Nikita Bokil

Mobile to Cloud: Building Distributed Apps (67-328) | 12/12/2015

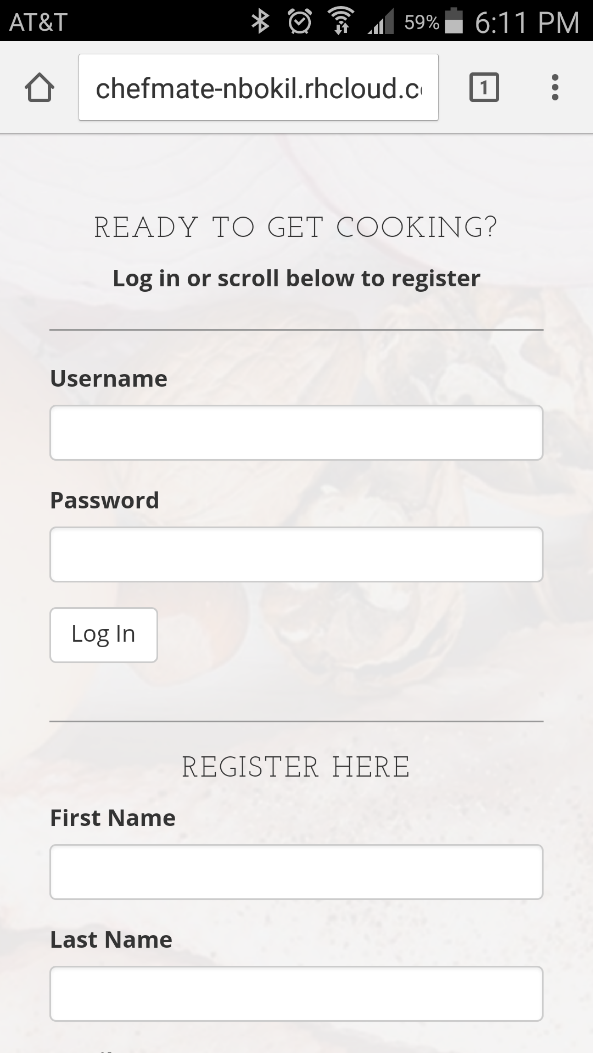
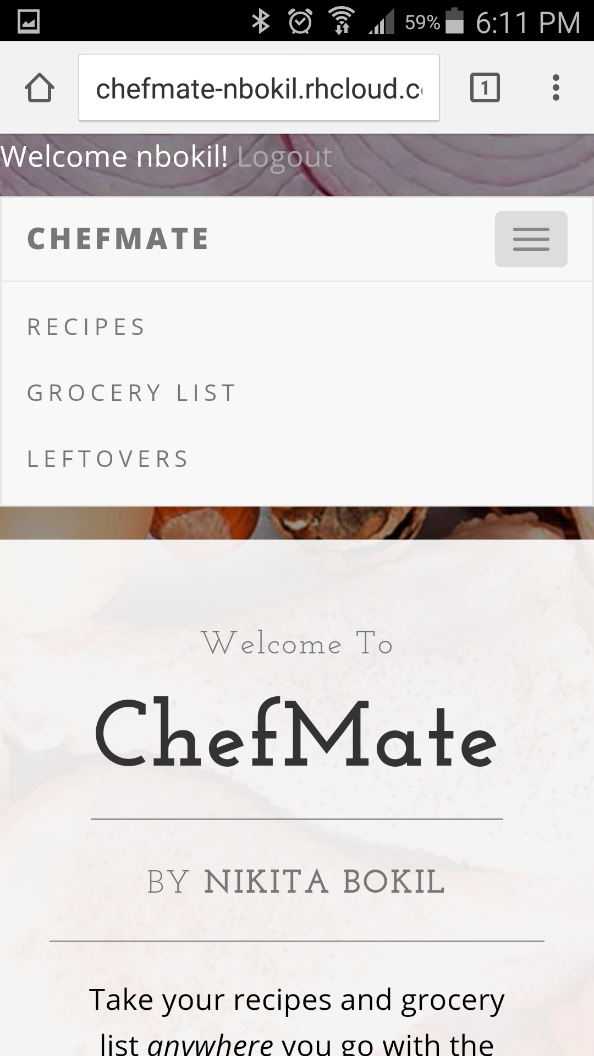
ChefMate Final App

# *http://chefmate-nbokil.rhcloud.com/*

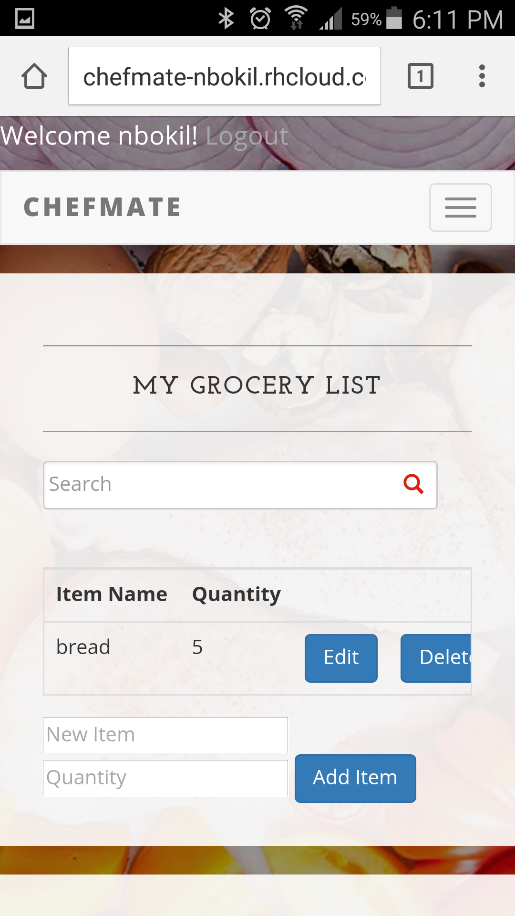
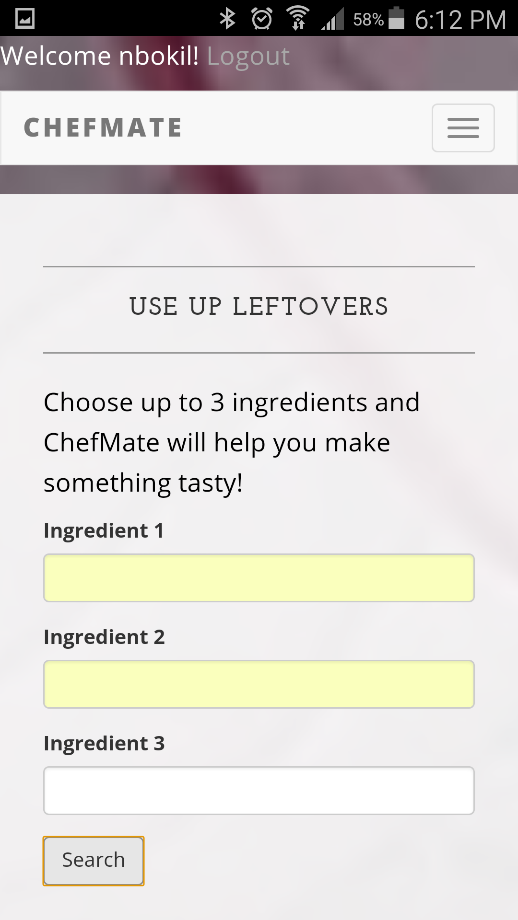
# Section 1: Description

ChefMate is an application that assists users in the kitchen by managing their grocery lists and generating recipes based on leftover items they may have. The application works by having users register for an account and log in to the dashboard. The user can then navigate the tabs to manage their grocery list or search for recipes. The grocery list tab stores a list of items to buy at the store, with fully functioning CRUD. The leftovers tab uses the Yummly API to search for recipes given a list of three leftover ingredients. The app is primarily designed to be mobile so that users can use it when they are in the grocery store or in the kitchen. The following are screenshots of user interaction:

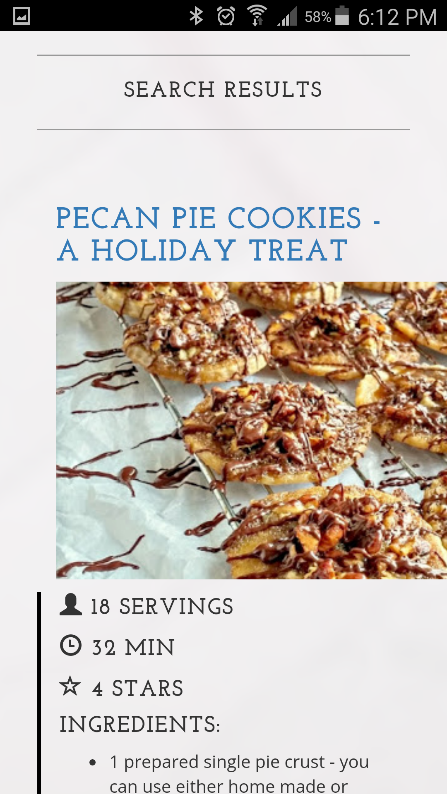
*Figure 1.1 – Login/Logout Figure 1.2 – Welcome screen*

**

*Figure 1.3 – Grocery List Figure 1.4 – Recipe Search*

**

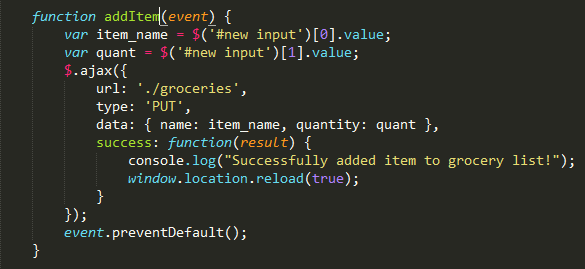
*Figure 1.5 – Recipe Search Results*

**

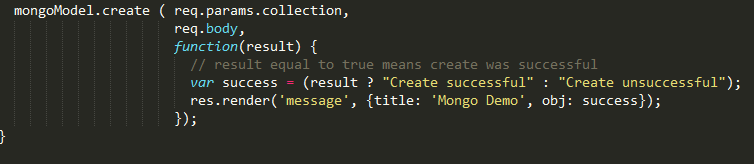
# Section 2: Client-Server Interaction

ChefMate updates the content displayed on the client based on interaction with the server in multiple functions, including user registration, login, and grocery list CRUD. For example, when a user inputs an item name, quantity, and clicks ‘Add Item’ to their grocery list (Figure 1.3), the client sends an ajax request for PUT(/:collection) which is handled by routes to call the create function in mongoModel.js. The server responds by sending data upon success, and the browser is updated with the new grocery list through the callback function. Screenshots of this interaction are shown below:

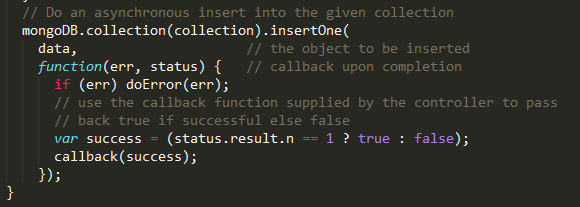
*Figure 2.1 – Client call to server*



*Figure 2.2 – dbRoutes handles call to server*



*Figure 2.3 – mongoModel creates item and performs callback*



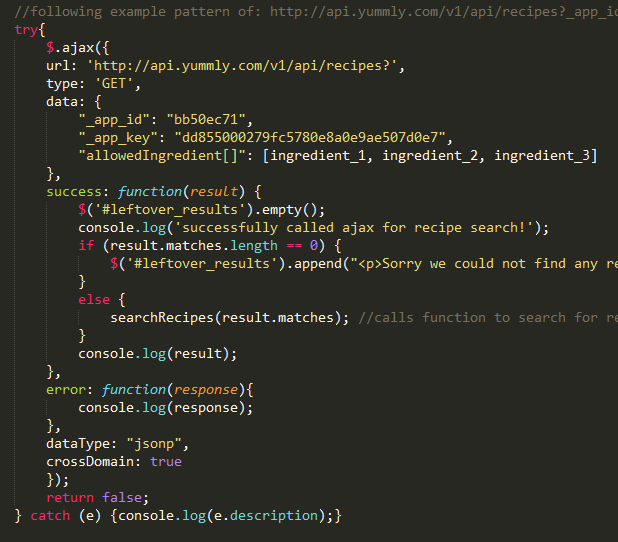
# Section 3: AJAX Interaction

Ajax interaction is used throughout my application when making calls to the server or to the API. For an example of ajax interaction please see the Figures in section 2 for adding an item, which display how an Ajax call was made to routes and a callback function rendered the result upon success.

# Section 4: 3rd Party API

ChefMate interacts with the 3rd party web service Yummly, whose documentation can be found here: <https://developer.yummly.com/documentation>. First, a GET request is performed to search for recipes with the parameter allowedIngredients[], which means the resulting recipes must have those ingredients. This request returns a set of “matched” recipes. Next, another set of GET requests are performed on each “matched” recipe\_id, obtained from the first request, to find the numberofservings, totalTime, and ingredientLines for each recipe. Due to the extensive API calls, only 10 matching recipe results are returned, which is appropriate considering the app is primarily mobile and users will not want to scroll through pages of recipes. Screenshots of the API calls are shown below:

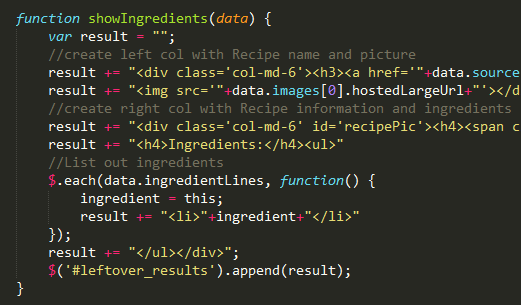
*Figure 4.1- First API Call to GET matching recipes*



*Figure 4.2 – Second API calls on each matched recipe*



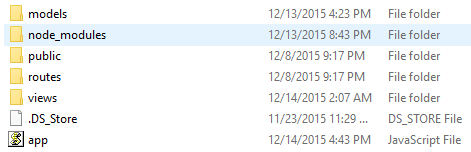
*Figure 4.3 – Display results on client side so they are nicely formatted*



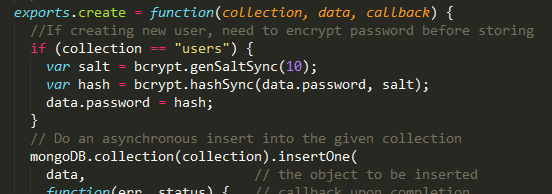
# Section 5: MVC

Separation of concerns is handled in ChefMate through Model-View-Controller structure. The model is contained in a separate file and handles all interactions with database and any algorithms manipulating data. For example, in order to encrypt passwords before they are stored to the database, the password is passed from the client to server, and then encrypted in the mongoModel.js file. The controller is the dbRoutes.js file, which renders the appropriate pages and manages calls from the client to server. The routes file also handles authorization, checking if a user is logged in or not to display the appropriate view. Subsequently, the views are stored in a views folder, and rendered by the routes. The views also use partials to reduce the amount of repetitive code, such as nav bar or header. Samples of MVC format are shown below:

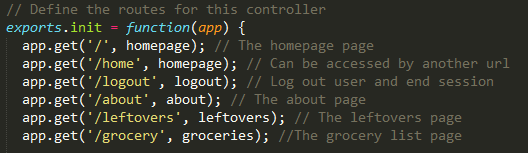
*Figure 5.1 – Structure of file directory*



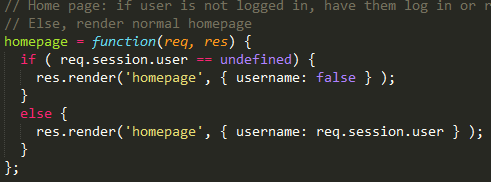
*Figure 5.2 – Model Logic for encrypting password upon registration*



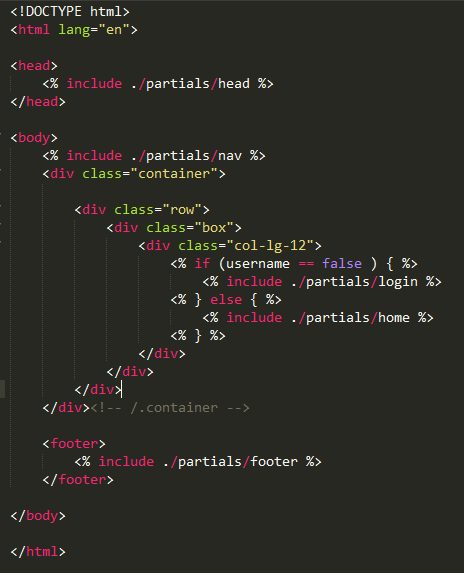
*Figure 5.3 – Routes handling URLs*



*Figure 5.4 – Routes handling authentication and rendering views*



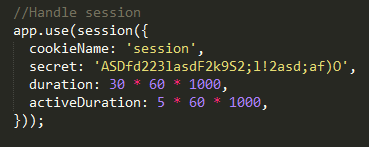
*Figure 5.5 – Views with partials*



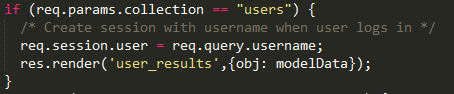
# Section 6: Cookies/Session

Sessions are used to manage web application state information. I used the node-module client-sessions to store ‘session’ information in a cookie, and access it through req.session. In the routes file, req.session.user refers to the current user, who is either a username or undefined depending on if the user is logged in. When a user logs in, req.session.user is set to the username they enter, and when a user logs out the session is reset using req.session.reset(); Authorization is managed through the session variable by preventing users from accessing pages if they are not logged in. Screenshots of session information are shown below:

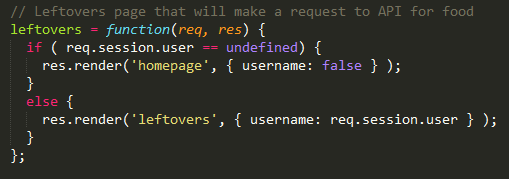
*Figure 6.1 – Initialize session in app.js*



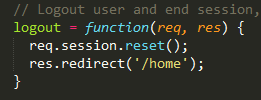
*Figure 6.2 – Create session user upon log in*



*Figure 6.3 – Handle authorization for users depending on session state*



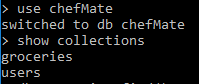
*Figure 6.4 – Reset session when user logs out*



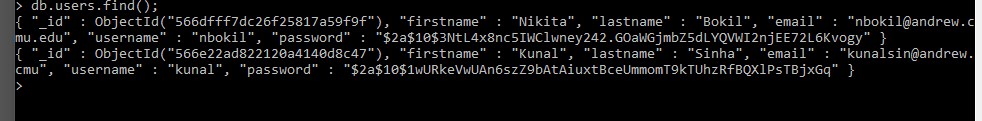
# Section 7: Persistent Data

The server-side database used in ChefMate is MongoDB. The information that is stored persistently is in the chefMate database, under two collections: users and groceries. Users stores {firstname, lastname, email, username, and password} when someone registers for the first time. Groceries stores {username, itemname, quantity} when a user adds items to his/her list. Screenshots of the stored data in my local mongo database can be seen below:

*Figure 7.1 – Collections*



*Figure 7.2 – Users Collection*



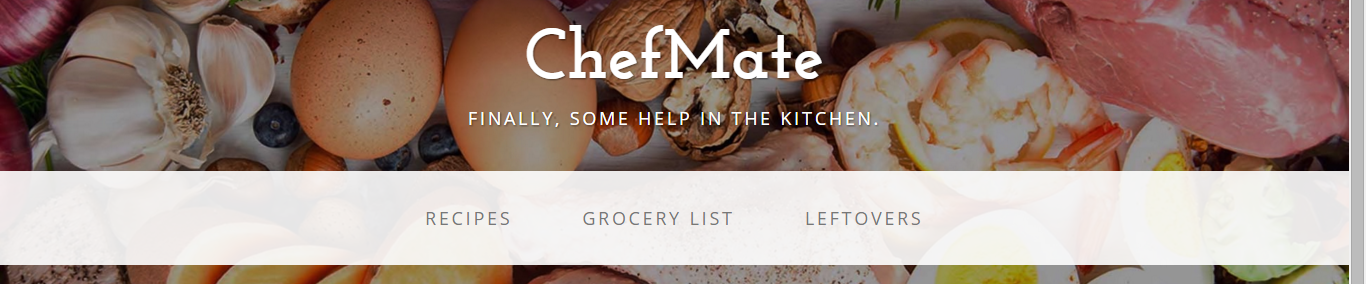
*Figure 7.3 – Groceries Collection*



# Section 8: Desktop Access

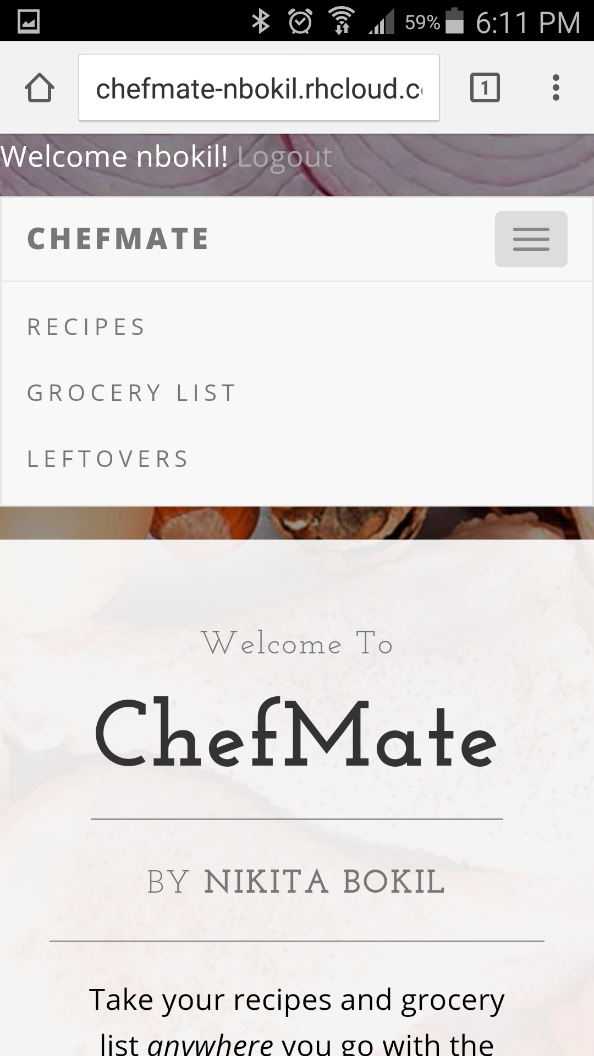
The access on desktop is very similar to mobile, with the exception of the navbar. Since I used Bootstrap for my CSS framework, it helps with automatically setting up responsive web design. For example, when creating a responsive table to display grocery items, I entered “<table class=”responsive”>, which changes the size of the table depending on the screen. On desktops, the navbar is displayed across the webpage as can be seen below:

*Figure 8.1 – Navbar*



# Section 9: Mobile Access

The access on mobile is very similar to desktop, with the exception of the navbar. On mobile, the navbar is collapsed and displayed as can be seen below:

**

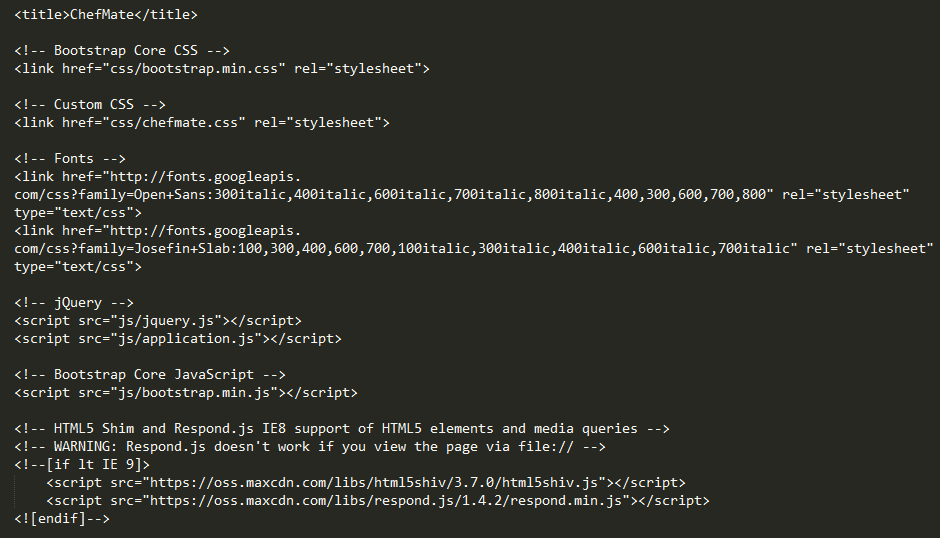
# Section 10: Work when deployed on Open Shift

The ChefMate application can be accessed on openshift at the following url: <http://chefmate-nbokil.rhcloud.com/>

# Section 11: Separation of JS, CSS, and HTML

Javascript, CSS, and HTML have been separated into their appropriate files. Semantic html styles were used so that CSS solely would determine style. For example, <em> tags were used to show emphasis, versus <i>. Javascript files and CSS files were referenced in the HTML header, as can be seen below:

*Figure 11.1 – Header*



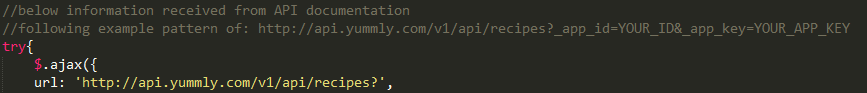
# Section 12: Appropriate choices of HTTP methods

The HTTP methods used are PUT, GET, POST, and DELETE in the appropriate areas. PUT is used to create users or grocery list items, GET is used to retrieve users for login or search for grocery list items, POST is used to update the quantity of a grocery list item, and DELETE is used to remove items from a grocery list. Samples can be seen in the code.

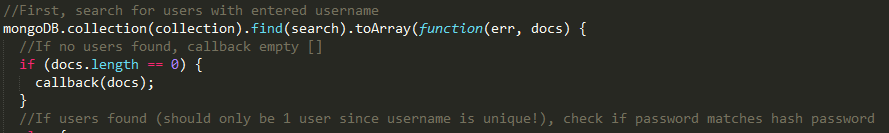
# Section 13: Good coding and commenting style

I have commented all my code with appropriate citations and explanations. Code has been separated to make it easy to read, following MVC architecture, separation of HTML, CSS, and JS, etc. Examples are shown below:

*Figure 13.1 – Commenting for API calls*



*Figure 13.2 – Commenting for Encrypted password matching*

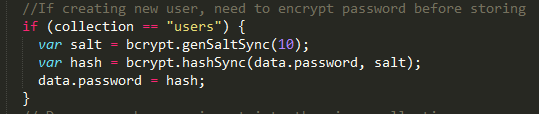


# Section 14: Bonus

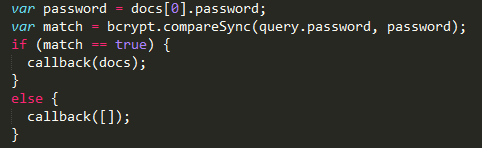
## Authentication and Authorization

Application employs authentication and authorization with the use of the node-modules ‘client sessions’ and ‘bcrypt’. When a user registers for the first time, his/her password is encrypted using the *bcrypt.genSaltSync()* function and *bcrypt.hashSync(password, salt)*, and then added to the database. When a user logs in, his/her encrypted password is checked with the password entered in the browser using the *bcrypt.compareSync(query.password, password)* function. If a user successfully logs in, their username is saved to the cookie session as *req.session.user = req.query.username*, which was obtained using the client-sessions node module. Screenshots of authentication are shown below:

*Figure 14.1 – Password Encryption*

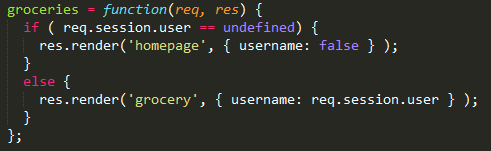


*Figure 14.2 – Password authentication check when user logs in*



Authorization occurs using the client-sessions node module. A user cannot access certain pages unless he/she is logged in. The login state is handled by *req.session.user = undefined,* or *req.session.user = username*. If a user accesses a page he/she is not authorized to, they will be re-routed to the login page. Screenshots of authorization are shown below:

*Figure 14.3 – Authorization for accessing grocery list*



## GitHub