**Mean Stack (REST API)**

So far in this course you've written Node.js code that interacts with MongoDB. The part of the stack that you've interacted with is the bottom part of the MEAN stack, the Node.js and the MongoDB part. The code you've written thus far is of limited use because you haven't addressed the key question of how Angular JS will interact with the Mongoose schemas that you've designed. At a high level, a REST API is an interface for client-side JavaScript running in, say, Google Chrome to read data from and store data to your server. REST, which stands for Representational State Transfer, is a paradigm for a browser to communicate with a server over HTTP, which is short for Hypertext Transfer Protocol, also known as that thing that loads Google's home page

The REST paradigm uses HTTP to manage state, using so-called CRUD operations, which is short for Create, Read, Update, and Delete.

**The HTTP verb GET corresponds to a read operation in the CRUD acronym.**

**An HTTP Request with verb POST corresponds to a create operation.**

**An HTTP Request with verb PUT corresponds to an update operation.**

**An HTTP Request with verb DELETE corresponds to a delete operation.**

The goal of the rest of this session is to build the REST API using Node.js on top of the Mongoose schemas you wrote in the Mongoose schema session. This REST API will expose endpoints that enable your client side JavaScript to create and load users, products, and categories.

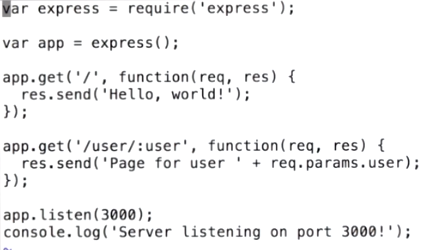
So let’s start with below recipes to understand it in better way

**Recipe1: Introduction to Express**

In this session, you will take your first step towards building a rest API and learn how to start an HTTP server in Node.js. The npm package you will be using to start an HTTP server is called Express.

It's the most popular Node.js web server and web application framework. Express is fast, flexible, and highly customizable. Express provides routing and other high-level abstractions that are indispensable for modern web development.

Create index.js file as below.



Starting an Express server is pretty easy. The Express module, which is accessed here via the require express call returns a function that you can use to create an Express app. You can then attach so-called routes to the app.

Routing is the term that describes telling Express which code to run for which types of HTTP requests. For instance, in this file, whenever somebody does a get on the slash route, Express will send back a page that contains the plain text "Hello, world."

Well, when you run node index.js, you should see the following output-- server listening on port 3000.

Now that the server is running, you should be able to see this output in a browser, so let's pull up Chrome and see what happens.

C:\Users\462439\Desktop\1.PNG

So when I go to Google Chrome and go to http://localhost:3000, I see this nice friendly "Hello, world" message, which is the result of this particular route handler.

Note that this file includes another route, this slash user route. The colon in this route delimits a so-called route parameter. For instance, if I navigate to /user/ MongoDB, I see the text page for user MongoDB with option empty.



So what's this option thing all about? Well, let's take a look at server.js see how server.js uses it. So in server.js, notice that we also use this req.query.option. This req.query object contains key value pairs representing the URL's query string.

The query string is effectively everything that comes after the question mark in the URL. For instance, if we go to /user/MongoDB and attach question mark option equals test, the req.query object will contain a key option with value string test.



**Recipe2: Dependency injection in NodeJS + ExpressJS**

Dependency injection is a software engineering practice that helps you break your code up into small, easy to maintain, chunks. The general idea of dependency injection is to separate initialization code from business logic, so your rest API route handlers never have to worry about say, setting up Mongoose models.

Open index.js and bad.js file from RestApi folder.

//bad.js



//index.js



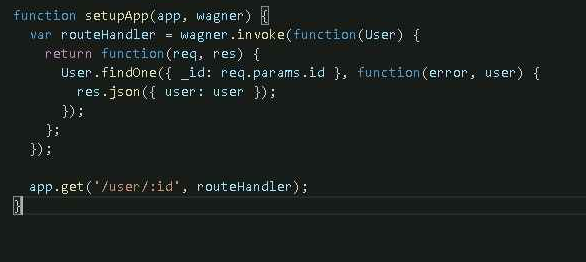
For instance, in this example, instead of using the Mongoose model directly, this business logic function takes the Mongoose model as a parameter. Now why access the Mongoose model in this way? First of all, since the function's dependencies are all specified as parameters, it's easy to refactor this whole function out into a separate file.

Now dependency injection may sound good theory, but what about when you want to add parameters to your functions. Suppose you wanted to have the myUserFunction depend on another Mongoose model. Now every place that calls myUserFunction needs to add a new parameter to the function to call.

