= product.price %>
            </p>

            <div class="actions">
              <a href="/products/<%= product.id %>" class="btn btn-primary">
                Show
              </a>

              <a href="/products/<%= product.id %>/edit" class="btn btn-primary">
                Edit
              </a>

              <form action="/products/<%= product.id %>/delete" method="post"  onsubmit="return
confirm('Are you sure you want to delete this product?');">
                <input type="hidden" name="id" value="<%= product.id %>">
                <button type="submit" class="btn btn-danger">DELETE</button>
              </form>
            </div>
          </div>
        </div>
      </div>
    <% } %>

  </div>
</section>

<% include ../partials/footer %>
```

Small Change to SCSS

Just copy and paste:

```
html {
  box-sizing: border-box;
}
*, *:before, *:after {
  box-sizing: inherit;
}

$background: #ecf0f1;
$button-colour: #e74c3c;

body {
  background-color: $background;

  .container {
    background-color: #fff;
    padding: 1em;
    margin-top: 1em;
    border-radius: 1em;

    .card {
      margin-bottom: 1em;

      .btn {
        background-color: $button-colour;
        border: $button-colour;
      }

      .actions {
        form {
          display: inline;
        }
      }
    }
  }
}
```

Author: Shaun McKinnon