AngularJS FoodMe Workshop

Instructions

Setup

Resources

- API Documentation : http://docs.angularjs.org/api
- Guide : http://docs.angularjs.org/guide
- Source : https://github.com/angular/angular.js/tree/master/src/ng
- ngModelController : https://github.com/angular/angular.js/blob/master/src/ng/directive/input.js#L891

Tools:

- Git: http://git-scm.com/
- NodeJS: http://nodejs.org/
- Some Text Editor (maybe Sublime Text 2: http://www.sublimetext.com/)

Install Global Dependencies

- Karma Test Runner, Bower Component Manager:
  ```
  npm install -g karma bower
  ```

Grab the code

```
git clone git://github.com/petebacondarwin/foodme.git
```

Local dependencies

```
cd foodme
npm install
```

Run Web Server (in separate terminal window)

```
node server/start.js
```

This will start a Node.js server and open your default browser on http://localhost:3000, where the app should be served.

Run the unit tests (in a separate terminal window)

```
karma start test/test.conf.js
```

This will start the Karma test runner, that will keep watching the files and whenever you make any change to a file, it will re-run all the unit tests.
It will also try to start Chrome. If you don't have Chrome installed, you can specify another browser for karma to use in the test-config.js file.

Step 1 - Static mocks

Starting point:
`foodme/step-1`

What we have:
This is an example of a static web-app mock that you might have received from a web-designer.

Goal:We need to turn the static mocks into an angular application:

What needs to be done?
- Install AngularJS using bower.
- Add an angular.js script tag.
- Add a app.js script tag.
- Create app.js file and define an angular module “foodMeApp”.
- Bootstrap the foodMeApp module in HTML.
- Verify that angular works by placing an angular expression {{1+2}} in your HTML.

Solution:
`foodme/step-1-solution`

Step 2 - Collect user data

Starting point:
`foodme/step-2`

What we have:
At this point, the angular application is bootstrapped.
We have a CustomerController in app.js and have implemented findRestaurants() to alert() the inputted text *(for now)*.

Goal:
Turn the mocked form into a working form. Use end-to-end testing to check that the form does what you expect it to.

What needs to be done?
- Run the end-to-end test scenarios to check your code.
- Add CustomerController to the HTML form element via the ng-controller directive.
- Add ng-model directive to the text input elements.
- Add ng-click directive to the form submit button to call findRestaurants() in CustomerController.

Solution:
`foodme/step-2-solution`

Step 3 - customer form validation

Starting point:
`foodme/step-3`

What we have:
A working form, which does not do any input validation.

Goal:
Add form validation and disable the submit button whenever a form has empty fields.

What needs to be done?
- Run the end-to-end test http://localhost:3000/test/e2e/runner.html to check your code.
- Add “customerForm” name to the form
- Add ng-disabled to the form submit button.
- Add required attributes to the input elements.

Solution:
`foodme/step-3-solution`

Step 4 - Customer model, localStorage persistence

Starting point:
`foodme/step-4`

What we have:
A working, self-validating, customer information form.

Goal:
Create a customer service which persists into localStorage.

What needs to be done?
- Create a localStorage service, to allow mocking in tests
- Create a customer service which persists name and address into localStorage

Solution:
`foodme/step-4-solution`

Step 5 - wire a customer model with CustomerController

Starting point:
`foodme/step-5`

What we have:
A customer form which does not persist, and a customer model which is auto-persisted when changed.

Goal:
Persist customer form into localStorage when the customer form is submitted.

What needs to be done?
- Copy customer name and address into the form in CustomerController constructor
- In the findRestaurants method copy customer information into customer service.

Solution:
`foodme/step-5-solution`

Step 6 - customer route

Starting point:
`foodme/step-6`

What we have:
All of the UI is in the index.html.

Goal:
Extract the UI from index.html into a separate file, and enable routing.

What needs to be done?
- Extract the HTML form index.html into views/customer.html, leaving behind only the chrome of the application, which is same on all pages. Place ng-view placeholder in where the extracted HTML used to be.
- Defined a root route in the app.js to point to views/customer.html and CustomerController.

Solution:
`foodme/step-6-solution`

Step 7 - static routes

Starting point:
`foodme/step-7`

What we have:
We're able to receive user information and persist it in localStorage, but top-level navigation doesn't work yet.
We created a customer service which persists name and address into localStorage, and wired it into app.js.
We defined a root route in the app.js to point to views/customer.html and CustomerController.
We created a view for each of the static pages: views/who-we-are.html views/how-it-works.html and views/help.html.

Goal:
Wire up the remaining views and get the navigation working.

What needs to be done?
- Run the end-to-end test http://localhost:3000/test/e2e/runner.html to check your code.
- Add routes in app.js for each menu item's view. Use menu link as the route url (without # prefix).