This is why typically you'll use a framework to handle dependency injection for you. In this recipe, you'll be using and npm package called Wagner, like the famous German opera composer. Wagner has a rich feature set that goes well beyond dependency injection but for the purposes of this course you'll be using Wagner as a dependency injector. Wagner lets you register named factories which are functions that return values. These values are known as services.

Open wagner.js from RestApi folder:



In this case, you register a service called User which corresponds to your user model. Now that you've registered the user model as a service, how do you access it? Wagner has this handy function called invoke.

Wagner's invoke function behaves much like AngularJS's invoke function. It takes a function and executes it. However, invoke also inspects the function's parameter list and pulls in services that match the parameter names. The function specified in this call to invoke takes a single parameter called user. Wagner looks for a service named User and calls the function with the correct service.

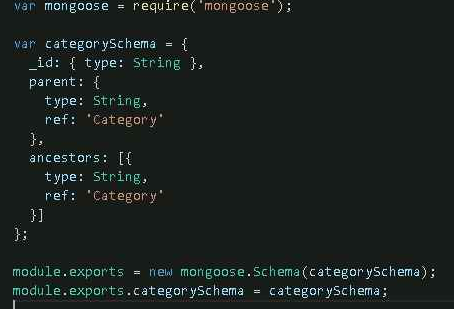
Since you're calling invoke here, the route handler variable is actually an Express route handler function, which you can then pass to the app.get function.

**Recipe3: Category API**

This API will provide two endpoints, one to load categories by their underscore ID and one to load categories by their parent category.

In this example, there are five files. The first file we'll take a look at is the category.js file. This category.js file is the same category schema that you saw in the schema design session.

//category.js



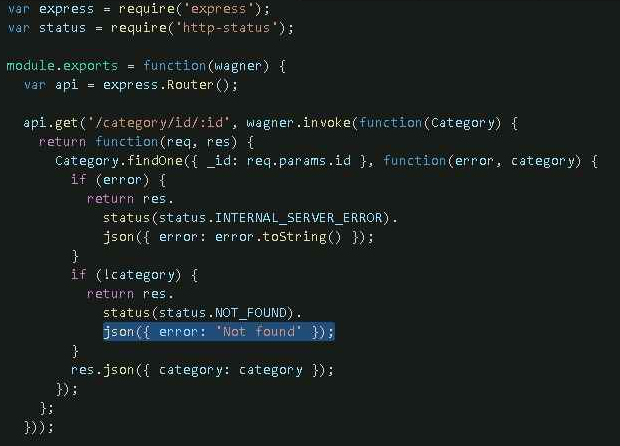
Next, let's take a look at the package.json file. The package.json file contains seven dependencies. You have seen all of these dependencies in previous lessons except for this http-status package. Http-status has a map of readable strings to HTTP status codes. So instead of writing 404, you just write status.notfound.

//Package.json



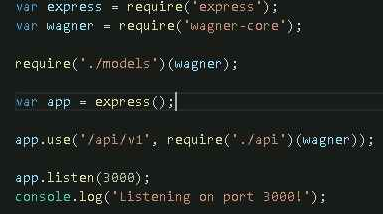
Now let's take a look at this api.js file. As you can see, this is where the http-status module is used. And instead of putting say, 404, we use status.notfound, which is the same thing as 404.

//api.js



Now, before we really dive into the api.js file, let's take a look at index.js first. The index.js file is the entry point for running your web application. To bootstrap the application, it uses the api.js file, and then starts an HTTP server on port 3000.

//index.js



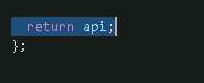
Let's take a look at this models.js file, which the index.js file uses to bootstrap your Mongoose models. So the models.js file handles everything you need to set up Mongoose and the category model. In particular, it connects to a MongoDB, creates a Mongoose model by including the schema, and registers the category service with Wagner.

//models.js

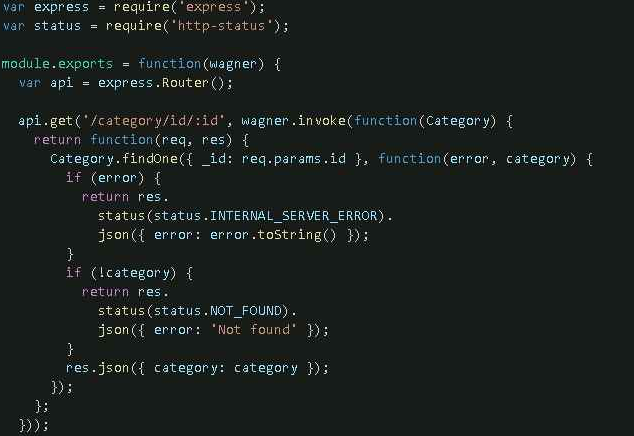


So here's the api.js file. It's the largest file you've seen so far, but don't worry, there's not very many new concepts here. First thing you're going to do is create a new Express router. And then at the end of the file, you're going to return the Express router so higher level apps can include the router using app.use.





