imjching committed with moose-horizons add npm install (#28) Latest commit bc22f4c a day ago

..

seed	Add README for homazon	a day ago
README.md	add npm install (#28)	a day ago

README.md

Homazon: Your one-stop home shop

Over the past couple of weeks you've already built Facebook and Yelp. Let's notch another win today by building Amazon. Okay, we won't be building *everything* at Amazon.com--maybe we'll leave Amazon Web Services for next week--but you will be building a kickass e-commerce site that allows users to browse products and shop from their mobile phone. You'll process payment information and manage inventory.

Introduction

There are two related but distinct parts to this project: the user view and the shopkeeper (admin) view (this part is bonus). If you are ever building projects in the real world, you'll see this pattern often and will need to build out the user/consumer part as well as the admin dashboard. So this is great practice! The user view, which will be a *mobile* web application, is the shopping interface that the user sees when they login. The shopkeeper view allows the shopkeeper to view and manage orders and inventory. The views are completely distinct, but they share and operate upon the same data model.

Overview

User interface diagrams

We're letting you go free here - this is going to be the first time you will make your own judgment calls on how to actually architect your application. Fear not, though... we are here to help.

Let's think through how exactly we want the "user flow" to look.

Core features:

Authentication

- Register / Create an account
- Login

Shop

- View a list of products (catalog)
- View details for a single product: title, photo (one or more), description
- View your shopping cart
- Checkout process for an order

Ok - let's get started.

Part 1. Express scaffolding

In the past we have given you most of the Express scaffolding you have needed. This time you will generate your own scaffolding using express-generator.

Go ahead and install express-generator globally with `npm install express-generator -g`. Now navigate to `/week05/day1`.

Use the `express homazon --view=hbs` command (from express-generator) to create a scaffolding (the view=hbs flag adds handlebars support). This will create an Express app named homazon in the current working directory.

> Note: Make sure that you are in your `day1` folder before typing `express homazon --view=hbs`. You should get a message stating `destination is not empty, continue? [y/N]`. Type y to continue.

Remember to then `cd` into the app folder before continuing. Next, type `npm install` to install the dependencies that came together with the express-generator.

Voila - your application is pretty much done for you! Jk. LOL! Where's the fun in that?

Previously we built all of our routes first, and then all of our models, etc. In the real world, this is not necessarily how we go about building our app. Instead, we look at the user flow (discussed above) and build the application iteratively and incrementally, feature by feature. For every step below, think about the models, model methods, and routes that are required.

Express gives you your `views` and `routes` folder by default. We recommend adding a `models` folder as well.

Make sure that you are testing your code as you go.

Checkpoint

At these checkpoints be sure to test your code by running `npm start`. If your code fails to work be sure to spend some time debugging before continuing on.

Part 2. ES6 Support

In the latest version of `Node.js` most ES6 syntax is supported, but there are still a few features that are not yet implemented ie: `import` and `export`.

To get support for these we can use a javascript compiler like Babel. Babel will translate our code to node-compatible javascript before it is run. To get all the packages we need run this command:

```
npm install --save-dev babel-cli babel-preset-es2015
```

Now that we have babel installed we need to make sure our start script uses it. Go ahead and replace our current start script in `package.json` with this:

```
babel-node ./bin/www --presets es2015
```

To ensure everything is working correctly lets replace each call to `require()` with `import` and each call to `module.exports` with `export default`

```
// Below are 2 examples of what you should change
// In app.js

// ...
import index from './routes/index';
// ...

// In index.js

import express from 'express';
// ...
export default router;
```

Checkpoint

Run your app and watch as the amazing babel compiles our code right before our eyes! You should be able to run your application without recieving any syntax errors. Try viewing it in the browser and you'll be greeted by the default express landing page.

Be sure to use `import` and `export` for the remainder of this project

If you would like to use `nodemon` click here for more detailed instructions.

Part 3. Authentication (sign up and login)

You know the drill by now.

First, create a `User` schema. We want our user to be able to login using a username/password combo. Here is the skeleton for your schema:

```
    username: String,
    password: String,
```

Remember to require the proper Mongoose modules in your model files.

Next, use Passport to create authentication routes.

Remember how to do Passport and authentication perfectly with no errors? Ok good... then you won't need these helpful hints.

To help jog your memory, here are some useful snippets for making Passport work!

- Hint

```
import express from 'express';
import session from 'express-session';
import passport from 'passport';
import LocalStrategey from 'passport-local';

app.use(session({
  secret: process.env.SECRET,
  store: new MongoStore({mongooseConnection: mongoose.connection})
}));

app.use(passport.initialize());
app.use(passport.session());

passport.serializeUser((user, done) => {
  done(null, user._id);
});

passport.deserializeUser((id, done) => {
  User.findById(id, (err, user) => {
    done(err, user);
  });
});
```

Lets exercise our ES6 skills by using Promises instead of callback functions in mongoose.

