**Recipe6: Nav Bar**

In the Recipe4, you built out a basic user menu directive that loaded information about the currently logged-in user via HTTP request. In this recipe, you'll break ground on the MEAN Retail application using the concepts you've learned thus far. The first thing you'll do is adapt the user menu directive to look like this.

//user-menu.html



So now refer Recipe6 folder the code structure for this app and it looks like this.



The entry point for the client is this index.html file. That's the HTML that corresponds to this page. Now the index.html file is pretty standard. It pulls in angular.js and the compiled client side JavaScript.

//index.html

Now the index.html file is pretty standard. It pulls in angular.js and the compiled client side JavaScript.



The body contains only one tag, this user menu directive.

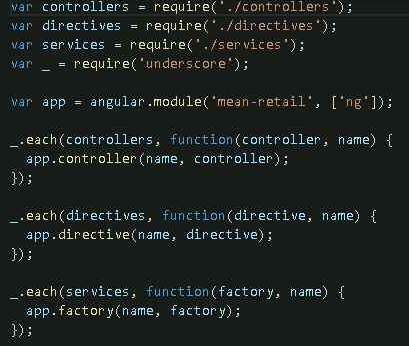
There are four client side JavaScript files. Index.js is the entry point for the application. And controllers.js and directives.js contain the app's controllers and directives, respectively. Then there's also services.js which contains the app services.

Browserify will be responsible for taking in index.js and compiling the bin/index.js file that the browser uses.

Here's what the index.js file looks like.

//index.js

All index.js is going to do is require in all the controllers, directives, and services. And build the mean-retail angular module.



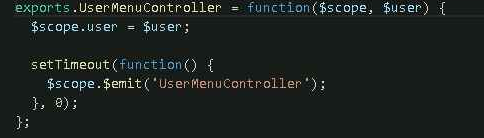
//directives.js

It combines this user menu controller with the user menu template.



//controllers.js

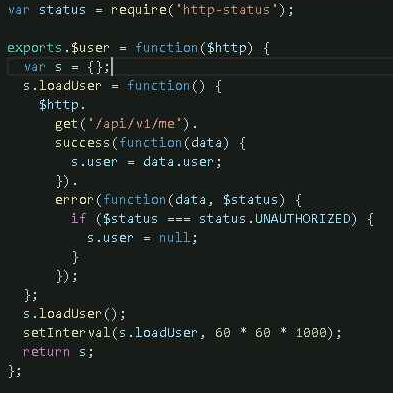
As you can see, it doesn't do terribly much other than defer to this $user service.



So let's take a look at services.js

//services.js

Here's the definition of the $user service



This service has one function, load user. It does an HTTP request to /api/v1/me to load data about the current user. It also, by default, re-does this request every hour.

Finally, gulp and browserify hold this all together.

//gulpfile.js



Once you click the Login button, you'll get logged in and get redirected back. And you'll see your Facebook profile picture pop-up here.



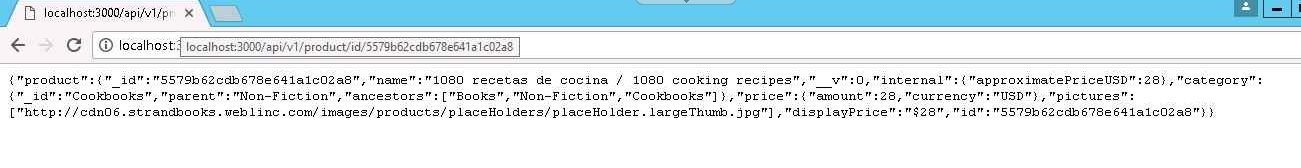


**Recipe7: Product view**

Now that you have a user menu, it's time to actually build out your first AngularJS view. The view you'll build is the product view, which is the canonical page for a particular product. The product view has two primary parts. It shows the category hierarchy for the product, and it shows detailed information related to the product.

This project has a small dataset of about 350 products, consisting of electronics and books. So in order to populate your MongoDB instance with this data, you will need to download this, and then you will need to run mongorestore. So once you run mongorestore, it will insert all the data from the dump directory.

The product view will need to load product data from the REST API. Now recall that your REST API has this/api/v1 /product /id /MongoDB ID route that enables you to load data about a product by its ID.



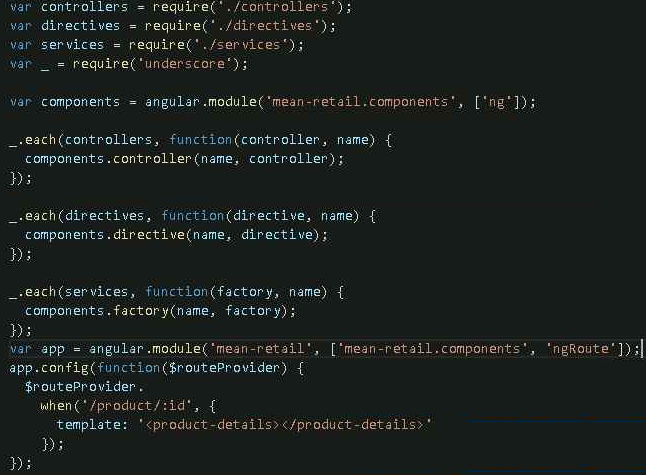
Your product view will just be a directive that loads this data and displays it in HTML format that looks like this.



So now refer Recipe7 folder the code structure for this app and it looks like this.



So first let's take a look at index.js, which is the main entry point for the JavaScript part of this application.



**Notice that you're going to have two separate modules now--**one that contains all the directives, controllers, and services, and one that defines the single page app routing.

Now notice that you have a new route, this /product route, which has a template that uses the product-details directive.

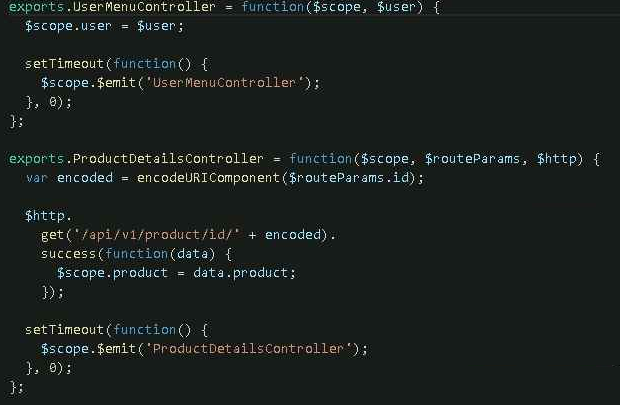
//directives.js

It's just a standard controller and template directive.



So let's take a look at ProductDetailsController.

//controllers.js



It sends an HTTP request to the REST API to get data about a product. So now this controller uses one AngularJS feature you haven't seen before. Note this $routeParams service. The routeParams service is a map from route parameters in the AngularJS URL to the route parameter values.

In AngularJS, this routeParams.id will contain the hex string of the product ID. So once again, the ProductDetailsController takes the product ID from the routeParams service, and makes an HTTP request for the product details. Once it has loaded the product details, you're going to expose the product details to the AngularJS scope.

//Product\_details.html

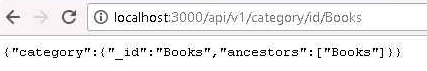
Now the HTML here is mostly cosmetic. Remember that there are two primary parts to the product view. There's the category hierarchy, which corresponds to this particular bit of the page. And then there's the actual product details, which are within this product-details-content div.



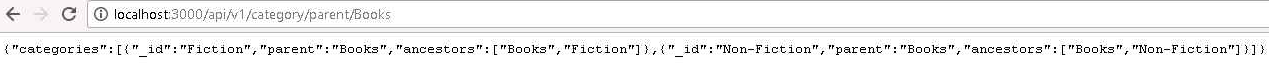
**Recipe8: Category view**

The view you'll build is the category view which will enable your user to browse products by category, as well as drill down into subcategories. In order to work with this data, you're going to need three API endpoints from the rest API session:

The first is the route for loading a category by ID.

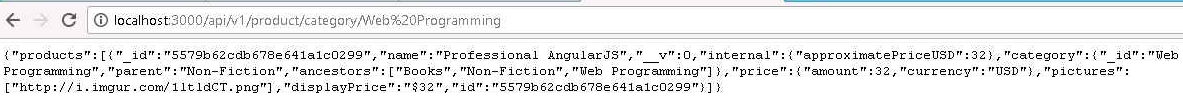


The second is the route for loading all categories by their parent field.



This will be useful for allowing users to drill down into child categories.

Now the third and final route that you're going to need is the route that loads all products that belong to a given category.



So now refer **Recipe8** folder the code structure for this app and it looks like this.



In index.js you’re going to have to add a new route.

//Index.js



The new route is this /category/:category route. Notice that this route has a template. Let's take a look at this category\_view.html template.

//category\_view.html



So as you can see, the category\_view.html template is pretty simple. It has these two directives. These two directives are this category tree directive and the category products directive.

Now the category tree directive corresponds to part of the view where the user can navigate the category tree. The category products directive is responsible for rendering this whole list of products.

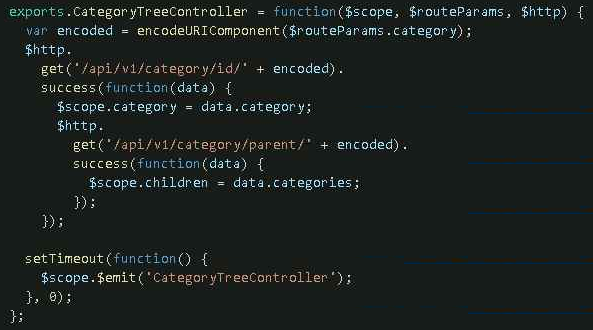
So now let's take a look at these two directives. First, let's take a look at the category tree directive

#directives.js



Let's take a look at the category tree controller first.

#controllers.js



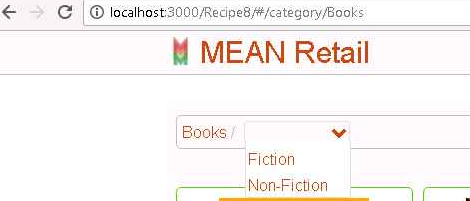
So this is the category tree controller. It's responsible for two simple tasks. It's going to load data from the category by ID route and then the category by parent’s route and expose both data points to the JavaScript scope.

So let's take a look at how the html uses this data. So this is templates/category\_tree.html,



The first part of this html should look pretty familiar. In particular, it's identical to the category navigation that you saw in the Product Details View Recipe.

The second part of the html, this child categories div, is new. So again, the new part corresponds to this dropdown.



So first, the child category dropdown only appears if there are children to show. And then there's an arrow that toggles the visibility state of this dropdown. Let's take a look at the category products directive.

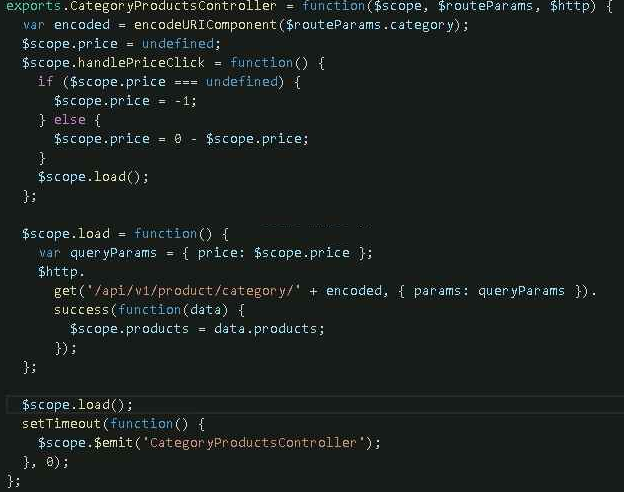
#directives.js



It also consists of just a template and a controller.

Let's take a look at the controller first.

#controllers.js

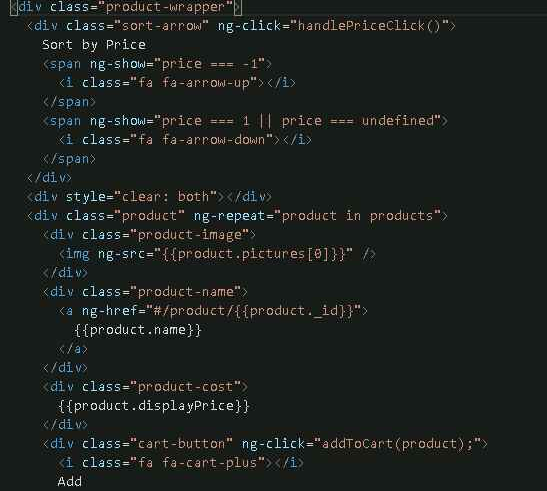


This controller defines two functions. The most important is this load function. So let's take a look at that first.

The load function is responsible for loading product data from the server. It queries the server for which products belong to the given category and then adds a special query parameter for sorting by price. And once the controller's done loading, it's going to query the server for data.

The other function is this handle price click function. This enables the user to toggle sorting by price. So now that you've seen how this controller works, let's take a look at the html.

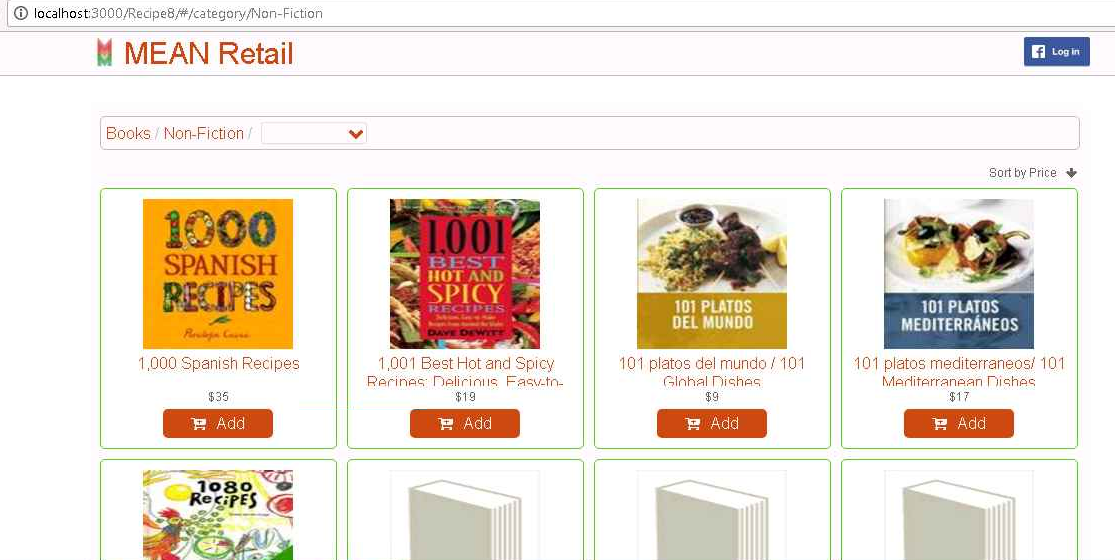
# category\_products.html



It has two distinct parts. The first thing we have is this price toggle or this sort arrow div. This handles sorting by price and shows some handy arrows to indicate the sort order to the user.

Now the second part of the directives html is the list of products represented by this product div. This part does an ng repeat over all the list of products. And for each product it's going to display the product's name with a link to the product detail view. And finally, it's going to display the price using the display price virtual that you learned about in the schema design session. And finally, there's a button that will enable you to add this product to your cart which you will implement when you implement the cart view.

Run this code and you will get below screen:



**Recipe9: Cart view**

So now that you've implemented the ability to browse through products by category, it's time to implement the ability to check out. So once you have a product in your cart, this view will allow you to enter in your credit card information, in this case a fake test credit card, and pay for your purchase.

**So this is how this looks in practice.**



And on check out click, you see, checked out successfully, and the cart is now empty.



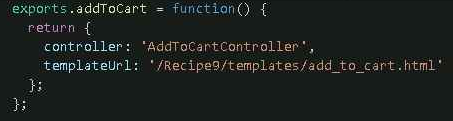
But first, in order to implement the checkout functionality, you first need to implement the ability to add a product to your cart. Thus far, you've only enabled users to browse products not actually add a product to their cart. However, in this example, the Add To Cart button actually does what it's supposed to do. So let's take a look at how this Add To Cart button is implemented.

So let's take a look at the HTML code for the product details view.



Notice that now, there's this add-to-cart directive that's replaced the old placeholder button. So here's the definition of the addToCart directive.

//directives.js



The add\_to\_cart.html file almost the same code that you saw in previous Recipe.

#add\_to\_cart.html



So when you click on this button, this addToCart method executes. And when the success variable is set, you get this nice, handy Added message that informs the user that the product was added to the cart successfully.

Let's take a look at the controller. So here's the AddToCart controller.

#controllers.js



This controller exposes one function, the addToCart function for modifying the user's cart. In this case, you're going to add the specified product with quantity one to the array, and then send a put request with the entire array over to the server.

So now that you have the ability to add a product to your cart, let's define the view an index.js that the users will use to checkout.

#index.js



This /checkout route and its corresponding checkout directive will contain all the logic for checking out.

So once again, here's the checkout directives definition.

#directives.js



**Let's take a look at the HTML, checkout.html first.**

#checkout.html

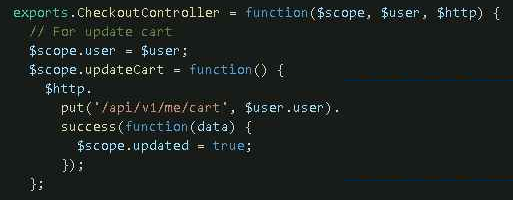
It's broken up into two major parts. The first part, this div with this ng-repeat and the class checkout product, is responsible for updating the cart, because you'd like your users to be able to update the item quantity before finally paying.



So again, the HTML does an ng-repeat over all items in the user's cart. Then it displays information about the product, like the picture, the name, and the price, and then exposes an input field that lets the user modify the quantity.

So once the user is happy with their updates, they can hit this update cart button, and that triggers this updateCart function.

Let's take a look at the JavaScript code corresponding to the update card HTML. So here's the checkout controller that corresponds to the checkout directive.



So in order to display the user's cart, all the JavaScript needs to do is pull down the $user service, and expose it on the scope. Similarly, to update the cart, all you need to do is make a put request to the correct API endpoint with the entire user object.

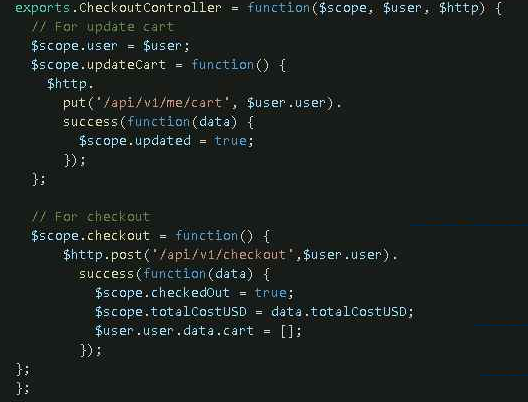
So now that you've implemented this part for updating your cart, let's take a look at how this credit card checkout functionality is implemented.

First, let's see what the HTML looks like in the checkout.html file.



In particular, there's this credit card number input. Then you also have a button that triggers the checkout function. Finally, there's these two little messages. The first one shows up when there's an error and the second one when the user has checked out successfully.

Let's take a look at the corresponding controller code.



Now the second half of checkout controller is responsible for posting data to the server.

Once you've succeeded, you could set this checked out property to true, and then your view will display a confirmation message to the user that looks like this down here.



**So congratulations.** You've implemented this MEAN retail shopping cart application.