Solution:
`foodme/step-7-solution`

Step 8 - navbar selection

Starting point:
`foodme/step-8`

What we have:
Working menu system, which does not show which menu is active.

Goal:
Add active class to the menu item which is currently active.

What needs to be done?
- Create NavbarController.
- Create routeIs() method which determines if the current path matches the menu.
- Place ng-class in the navbar HTML which uses routeIs() method to add active class.

Solution:
`foodme/step-8-solution`

Step 9 - empty restaurants route + redirections

Starting point:
`foodme/step-9`

What we have:
Collect customer information form as the main screen.

Goal:
Create a list of restaurants placeholder as the main screen, move customer information screen into separate route definition.

What needs to be done?
- Create RestaurantController, which redirects to CustomerController if the current customer is not defined.
- Change the root route / to point to RestaurantController and the previous CustomerController now points to /customer route.
- Redirect to RestaurantControler form CustomerControler when the customer fills in their information.

Solution:
`foodme/step-9-solution`

Step 10 - static restaurants view

Starting point:
`foodme/step-10`

What we have:
Placeholder for the views/restaurants.html.

Goal:
Add static web-designer mocks to the views/restaurants.html placeholder.

What needs to be done?
- cut-and-paste web-designer mock data into views/restaurants.html.

Solution:
`foodme/step-10-solution`

Step 11 - Create fake restaurant data

Starting point:
`foodme/step-11`

What we have:
Static designer mock of restaurants.

Goal:
Add some fake restaurants into the controller.

What needs to be done?
- Create a fake restaurant model in the RestaurantController.

Solution:
`foodme/step-11-solution`

Step 12 - Data-bind to fake restaurant data

Starting point:
`foodme/step-12`

What we have:
An HTML mock with some hardcoded data for restaurants, working navigation, and defined views for the static pages.

Goal:
Add data binding to the mock so that the data in the views can be generated dynamically based on the data in the controller.

What needs to be done?
- Run the end-to-end test http://localhost:3000/test/e2e/runner.html to check your code.
- Refactor the views/restaurants.html to data bind to the data defined in the RestaurantController. (note: all restaurant images are stored as <restaurant.id>.jpg under /img/restaurants/)

Useful links
http://docs.angularjs.org/api/ng.directive:ngRepeat

Solution:
`foodme/step-12-solution`

Step 13 - fetch restaurants data via $http

Starting point:
`foodme/step-13`

What we have:
Designer mock of restaurants bound to fake data.

Goal:
Bind restaurants to data from the server.

What needs to be done?
- Replace the fake data-set of restaurants with a call to $http.get(). (Note the URL to use is "/api/restaurant").
- Use ng-src directive to ensure no prebinding calls to invalid image source URLs

Useful links
http://docs.angularjs.org/api/ng.directive:ngSrc

Solution:
`foodme/step-13-solution`

Step 14 - replace $http with Restaurant resource

Starting point:
`foodme/step-14`

What we have:
Restaurants page which is populated with $http.get() service.

Goal:
Simplify data fetching through the use of $resource over $http.get() service.

What needs to be done?
- Add dependencies on ngResource in foodMeApp module.
- Create the Restaurant resource.
- Inject Restaurant resource into RestaurantsController.
- Replace $http.get() with Restaurant.query().

Solution:
`foodme/step-14-solution`

Step 15 - wire up the 'deliver to' info panel

Starting point:
`foodme/step-15`

What we have:
Restaurants list.

Goal:
Add deliver to information on top of the restaurants view.

What needs to be done?
- Inject customer service into RestaurantsView and expose it to the view.
- Add HTML to show current deliver information in views/restaurants.html.

Solution:
`foodme/step-15-solution`

Step 16 - add ngPluralize

Starting point:
`foodme/step-16`

What we have:
Count of restaurants without pluralization. (we show 1 restaurants found!)

Goal:
Change the message depending on the number of restaurants returned.

What needs to be done?
- Replace the restaurant count message with ng-pluralize.

Solution:
`foodme/step-16-solution`

Step 17 - static menu view

Starting point:
`foodme/step-17`

What we have:
List of restaurants, but no view of the menu for a restaurant
A mock of the menu view from the designer

Goal:
Show the static restaurant menu mock from the web-designer.

What needs to be done?
- Run the end-to-end test http://localhost:3000/test/e2e/runner.html to check your code.
- Create empty MenuController as a placeholder.
- Create parameterized /menu/:restaurantId route to point to the MenuController and the views/menu.html mock.

Solution:
`foodme/step-17-solution`

Step 18 - add fmDeliverTo directive

Starting point:
`foodme/step-18`

What we have:
An inline "deliver to" message that is repeated in both the views/restaurants.html and views/menu.html views.

Goal:
Extract the message into a reusable component. Introduce unit tests into our code, and run them.