Mongoose async operations, like `.save()` and queries, return Promises. By default these are not the same Promises included with ES6. Fortunately for us we can change mongoose's Promise library with only 1 line! Pretty convenient! Go ahead and add this line after your mongoose import in `app.js` :

```
mongoose.Promise = global.Promise;
```

Now we can use ES6 Promises with mongoose like this:

```
user.save().then((doc) => {
  // handle response
});
```

To perform queries with promises we need to use `.exec()`:

```
user.find().exec().then((docs) => {
  // handle response
});
```

Be sure to use Promises for mongoose throughout the remainder of this project

Implementing auth with Passport requires a few steps:

1. Require the necessary Passport modules (reference past projects to help you)
2. Create a LocalStrategy with Passport
3. Implement a route to GET the register (sign up) page
4. Implement a route to POST registration data to
5. Implement a route to GET the login page
6. Implement a route to POST login data to
7. Implement a route to GET the logout page

Checkpoint

When you're done with Part 3 you should be able to Register a user and login. See

Part 4. Products and Product List

So now we have logged in let's buy stuff!

Lets think about this in terms of your model and your routes. At the most basic level our store consists of a bunch of products. We'll make it more complex later, but for now, we take baby steps. Let's create our `Product` schema. A product will have a title, a description, and an image. For now, we'll pretend that the product is free.

```
  title: String,
  description: String,
  imageUri: String
```

Model done. Time for routes and views.

We want to be able to navigate between 2 main views: a view to see the entire list of products (just titles) and a detailed product view that shows the title, description, and image for a single product.

```
router.get('/', (req, res, next) => {
  // Insert code to look up all products
  // and show all products on a single page
});

router.get('/product/:pid', (req, res, next) => {
  // Insert code to look up all a single product by its id
  // and show it on the page
});
```

Create the relevant routes and views.

Checkpoint

At this point you should be able to render the landing page. Next lets add some `Products`!

Note, you won't be able to see any products because we are not writing any routes to "create" new products... yet. Instead, we are going to seed our database with existing product data from the `products.json` file.

You will need to create your own script to seed the database.

- Hint

```
import products from '../../seed/product.json'
var productPromises = products.map((product) => (new Product(product).save()));
Promise.all(productPromises)
  .then(() => (console.log('Success. Created products!'))
  .catch(err) => (console.log('Error', err))
```

Checkpoint

Now you should be able to list out all of the products defined in `products.json` see. If you navigate to a specific `products` id you should be able to see a detailed view of that product see.

Part 5. Shopping Cart

Ok, so we have logged in and now we can view products! Sweet.

Our next step is to implement a shopping cart that "saves" any products that you want to buy. Create the shopping cart as an array that lives inside `req.session` . Be careful with how you initialize this: make sure it's initialized (as an empty array) before you put the first item in it.

We don't need the cart to be in a model; rather, we'll just store the cart as part of our `req.session` and update it as needed. Let's think through all the actions we need for a shopping cart

1. Get a current cart
2. Add an item to a cart
3. Delete an item in a cart
4. Empty the entire cart

Let's think about the routes that correspond to each of these actions.

```
router.get('/cart', (req, res, next) => {
  // Render a new page with our cart
}

router.post('/cart/add/:pid', (req, res, next) => {
  // Insert code that takes a product id (pid), finds that product
  // and inserts it into the cart array. Remember, we want to insert
  // the entire object into the array...not just the pid.
}

router.delete('/cart/delete/:pid', (req, res, next) => {
  // Insert code that takes a product id (pid), finds that product
  // and removes it from the cart array. Remember that you need to use
  // the .equals method to compare Mongoose ObjectIDs.
}

router.delete('/cart/delete', (req, res, next) => {
  // Empty the cart array
});
```

Checkpoint

After completing there routes be sure to test their functionality. Try using `console.log()` to print the cart at the starting and ending point of each route.

Now time to figure out our views. We want to do a few things here:

1. Add an "Add to cart" button to the detailed product view that adds the product to the cart.
2. We want to create the view for the actual cart itself (i.e. once I add something to my cart, I want to be taken to a page that shows my entire shopping cart). This corresponds to your first GET request for `/cart`. On this page, the user should be able to see a list of items (title and description) in the shopping cart, with the option to view each product, delete individual products from the cart, or clear the cart completely.
3. On the main product list page (at the bottom) we also want to see the number of items in the cart. Display a message that says "Your shopping cart contains X items". In order to do this, you need to create a function that counts the items in the cart and renders that count in the view.

Checkpoint

You should now be able to log in, add your shipping info, be taken to a page with all of the products, click and view product details, add the product to your cart, and view/edit your cart. see

Boom, baby!

Part 6. Payment and Checkout

Payment info is a bit more complicated. After the user has successfully saved their shipping info, direct them to another form to enter payment info. You'll use an embedded Stripe form for this. Stripe does the hard work, collecting and validating the credit card info, and returning a token.

You were not alive in the pre-Stripe era. The pre-Stripe era was worse than the Ice Age. Death and destruction ruled the world. You needed a merchant account with large financial institutions in order to process credit card payments. Your application had to connect to a payment gateway, which connected to the larger financial network for authorizing transactions.

Stripe has taken all the tedium and duplicative effort inherent to processing payments and abstracted it all behind a single, developer-friendly API. Yes, holding cards is hard. Yes, there's a lot of regulation around processing payments. Whatever. Doesn't matter – that's Stripe's problem, not yours :). Oh, and Stripe supports 100+ currencies out of the box. In addition to credit and debit cards, Apple Pay, Android Pay, you can also easily support Bitcoin, Alipay, or Amex Express Checkout.

Get started with Stripe

1. Create a new Stripe account using the link above
2. Select "API" on the left side of the dashboard
3. You will need both the "Publishable key" and "Secret Key" in order to use the Stripe API so we should save them as environmental variables
4. We're going to use the simplest tool that Stripe offers: Checkout.

   Checkout takes care of building the payment form, validating user inputs, securing data, and sending it to Stripe without having it touch your server. Once you submit a Checkout form, you get back a token, which you save in order to actually create the charge.

   Carefully copy this `form` into your cart view and be sure to update each of the fields with your own values. Also be sure to add any `input` fields you may need above the script. The first one has been done for you.

```
<form action="/your-server-side-code" method="POST">
  <input type="text" name="name" placeholder="Name">
  <script
    src="https://checkout.stripe.com/checkout.js" class="stripe-button"
    data-key="<PUBLISHABLE_KEY>"
    data-amount="999"
    data-name="Demo Site"
    data-description="Widget"
    data-image="https://stripe.com/img/documentation/checkout/marketplace.png"
    data-locale="auto">
```

```
  </script>
</form>
```

The following data will be recieved when the form is submitted:

```
req.body.stripeToken
req.body.stripeEmail
// Any other data you passed into the form
```

5. Next it's time to install stripe in our application and set up an endpoint to access this data

```
npm install stripe --save
```

6. We can create a customer and a charge in Stripe using Promises like this.

```
import stripePackage from 'stripe';
const stripe = stripePackage('<SECRET_KEY>');

// More details in link above
stripe.customer.create;
```

Pay close attention to the flow of the linked code. First the customer is created and saved to the database then they are charged using the customer id that was just created. This will be important for the next step.

7. `stripeTokens` are used to access a customers information and can only be used once. With that in mind lets store the customer info in our own database so they won't have to re-enter their information the next time they make a purchase.

   Create a new model called `Payment`. A `Payment` should contain all the data we recieved from Stripe plus the _id of the current user. We can use this in the future to provide a list of payment methods to our user.

```
stripeBrand: String,
stripeCustomerId: String,
stripeExpMonth: Number,
stripeExpYear: Number,
stripeLast4: Number,
stripeSource: String,
status: Number,
// Any other data you passed into the form
_userid: mongoose.Schema.Types.ObjectId
```

8. Once you create a customer on Stripe's servers, create a new `payment` object on our end and save it to the database. You should add this functionallity to the code you wrote in step 7.

Checkpoint

At this point you should be able to create a customer and generate a payment document in mLab. Be sure to test your code thoroughly before continuing. see

With these final two pieces of information in place, we can complete the order flow. Add a button to your cart page that says "Purchase." For now, your flow should only do two things:

1. Create a new `Order` document containing:
   - timestamp
   - contents of the order (pass the entire objects again)
   - user that placed the order
   - the associated payment info
   - the associated shipping info
   - order status
   - the subtotal
   - the total

2. Display a page to the user thanking them for their order and detailing the order. It should look like this

In reality we'd want to do other things here such as charging the user's card, maybe generating a shipping label and calculating estimated ship time, etc.

Checkpoint

You've completed all the main functionality of Homazon! Congrats! Be sure to double check that all of your functionality works before continuing on to the bonus.

Bonus: Shopkeeper (admin) view

Core features:

- Admin Login
- Ability to view and manage product list and subproducts
- Ability to manage inventory per product
- Ability to view list of orders and update order status
- Ability to view list of users, update users (e.g., setting admin status for access to shopkeeper view)

Bonus Part 1. Authentication

Create an admin "role" on the User so that only certain users are allowed to access the shopkeeper view. Add a button to switch to shopkeeper view on the home (product list) page, only visible to these admin users.

The root path for logged-in admin users is different from the root path for non-admin users. Admins should go to an admin dashboard page where they can view all the orders!

Bonus Part 2. Product list

Display the entire list of products. Allow the admin to delete products. When the admin taps on a product, display a form that lets the user edit the product (title, description, image).

You will need routes for the following:

- GET all products
- POST to create a product
- DELETE to delete a product
- PUT to update a product

Do the same with the subproducts (and the admin should be able to change the stock).

Give some thought to the question of what happens if a product or subproduct are deleted or modified, but a user has 1. already placed the product in their shopping cart, or 2. already placed an order for this product. Refer to the concurrency section above--did you solve this problem already?

Bonus Part 3. View and manage orders

Allow the shopkeeper to view and edit existing orders, updating e.g. the order status.

Bonus Part 4. View user list

Allow the shopkeeper to view a list of users.