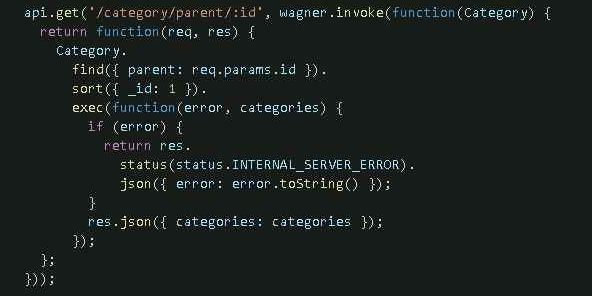
There's two routes in this router. This first route loads categories by their underscore ID field.



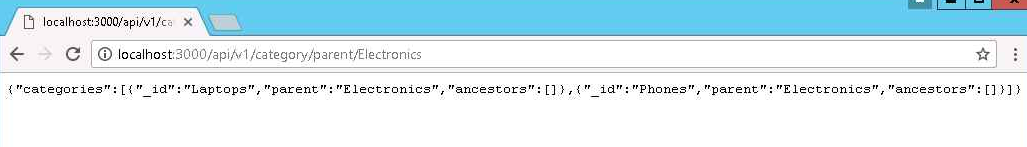
For instance, if I pull up a browser and go to localhost:3000/a pi/v1/category/id/electronics, I'm going to get the details for the electronics category.



And this is the route that's going to load categories by their parent category.



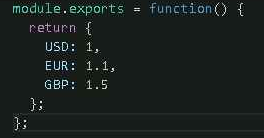
For instance, once again, visiting /api/v1/category/parent/electronics will load all categories whose parent category is Electronics.



**Recipe4: Product API**

There are two new files in this recipe. There's product.js file and the fx.js file. Let's take a look at fx.js first. The fx.js file contains the same fake exchange rates that you saw in the Mongoose schema design session.

//fx.js



The product.js file contains the product schema that you defined in the Mongoose schema design session.

//product.js



Now the models.js file has changed a bit because now there's two models instead of one. So change your model.js file as below.

//model.js



The new product model was created here. The code that registers the services has changed a little so you don't have to write a separate wagner.factory call for every model. This uses the each function from the \_package that you saw in the introductory chapter. This ensures that our code stays dry. D-R-Y stands for Don't Repeat Yourself.

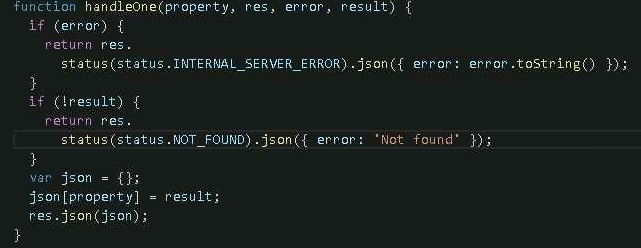
Let's take a look at the api.js file which is going to define the route handlers for the two API endpoints for this recipe. There are two get routes here. The first one loads a product by its ID and the second one loads its product by its ancestor category.

//api.js



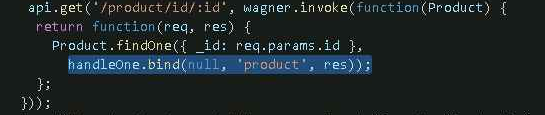
The route that loads a product by its ID is pretty trivial. The only major difference between this and the route that loads a category by its ID is this handleOne.bind thing that you're using as a callback.

Now, the bind function is a powerful JavaScript tool for reusing code. Let's take a look at the handleOne function. Here's this handleOne function.

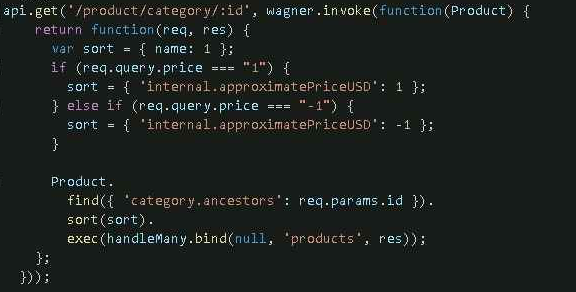


Notice that takes four parameters: a property name, an express response object, and then an error and a result. These error and result fields come from a findOne call.

When you call handleOne.bind with these three parameters, the first parameter is null variable in handleOne, which you don't use in this example. The second and third parameters, product and the Express response object become these first two parameters to handleOne.



Now let's take a look at the second API endpoint. The second API endpoint returns all products that belong to a given category, including its subcategories.



This API endpoint has one additional caveat. The user can specify if they want the products sorted by price. For instance, when you navigate to /api/v1/products/categories/electronics, you will get back the products ordered by name. But if you specify price equals 1 in the query string like /api/v1/products/categories/electronics?price=1, you get the products ordered by lowest price first and if the price is negative 1, it will use descending.

**Now let's take a look at the code that tests this API.**

The test for the route that loads products by ID is pretty analogous to the category API test.

db.products.insert({

\_id: '000000000000000000000001',

name: 'LG G4',

price: {

amount: 300,

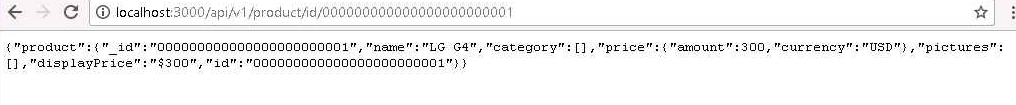
currency: 'USD'

}

});

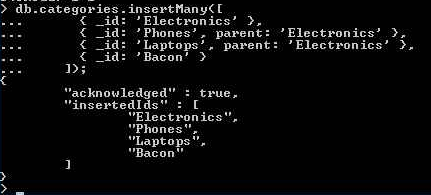
First, you create a product, which is the LG G4. And run your api as nodemon index.js and access url:

<http://localhost:3000/api/v1/product/id/000000000000000000000001>



The test for the route that loads products by category is more complex, because it needs to create a bunch of data.

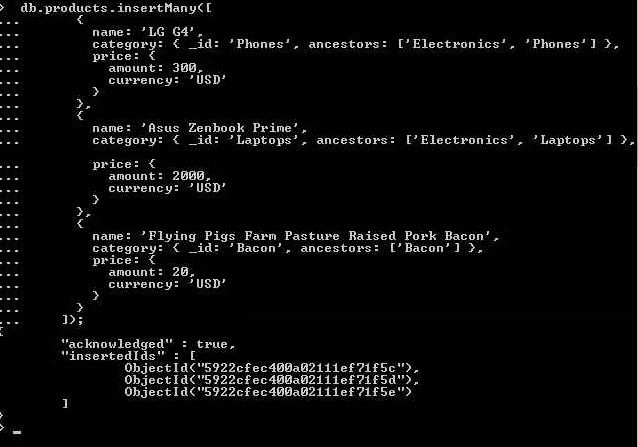
First create four categories-- Electronics, Phones, Laptops, and Bacon.



You're also going to create three products-- the LG G4, which is in category Phones;

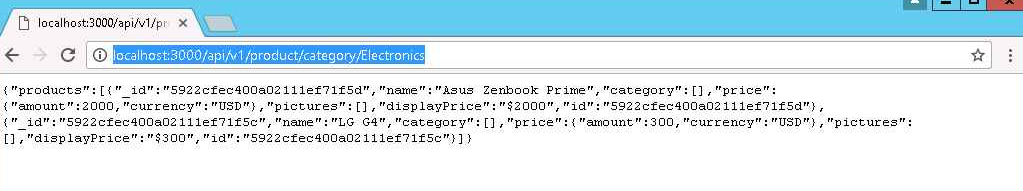
The Asus Zenbook Prime, which is in category Laptops;

And finally, some Bacon, which is in category Bacon.



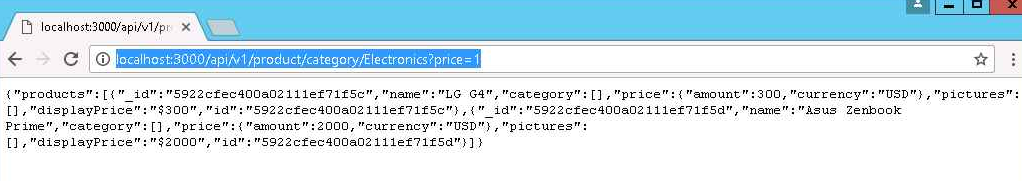
Then make an HTTP request to product category Electronics

<http://localhost:3000/api/v1/product/category/Electronics>



Finally, you're going to test that when you specify price is 1, you’re going to get the products sorted in ascending order by price.

<http://localhost:3000/api/v1/product/category/Electronics?price=1>



**Recipe5: User and Cart APIs**

The final pieces of the API will revolve around individual users, including logging in, adding a product to the user's carts, and checking out.Once you complete the rest of the session, you will be able to log in with your Facebook account, add a product to your cart, and do a fake payment through Stripe, which is a simple online payment processing service.In this recipe, you will focus on building two routes, one that shows the currently logged in user's cart, and one that allows the user to modify their cart.

Add user.js file contains the user schema that you defined in the Mongoose schema design session.

//user.js



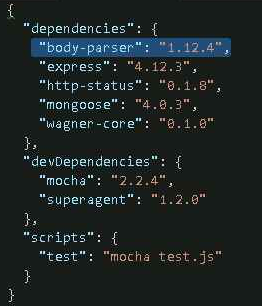
Now the models.js file has changed a bit because now there's one more model. So change your model.js file as below.

//model.js

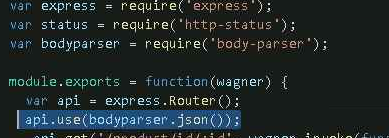


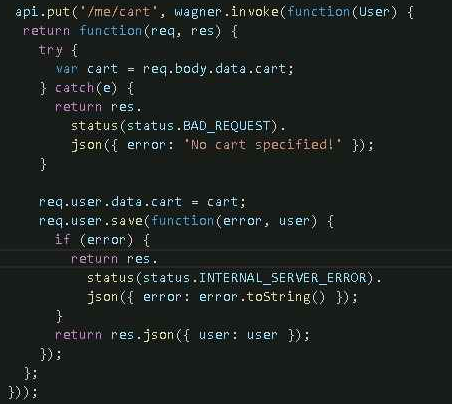
In this recipe, you will use one new package, body-parser package. This package parses http request bodies.

//package.json



The real changes are going to be in the api.js file. Add body-parser in api.js file.



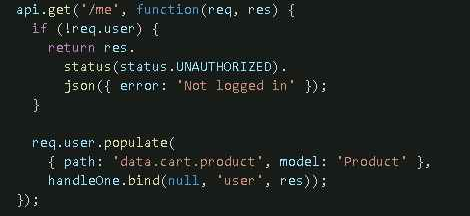
For instance, let's take a look at the route handler for modifying the user's cart.

The cart takes this data.cart field from the request body, which the body parser middleware makes available. If there isn't a data.cart field, the handler will return a bad request error status. Once this route handler has this data.cart field from the http request body, it sets the logged in users cart to the provided value and saves the user.

Well, there are two interesting assumptions underlying this code. First, this req.user property. In the Facebook login recipe, you will define middleware similar to the body parse of middleware that will handle setting the req.user variable to the currently logged in user.

Second you may be concerned about data integrity when overriding the entire cart array with user provided data. Of course, Mongoose handles casting invalidation under the hood. So if a malicious user tries to corrupt their own data by providing an invalid object ID or a negative quantity, the Mongoose save will fail. This is why Mongoose is such an integral part of MEAN stack web development. Validating data is typically a pain, but Mongoose enables you to declare a schema once, and then save documents without explicitly validating them.

The second route you'll write is a route that loads the currently logged in user's cart. Here's the route Handler for getting the currently logged in user's data, including their cart.



Mongoose has this very handy populate function, that superficially behaves like an SQL join operation. Once you have this user document, when you call populate on the data.cart.product, path, Mongoose will replace the product ID with the corresponding product from the database. It will actually execute a query to get all of the products pointed to in the user's cart.

**Recipe6: Integrating Facebook OAuth**

In the user API recipe, you set up a basic rest API for modifying and loading a user's cart. However, you avoided some key questions about how you will enable users to actually register and log in to your site. Namely, how does a user register? How does user log in? And how do you track which user is currently logged in?

To enable Facebook authentication, you're going to add three new packages.

//package.json

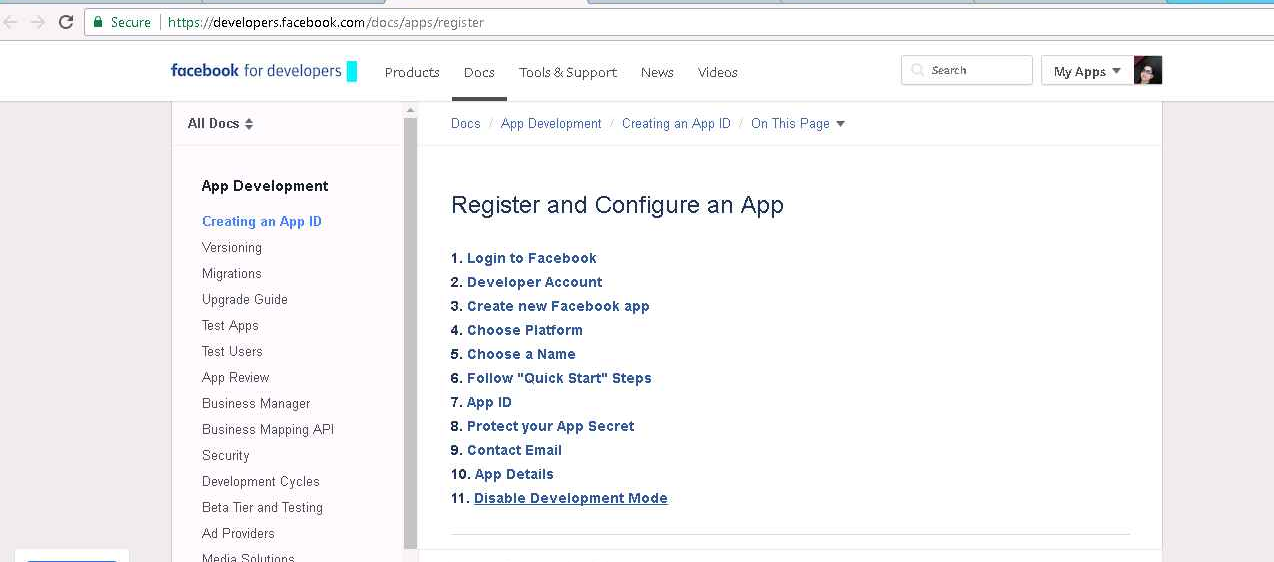


**express-session:** This package is similar to the express body parser package you saw in the user API recipe. It defines an express middleware. This particular middleware handles session storage or identifying a logged in user based on a cookie. This package is the easiest way to introduce session storage into your express app.

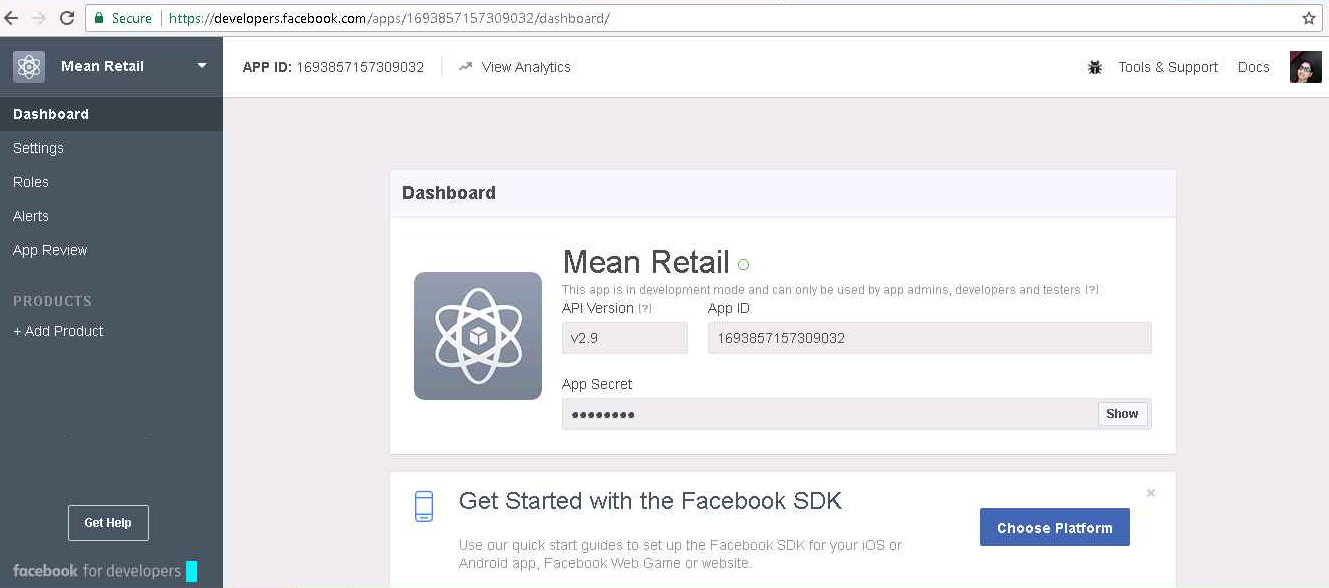
**Passport:** Passport is the most popular authentication framework for node.js. However, by default, passport only supports conventional username and password authentication.

**passport-facebook:** The passport Facebook module enables passport to handle Facebook login.

But first, before you integrate with Facebook login, need to go to developer.facebook.com and create a new Facebook app. Create App ID, Next, type in a name and choose a category, and then make sure not to touch any other parameters.

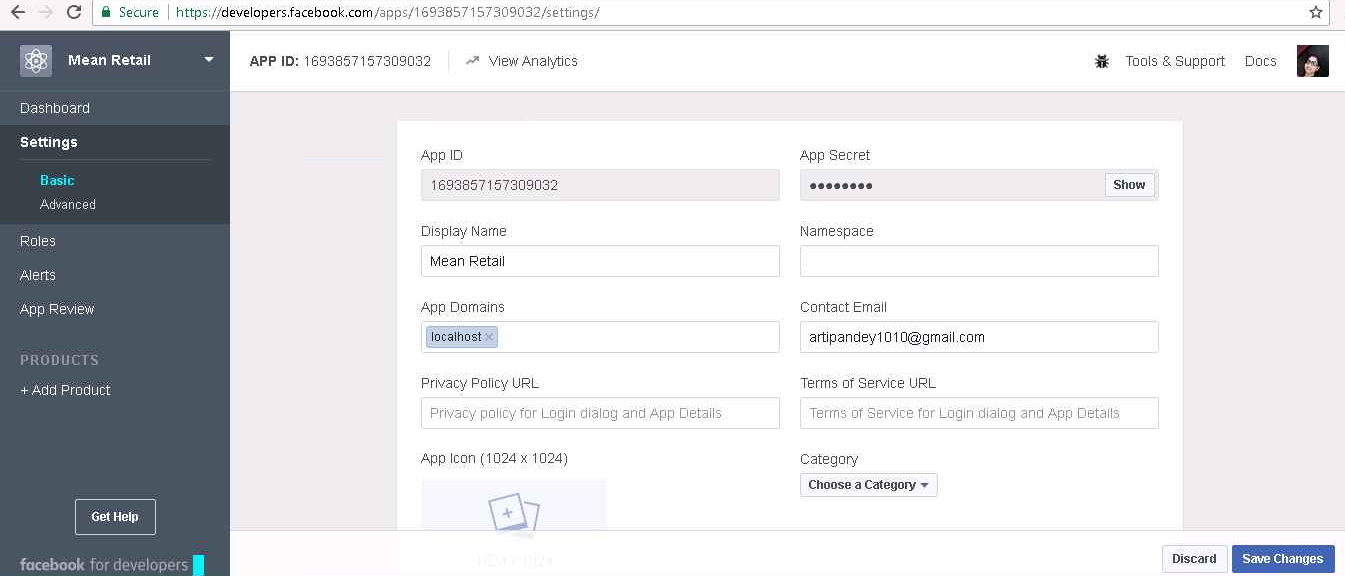


Once you have a Facebook application, you should see a screen that looks like this.



Note the App ID and the App Secret. You will need both of these to allow users to log in with Facebook.

To finish configuring your application, you need to go to the Settings tab. You're going to have to add localhost. To your App Domain section.

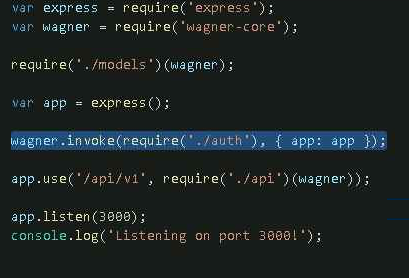


And then you're going to click Add Platform. Choose Website. And enter in localhost:3000 into the site URL. Then don't forget to hit Save.



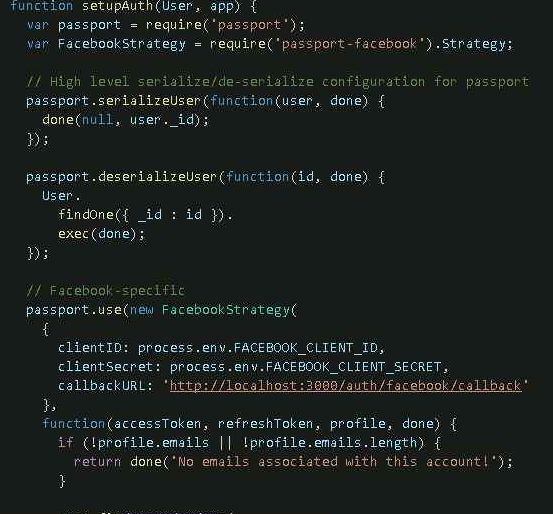
Again, make sure you get this right, or you're going to start getting cryptic app canvas URL errors.

Add wagner.invoke function in index.js file as below



So let's take a look at the auth.js file.

//auth.js



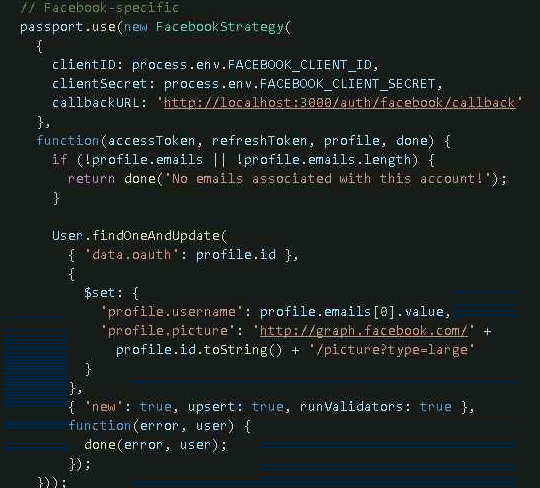
This function that it exports is responsible for taking an express app and attaching Facebook login to it. At a high level, this function does five distinct things.

The first thing you're going to do is require in passport and require in passport-facebook.



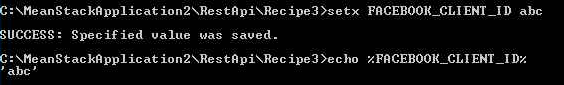
Next, you need to tell passport how to uniquely identify a user. In this case, you will use the MongoDB\_ID field to serialize a user, and deserialize it.

Third of all, you're going to set up a Facebook strategy for passport. This configures passport for using Facebook oauth. In particular, you need to give passport your app's ID and secret. In this case, this function pulls the app ID and secret from the FACEBOOK\_CLIENT\_ID and FACEBOOK\_CLIENT\_SECRET environment variables.



You also need to give Facebook a callback URL. This is the URL that users will be redirected to after they have successfully logged in. Environment variables are a lightweight mechanism for registering global variables in your shell.

So to create a new environment variable, you use this setx FACEBOOK\_CLIENT\_ID is equal to whatever your client ID is. And to show the current value of the environment variable, we will use echo followed by the environment variable name.



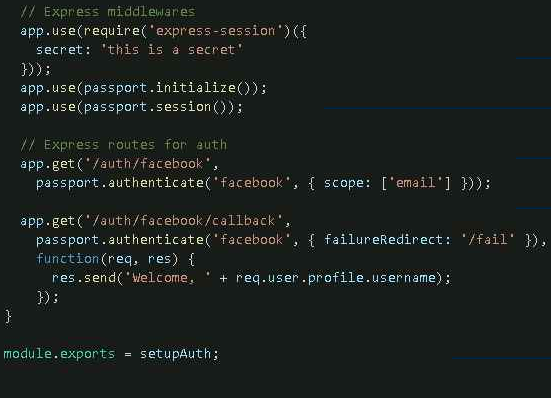
Now once you've set up an environment variable, you can then access it in node.js with this process.env property. Also, in order to properly set up a Facebook strategy, you need a way to get the user when they initially log in. That's what this entire function is responsible for.

First, this function errors out if the user doesn't have any emails, otherwise you're going to use Mongoose's findOneAndUpdate function to create a new user if necessary and return the updated user.



This findOneAndUpdate function does exactly what it sounds like. It finds a user by their oauth ID, as provided by Facebook, and updates their profile data based on the data returned from Facebook. And then it's going to return the updated user.

So once this is done you've successfully told passport how to integrate Facebook OAuth with your application. All that's left is to set up the Express middleware and routes. So the fourth thing you need to do is set up Express middleware.



Order matters here, so be careful. First thing you need to do is set the Express session middleware with a secret that can be whatever secret you want. Second is the Passport middleware. And then finally is the Passport session middleware.

Finally, you need to register two routes. The first route is the route the user visits to start the login process. The second route is the route that Facebook sends users back to, after they've successfully enabled your app. This second route, the callback URL, needs to match the callback URL that you specified in your Facebook strategy.

So in other words, the first route redirects the user to Facebook, and tells them that we're requesting their email address. The second route logs the user in and shows a success message.

**Recipe7: Checkout API**