What needs to be done?
- Start the unit testing server: `karma start test/unit.conf.js`
- Create fmDeliverTo.js component using the directive api. The component should use the js/directives/fmDeliverTo.html as its template.
- Replace the inline HTML with the extracted component.

Solution:
`foodme/step-18-solution`

Step 19 - data-bind the menu view

Starting point:
`foodme/step-19`

What we have:
Static web-designer mock of the menu.

Goal:
Data-bind the menu to real data from the server.

What needs to be done?
- Retrieve restaurant from the server.
- Update the bindings in the HTML to bind to actual data.

Solution:
`foodme/step-19-solution`

Step 20 - stars & dollars filters

Starting point:
`foodme/step-20`

What we have:
Currently prices and ratings show up as numbers.

Goal:
Create a filter to display prices and ratings using $ and ★.

What needs to be done?
- create $ and ★ filter and register it with foodMeApp.

Solution:
`foodme/step-20-solution`

Step 21 - fmRating directive to replace filters and data-bind to filter

Starting point:
`foodme/step-21`

What we have:
Filters to render $ and ★.

Goal:
Replace these filters with a more versatile directive that can do both one-way binding (similar to what filters did) as well as two-way data-binding.

What needs to be done?
- convert dollar and star filters to a fmRating element directive (*restrict: 'E'*).
- this directive should have an databound attribute called rating, which is updated when you click on a symbol
- it should also take the following interpolated attributes: max (default to 5), symbol (default to *) and readonly (default to false)
- add behaviour to mouse over to highlight symbol.
- replace all instances of rating filter with this directive, applying readonly accordingly

Solution:
`foodme/step-21-solution`

Step 22 - wire up rating and price filters

Starting point:
`foodme/step-22`

What we have:
Currently the filtering in our restaurants view does nothing.

Goal:
Enable filtering of the restaurant list. Add unit tests to the new filtering code.

What needs to be done?
- Create a filter property (to hold price and rating) on RestaurantsController
- Watch the filter property of RestaurantsController and filter the list of restaurants accordingly.

Solution:
`foodme/step-22-solution`

Step 23 - fmCheckboxList Directive

Starting point:
`foodme/step-23`

What we have:
A static list of cuisines.

Goal:
Turn the static list into a dynamic list and add unit tests for it.

What needs to be done?
- create a fmCheckboxList directive which works with ng-model to render a dynamic list of checkboxes.
- add fmCheckboxList.js to index.html
- update RestaurantsController to provide the list of cuisine types on the scope and to filter based on the cuisine type
- wire up this list of cuisine types to the fmCheckboxList, removing static checkbox list

Solution:
`foodme/step-23-solution`

Step 24 - add sorting

Starting point:
`foodme/step-24`

What we have:
Restaurants are returned always in the same order.

Goal:
Allow sorting of the restaurant list and add unit tests for it.

What needs to be done?
- implement sorting algorithm on RestaurantsController
- add ng-click handlers to links on headers of restaurants table to trigger sorting

Solution:
`foodme/step-24-solution`

Step 25 - cart + add/remove stuff from cart

Starting point:
`foodme/step-25`

What we have:
Menu view does not allow ordering.

Goal:
Allow adding menu items into a cart.

What needs to be done?
- Create a cart service which keeps track of ordered items.
- Inject cart service into MenuController.
- Enable links to add items into the cart service.
- Show the checkout button if the cart has items
- Show cart totals

Solution:
`foodme/step-25-solution`

Step 26 - static checkout view

Starting point:
`foodme/step-26`

What we have:
We have a static mock checkout view from the designer

Goal:
Add static mock of the checkout view.

What needs to be done?
- Create CheckoutController.
- Add new route /checkout to routes.

Solution:
`foodme/step-26-solution`

Step 27 - wire up checkout view

Starting point:
`foodme/step-27`

What we have:
Static checkout view.

Goal:
Wire up the checkout view to show the contents of the cart and allow the purchase.

What needs to be done?
- Update the static mock with databindings to render the content of the shopping cart.
- Implement a purchase() button in CheckoutController which submits the order to server and clears the shopping cart.

Solution:
`foodme/step-27-solution`

Step 28 - static thank-you view

Starting point:
`foodme/step-28`

What we have:
We don't have checkout confirmation page

Goal:
Add web-designer mocks of the checkout confirmation page.

What needs to be done?
- Create ThankYouController.
- Put checkout mocks into views/thank-you.html.
- Add new route /thank-you to routes.

Solution:
`foodme/step-28-solution`

Step 29 - wire up thank-you view

Starting point:
`foodme/step-29`

What we have:
Static mock of thank you page.

Goal:
Add behavior to the mock.

What needs to be done?
- Show order ID as return from the server

Solution:
`foodme/step-29-solution`

Conclusion

What we have: