Workshop Instructions

Please see the **Prerequisites Document** before you get started

Get the code

Use git to clone the repository from github:

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

The project has an **app** folder that will hold the application and a **test** folder that will hold the tests and test config for the application. There are some other configuration and support files in the project's root folder:

- **.bowerrc** - tells bower where to put components it downloads
- **.gitignore** - tells git not to commit folders that are creatd by npm and bower
- **bower.json** - tells bower what components are needed
- **package.json** - tells npm what tools are needed

You are now ready to start building the application!

Git Tools

If you want to jump to a step then you can use:

```
git checkout –f step-X
```

where **step-X** is the name of the step to jump to and **-f** tells git to overwrite any changes you made (be careful using this).

If you want to commit your changes that you have made then you can use:

```
git commit –A
```

where **-A** tells git to include all changed files (added, modified and deleted) to the commit.

View Changes on GitHub

You can follow the changes between two steps in the github repository by going to URLs of the form:

https://github.com/petebacondarwin/foodme/compare/step-X...step-Y

Background Reading

Many of the steps have a Topics section. In here you will see keywords of things to read up on that are related to this step. You can find out about these topics at the Angular documentation site:

https://docs.angularjs.org/

Step 1 - Setup the Angular App

We start with an initial static HTML mock up, such as you could have received from a designer. The files are all contained in the app folder, notably:

- css - contains the CSS stylesheets
- img - contains images for the application - such as restaurant photos
- views – contains partial HTML mock ups of other views in the application
- index.html – a mock up of the landing page, which will become the main page for the Angular application
- js - will hold the JavaScript application code and libraries

You can view the landing page by running:

```
npm start
```

This will install the necessary node tools and bower components then start a local web-server for you that will listen on port 8000. You can now browse the application at:

http://localhost:8000

You can leave this local web server running in the background and use a different terminal/command prompt to execute any other commands later in the workshop. This way at any time you can refresh the browser and see the current state of the application.

Goals

- Convert the static HTML to an Angular application

Topics

- Bower
- Angular Apps
- Directives

Tasks

1.1. Add Angular to bower.json

Add angular and angular-mocks and angular-i18n to the dependencies in bower.json:

```
"dependencies": {
  "bootswatch": "~3.1.1",
   "angular": "^1.2.16",
   "angular-mocks": "^1.2.16"
 }
```

This will tell bower what components to install the next time it is run. We can run bower by using:

```
npm install
```

1.2. Create Angular app module

Create a JavaScript file, **app/js/app.js**, that will contain the top level angular module:

```
angular.module('foodme', []);
```

1.3. Load JavaScript files in index.htm

```
<head>
  ...
  <script src="bower_components/angular/angular.js"></script>
  <script src="js/app.js"></script>
</head>
```

1.4. Define ng-app in the index page

```
<html lang="en" ng-app="foodme">
```

This tells Angular to parse and compile all HTML below the **<html>** element. We could have put the **ng-app** directive on a different element if we only wanted a portion of our page to be compiled by Angular.

Step 2 - Implement a Customer Info page

We now have an Angular application running but it doesn't do very much. Let's add some functionality. Angular implements functionality by decorating the HTML with **directives** and **bindings**. Directives are HTML elements and attributes, which trigger functionality and bind data from the Angular scope to the HTML elements. Bindings use double curly braces ({{}}) to bind the HTML to data, which Angular will keep in sync for us.

Goals

- Bind the Customer Info form to a model on the Angular scope.
- Display the customer info when the user clicks "Find Restaurants".

Topics

- Scopes
- Data-binding

Tasks

2.1. Bind the customer name input element to **customer.name**

Change **index.html**:

```
<input type="text" …  ng-model="customer.name">
```

2.2. Bind the customer address input element to **customer.address**

Change **index.html**:

```
<input type="text" ... ng-model="customer.address">
```

2.3. Bind the click event of the "Find Restaurants" button

Change **index.html**:

```
<button ... ng-click="findRestaurants()">Find Restaurants!</button>
```

2.4. Create a **CustomerController** in a new **customer** module

Create **js/customer.js**:

```
angular.module('customer', [])

.controller('CustomerController', [‘$scope', function($scope) {
  $scope.customer = {
    name: "Joe Black",
    address: "432 Wiggly Rd, Mountain View, 94043"
  };
  $scope.findRestaurants = function() {
    alert(customer.name ' - ' customer.address);
  };
}]);
```

2.5. Add the customer module to our Angular app

Change **index.html**:

```
<script src="js/customer.js"></script>
```

Change **js/app.js**:

```
<script src="js/customer.js"></script>
```

2.6. Bind the view to the CustomerController

Change **index.html**:

```
<div class="modal-dialog" ... ng-controller="CustomerController">
```

2.7. Write unit tests for the CustomerController

Create **test/unit/customer.spec.js**:

```
describe("customer", function() {

  beforeEach(module('customer'));

  describe("CustomerController", function() {
    it("should initialize the scope",
      inject(function($rootScope, $controller) {

      $controller('CustomerController', { $scope: $rootScope });
      expect($rootScope.customer).toEqual({
        name: "Joe Black",
        address: "432 Wiggly Rd, Mountain View, 94043"
      });

      expect($rootScope.findRestaurants).toEqual(
        jasmine.any(Function)
      );
    }));
  });

});
```

You can start the Karma test runner to watch and execute these tests by running:

```
npm test
```

You can leave this running in the background as you continute to developer so you get immediate feedback on any broken unit tests.

Extras

Try debugging the controller code, in the browser's debug console:

- Set a breakpoint in the **CustomerController**'s constructor function. You can see the **$scope** being initialized when you refresh the page.
- Set a breakpoint in the **findRestaurants** method. You can see the **alert()** being triggered when you click on the button.

Step 3 - Add Validation to the Customer Info Form

We have bound the form inputs to the scope and can handle a click event. Now let's add some validation to our customer form.

Goals

- Make name and address on the Customer Info form required fields
- Disable the "Find Restaurants!" button when the form is not valid

Topics

- Form Validation

Tasks

3.1. Give the form a name so we can access its validity

Change **index.html**:

```
<form ... ng-controller="CustomerController" name="customerForm">
```

Now we can access validity properties of the form on the scope such as **$scope.customerForm.$valid**.

3.2. Add required attributes to the input elements

Change **index.html**:

```
<input type="text" ... ng-model="customer.name" required>
…
<input type="text" ... ng-model="customer.address" required>
```

Notice that, when the inputs are empty, they now have the **ng-invalid** CSS class on them. In the **css/app.css** stylesheet, there is a rule that gives the input box a red border when this happens.

3.3. Disable the "Find Restaurants" button when the form is not valid

Change **index.html**:

```
<button class="btn btn-primary" ng-click="findRestaurants()"
        ng-disabled="customerForm.$invalid">Find
Restaurants!</button>
```

Now whenever at least one of the inputs is invalid, i.e. empty, the whole form is invalid, which in turn disables the button.

3.4. Add end to end tests to check the validation

Change **test/e2e/app.scenario.js**:

```
describe('Customer info form', function() {

  beforeEach(function() {
    browser.get('index.html');
  });

  it('should add $invalid CSS class to invalid inputs', function() {
    var nameInput = element(by.model('customer.name'));
    nameInput.clear();
    expect(nameInput.getAttribute('class')).toContain('ng-invalid');
  });

  it('should disable the button when a field is invalid', function()
{
    var nameInput = element(by.model('customer.name'));
    var getRestaurantsButton =
          element(by.css('form[name="customerForm"] button'));
    nameInput.clear();
    expect(getRestaurantsButton.getAttribute('disabled'))
        .toEqual('true');
  });
})
```

You can run the end to end tests with Protractor. First make sure that the local web server is running:

```
npm start
```

Then at a new command prompt

```
npm run protractor
```

Step 4 - Persist the Customer Info in LocalStorage

It is nice if the customer does not have to enter their information every time. We can use the browser's local storage to persist this information between visits to the site.

Goals

- Create a **customerInfo** angular service, which persists its value **LocalStorage**.
- Bind the **customerInfo** service to the Customer Info form.

Topics

- Angular Services
- Dependency Injection
- Scope Watches

Tasks

4.1. Create a new localStorage module with localStorage and localStorageBinding services

Create **js/common/localStorage.js**:

```
angular.module('common/localStorage', [])

.value('localStorage', window.localStorage)

.factory('localStorageBinding', ['localStorage', '$rootScope',
    function(localStorage, $rootScope) {

  return function(key, defaultValue) {
    defaultValue = JSON.stringify(defaultValue || {});

    var value = JSON.parse(localStorage[key] || defaultValue);

    $rootScope.$watch(function() { return value; }, function() {
      localStorage[key] = JSON.stringify(value);
    }, true);

    return value;
  };
}]);
```

Wrapping the **localStorage** browser object like this makes it easier to mock out when testing. The **localStorageBinding** service is a function that will return an object that is persisted to the **localStorage** every time it changes. This uses the **$scope.$watch** mechanism to know when the object has changed.

4.2. Add a customerInfo service to the customer module, which uses localStorageBinding

Change **js/customer.js**:

```
angular.module('customer', ['common/localStorage'])

.factory('customerInfo', ['localStorageBinding',
    function(localStorageBinding) {

  return localStorageBinding('fmCustomer');

}])
```

Adding **common/localStorage** as a module dependency simply ensures that this module will also be loaded into the Angular app when the **customer** module is loaded.

4.3. Add unit tests for the customerInfo service

Change **test/unit/customer.js**:

```
describe('customer', function() {
  var customer, localStorage, $rootScope;

  ...

  describe("customerInfo", function() {
    beforeEach(module(function($provide) {
      localStorage = {
        fmCustomer: '{"name":"init-name","address":"init-address"}'
      };
      $provide.value('localStorage', localStorage);
    }));

    beforeEach(inject(function(_customerInfo_, _$rootScope_) {
      customer = _customerInfo_;
      $rootScope = _$rootScope_;
    }));


    it('should update any change to localStorage', function() {
      $rootScope.$apply(function() {
        customer.name = 'Michael Jackson';
        customer.address = '2231 Planet Mars, Apt 501';
      });

      expect(localStorage.fmCustomer).toBe(
'{"name":"Michael Jackson","address":"2231 Planet Mars, Apt 501"}'
      );
    });


    it('should load initial value from localStorage', function() {
      expect(customer.name).toBe('init-name');
      expect(customer.address).toBe('init-address');
    });
  });
});
```

Here we provide a mock **localStorage** service for our test.

You might notice that we are using a special syntactic sugar for injecting the dependencies (**_customerInfo_** and **_$rootScope_**). The injector ignores the underscores wrapping the variable name and makes it easier for us to assign to local variables.

Also, we are using **$rootScope.$apply()** to make changes to the scope. This ensures that a **$digest** is run after the change has been made.

4.4. Load the localStorage.js file in the index page

Change **index.html**:

```
<script src="js/common/localStorage.js"></script>
```

4.5. Inject customerInfo into CustomerController and attach to scope

Change **js/customer.js**:

```
.controller('CustomerController', ['$scope', 'customerInfo',
    function($scope, customerInfo) {

  $scope.customer = customerInfo;

  ...
}]);
```

4.6. Add unit tests for CustomerController

Change **test/unit/customer.js**:

```
...

describe('CustomerController', function() {
  var customer, scope;
  beforeEach(inject(function($controller, $rootScope) {
    customerInfo = {
      name: 'Bob Green', address: '123 Main St; Anytown AB 12345'
    };
    scope = $rootScope;
    $controller('CustomerController', {
      $scope: scope, customerInfo: customerInfo
    });
  }));

  it('should set up customer from customerInfo service', function() {
    expect(scope.customer.name).toEqual('Bob Green');
    expect(scope.customer.address)
      .toEqual('123 Main St; Anytown AB 12345');
  });

  it('should save customer name and address to customer', function()
{
    scope.customer.name = 'newName';
    scope.customer.address = 'newAddress';
    scope.$digest();
    expect(customerInfo).toEqual({
      name: 'newName', address: 'newAddress'
    });
  });
});
```

Step 5 - Add Routing and Static Views

We have a simple Angular app with one working form, which can validate the input and handle button events. Now let's add routing so that we can navigate to some static information views.

Goals

- Implement routing using the ngRoute Angular module

Topics

- Routing and Views

Tasks

5.1. Install and implement ngRoute

Change **bower.json**:

```
"dependencies": {
  "bootswatch": "~3.1.1",
  "angular": "^1.2.16",
  "angular-mocks": "^1.2.16",
  "angular-i18n": "^1.2.16",
  "angular-route": "^1.2.16"
}
```

Now run **npm install** to download and install the extra **angular-route** component. It should appear in the **app/bower_components** folder.

Change **index.html**:

```
<script src="bower_components/angular/angular.js"></script>
<script
  src="bower_components/angular-route/angular-route.js"></script>
```

We need to load up this component after angular.js has loaded.

5.2. Move the Customer Info form into its own view

Create **views/customerInfo.html**:

```
<div class="fm-customer-bkg img-rounded"></div>
<form class="modal-dialog" novalidate
  ng-controller="CustomerController" name="customerForm">
  ...
</form>
```

We move all the Customer Info form HTML from the index.html into a “partial” view template.

Change **index.html**:

```
...
<div class="row">
  <div class="col-lg-10 col-lg-offset-1">
```

```
      <ng-view></ng-view>
    </div>
  </div>
  ...
```

We replace all the Customer Info form HTML with a single **<ng-view>** element. As the route changes the HTML of the current view will appear below this element.

5.3. Add the customer-info route

Change **js/app.js**:

```
angular.module('foodme', ['ngRoute', 'customer'])

.config(['$routeProvider', function($routeProvider) {
  $routeProvider
    .when('/', {
      controller: 'CustomerController',
      templateUrl: 'views/customerInfo.html'
    });
}]);
```

We add a dependency on **ngRoute** to ensure that the module is loaded into our application.

In a **config** block of our application module, we configure a route for the root '**/**' path of our application. This corresponds to http://localhost:8000/index.html#/ where the bit after the hash is the path.

The route definition tells ngRoute that when the browser goes to the '**/**' path, it should replace fill the **<ng-view>** element with the contents found in **views/customerInfo.html** and bind the **CustomerController** to this view.

5.4. Add routes for static views

Change **js/app.js**:

```
.when('/who-we-are', {
  templateUrl: 'views/who-we-are.html'
})
.when('/how-it-works', {
  templateUrl: 'views/how-it-works.html'
})
.when('/help', {
   templateUrl: 'views/help.html'
});
```

There are other static (pure HTML) views in our **views** folder. Here we configure routes to map paths (such as '**/who-are-we**') to the relevant template.

5.5. Add end to end test to check the new views are accessible

Change **test/e2e/app.scenario.js**:

```
describe('Static views', function() {

  it('should show the help view at /help', function() {
    browser.get('index.html#/help');
    expect(element(by.css('.fm-heading')).getText())
      .toEqual('Help');
  });

  it('should show the "who we are" view at /who-we-are', function() {
    browser.get('index.html#/who-we-are');
    expect(element(by.css('.fm-heading')).getText())
      .toEqual('Who we are');
  });

  it('should show the "how it works" view at /how-it-works',
      function() {
    browser.get('index.html#/how-it-works');
    expect(element(by.css('.fm-heading')).getText())
      .toEqual('How it works');
  });
});
```

Step 6 - Add Dynamic Navigation Support

We can now access different views that are mapped to URL paths by routes. Let's make the navigation bar more dynamic so that it shows the active route by highlighting the current navigation menu item.

Goals

- Create a **NavbarController**, which will provide an **routeIs()** helper to identify the active route.
- Use the **ng-class** directive to apply the **active** CSS class to the navbar item that matches the current route.

Topics

- ngClass Directive

Tasks

6.1. Create a NavbarController in a new navigation module

Create **js/navigation.js**:

```
angular.module('navigation',[])

.controller('NavbarController', ['$scope', '$location',
    function($scope, $location) {

  $scope.routeIs = function(routeName) {
    return $location.path() === routeName;
  };

}]);
```

Create a NavbarController, which provides a helper to identify the current route.

Change **js/app.js**:

```
angular.module('foodme', ['ngRoute', 'customer', 'navigation'])
```

Add the navigation module as a dependency of the foodme application module.

Change **index.html**:

```
<head>
  ...
  <script src="js/common/localStorage.js"></script>
  <script src="js/navigation.js"></script>
</head>
```

Add the new JavaScript file to the index page to load the new controller code.

6.2. Use NavbarController to indicate the active nav item

Change **index.html**:

```
<div class="navbar navbar-default" ng-controller="NavbarController">
  <div class="container-fluid">
    ...
    <div class="collapse navbar-collapse">
       <ul class="nav navbar-nav">
          <li ng-class="{active: routeIs('/')}">
             <a href="#/">Home</a></li>
          <li ng-class="{active: routeIs('/how-it-works')}">
             <a href="#/how-it-works">How it works</a></li>
          <li ng-class="{active: routeIs('/who-we-are')}">
             <a href="#/who-we-are">Who we are</a></li>
       </ul>
       ...
```

For each navbar item, use the **ng-class** directive to modify the CSS class of the **<li>** elements based on whether the current URL path matches.

6.3. Add end to end test to check navbar items active correctly

Change **test/e2e/app.scenario.js**:

```
describe('Navbar', function() {
  it('should activate the current navbar item', function() {
    var activeItem = element(by.css('.active'));

    browser.get('index.html#/');
    expect(activeItem.getText()).toEqual('Home');

    browser.get('index.html#/how-it-works');
    expect(activeItem.getText()).toEqual('How it works');

    browser.get('index.html#/who-we-are');
    expect(activeItem.getText()).toEqual('Who we are');
  });
});
```

Step 7 - Create Restaurant List View

We have navigation and a basic customer info form. Now let’s start working through the main work flow of the application: browsing through a list of restaurants.

Goals

- Move the customer info view to its own customer-info route
- Create a home route that displays a list of restaurants and the customer delivery info
- Redirect the home route to the customer-info route if the customer has not entered their info.

Topics

- ng-repeat
- HTTP Requests (**$http**)
- Promises

Tasks

7.1. Create customer-info route that redirects to home when “Find Restaurants” is clicked.

Change **js/app.js**:

```
$routeProvider
  .when('/customer-info', {
     controller: 'CustomerController',
     templateUrl: 'views/customerInfo.html'
  })
```

Change **js/customer.js**:

```
.controller('CustomerController', [
   '$scope', 'customerInfo', '$location',
   function($scope, customerInfo, $location) {

  $scope.customer = customerInfo;

  $scope.findRestaurants = function() {
    $location.path('/');
  };
}]);
```

Inject the **$location** service, and then redirect to the home route when the **findRestaurants()** method is called.

Change **test/e2e/app.scenario.js**:

```
describe('Customer info form', function() {
  beforeEach(function() {
    browser.get('index.html#/customer-info');
  });
  ...
```

Update the end to end test to point to the new customer-info route.

Change **test/unit/customer.spec.js**:

```
it('should redirect the user to restaurant list',
    inject(function($location) {

  $location.path('/new-customer');
  expect($location.path()).toEqual('/new-customer');

  scope.findRestaurants();
  expect($location.path()).toEqual('/');
}));
```

7.2. Create route for restaurant list view with a **RestaurantsController**, which redirects to the **customer-info** route if no customer address is found.

Change **js/app.js**:

```
angular.module('foodme',
  ['ngRoute', 'customer', 'navigation', 'restaurants'])

.config(['$routeProvider', function($routeProvider) {
  $routeProvider
    .when('/', {
      controller: 'RestaurantsController',
      templateUrl: 'views/restaurant-list.html'
    })
```

Add the **restaurants** module as a dependency and the new home route to display the Restaurant List view.

Create **js/restaurants.js**:

```
angular.module('restaurants', ['customer'])

.controller('RestaurantsController',
    ['$scope', 'customerInfo', '$location',
    function($scope, customerInfo, $location) {

  if (!customerInfo.address) {
    $location.path('/customer-info');
  }

}]);
```

If the customer address is missing then redirect to the **customer-info** view.

Change **index.html**:

```
<head>
  ...
  <link rel="stylesheet" href="css/customer.css">
```

```
<link rel="stylesheet" href="css/restaurant-list.css">
<link rel="stylesheet" href="css/fm-rating.css">
...
<script src="js/restaurants.js"></script>
...
```

Add some new CSS styles for the restaurant view and the new JavaScript file.

7.3. Add static restaurant data to the RestaurantsController

Change **js./restaurants.js**:

```
$scope.restaurants = [
  {
    "id":"angular",
    "name":"Angular Pizza",
    "cuisine":"pizza",
    "price":1,
    "rating":5,
    "description":"Home of the superheroic pizza!"
  },{
    "id":"tofuparadise",
    "name":"BBQ Tofu Paradise",
    "cuisine":"vegetarian",
    "price":2,
    "rating":1,
    "description":"Vegetarians, we have your BBQ needs covered. Our
home-made tofu skewers and secret BBQ sauce will have you licking
your fingers."
  },{
    "id":"beijing",
    "name":"Beijing Express",
    "cuisine":"chinese",
    "price":2,
    "rating":4,
    "description":"Fast, healthy, Chinese food. Family specials for
takeout or delivery. Try our Peking Duck!"
  }
];
```

Add some dummy data to the **RestaurantsController**.

7.4. Bind the restaurant data to the restaurant list view

Change **views/restaurant-list.html**:

```
<div class="fm-heading">
  {{restaurants.length}} restaurants found!
</div>
...
<tr ng-repeat="restaurant in restaurants">
  <td>
    <a href="#/menu/{{restaurant.id}}">
      <img class="img-rounded pull-left"
           ng-src="img/restaurants/{{restaurant.id}}.jpg">
        <strong>{{restaurant.name}}</strong>
    </a>
    <p>{{restaurant.description}}</p>
  </td>
  <td>{{restaurant.rating}}</td>
  <td>{{restaurant.price}}</td>
...
```

Replace all the mock restaurant HTML with this repeated template. Angular will create a **<tr>** element for each item in **restaurants** and bind the repeated template to each item. This will then be kept in sync by Angular if the **restaurants** data changes later.

7.5. Add end to end tests for restaurant list view

Change **test/e2e/app.scenario.js**:

```
beforeEach(function() {
  browser.get('index.html');
  browser.executeScript(
    'window.fmTempCust = window.localStorage.getItem("fmCustomer");'
  );
  browser.executeScript(

'window.localStorage.setItem("fmCustomer",\'{"address":"abc"}\');'
  );
});
afterEach(function() {
  browser.executeScript(
    'window.localStorage.setItem("fmCustomer", window.fmTempCust);'
  );
});
```

Since we now redirect when the localStorage does not contain the customer address we must modify this. These **beforeEach()** and **afterEach()** blocks ensure that any customer info is not permanently destroyed by these tests.

```
describe('Restaurant List', function() {

  it('should redirect to customer-info if no customer address',
      function() {

    browser.executeScript(
      'window.localStorage.removeItem("fmCustomer");'
    );
    browser.get('index.html#/');
    expect(browser.getCurrentUrl()).toContain('customer-info');

    browser.executeScript(
    'window.localStorage.setItem("fmCustomer",\'{"address":"abc"}\');
    ');
    browser.get('index.html#/');
    expect(browser.getCurrentUrl()).not.toContain('customer-info');
  });

  it('should display three restaurants', function() {
    browser.get('index.html#/');
    var heading = element(by.css('.fm-restaurant-list .fm-heading'))
    var rows = element.all(by.css('.fm-restaurant-list tbody tr'));
    expect(heading.getText()).toEqual('3 restaurants found!');
    expect(rows.count()).toEqual(3);
  });
});
```

7.6. Load the restaurant data from the server using $http

Change **js/restaurants.js**:

```
.controller('RestaurantsController',
  ['$scope', 'customerInfo', '$location', '$http',
  function($scope, customerInfo, $location, $http) {
...
  $http.get('data/restaurants.json').then(function(response) {
    $scope.restaurants = response.data;
  });
...
```

Inject the **$http** service into **RestaurantsController** and use it to “get” the restaurant data from the server. **$http** calls are asynchronous and so return a **promise** for the response, rather than the response itself. To get access to the response, we provide a **then()** handler, which “then” attaches the response data to the scope, when the response arrives.

Change **test/e2e/app.scenario.js**:

```
expect(heading.getText()).toEqual('39 restaurants found!');
expect(rows.count()).toEqual(39);
```

7.7. Bind the customerInfo to the deliverTo panel in the restaurant list view

Change **js/restaurants.js**:

```
$scope.deliverTo = customerInfo;
```

Change **views/restaurant-list.html**:

```
Deliver to: {{deliverTo.address}}
```

Step 8 - Filter Restaurants by Price and Rating

There are now 39 restaurants in the list. It would be good to filter these based on their rating and how expensive they are.

Goals

- Create Angular filters (Stars and Dollars) to modify how ratings and prices are displayed
- Filter the restaurant list by rating and price
- Use ngPluralize to fix pluralization restaurant count, e.g. not "1 restaurant**s** found!"

Topics

- Angular Filters
- ngPluralize

Tasks

8.1. Create Stars and Dollars Angular filters

Create **js/common/filters.js**:

```
angular.module('common/filters', [])

.filter('dollars', function() {
  var DOLLARS = {
    1: '$',
    2: '$$',
    3: '$$$',
    4: '$$$$',
    5: '$$$$$'
  };

  return function(dollarCount) {
    return DOLLARS[dollarCount];
  };
})

.filter('stars', function() {
  var STARS = {
    1: '\u2605',
    2: '\u2605\u2605',
    3: '\u2605\u2605\u2605',
    4: '\u2605\u2605\u2605\u2605',
    5: '\u2605\u2605\u2605\u2605\u2605'
  };

  return function(dollarCount) {
    return STARS[dollarCount];
  };
});
```

Create **test/unit/common/filters.spec.js**:

```
describe("filters", function() {
  beforeEach(module('common/filters'));

  describe("dollars", function() {
    it("should return a string of repeated dollars",
        inject(function(dollarsFilter) {
      expect(dollarsFilter(3)).toEqual('$$$');
    }));
  });

  describe("stars", function() {
    it("should return a string of repeated stars",
        inject(function(starsFilter) {
      expect(starsFilter(3)).toEqual('\u2605\u2605\u2605');
    }));
  });
});
```

Change **js/app.js**:

```
angular.module('foodme', [
  'ngRoute', 'customer', 'navigation', 'restaurants',
'common/filters'
])
```

Change **index.html**:

```
<script src="js/common/filters.js"></script>
```

8.2. Bind the Stars and Dollars filters to the restaurant list view

Change **views/restaurant-list.html**:

```
{{restaurant.rating | stars}}
...
{{restaurant.price | dollars}}
```

We use the pipe character "**|**" to apply a filter to a binding in an Angular template.

8.3. Filter the list of restaurants by price and rating

Change **js/restaurants.js**:

```
function filterRestaurants() {
  $scope.filteredRestaurants = [];
  angular.forEach($scope.restaurants, function(restaurant) {
    if ( ( !$scope.rating || restaurant.rating >= $scope.rating ) &&
         ( !$scope.price || restaurant.price <= $scope.price ) )
    {
      $scope.filteredRestaurants.push(restaurant);
    }
  });
}

$scope.$watch('rating', filterRestaurants);
$scope.$watch('price', filterRestaurants);
```

Add two watches to the **RestaurantsController**, which will filter the restaurant list when **rating** or **price** change.

Change **views/restaurant-list.html**:

```
<select ng-model="rating">
  <option value="">Any</option>
  <option value="1">★</option>
  <option value="2">★★</option>
  <option value="3">★★★</option>
  <option value="4">★★★★</option>
  <option value="5">★★★★★</option>
</select>
...
<select ng-model="price">
  <option value="">Any</option>
  <option value="1">$</option>
  <option value="2">$$</option>
  <option value="3">$$$</option>
  <option value="4">$$$$</option>
  <option value="5">$$$$$</option>
</select>
...
{{filteredRestaurants.length}} restaurants found!
...
<tr ng-repeat="restaurant in filteredRestaurants">
...
```

Replace the mock price and rating filter controls with **<select>** inputs, which are bound to the values on the scope being watched by the **RestaurantsController**. Use **filteredRestaurants** rather than **restaurants** in the list.

8.4. Automatically pluralize the number of matching restaurants

```
<ng-pluralize
  count="filteredRestaurants.length"
  when="{'0'    : 'No restaurants found.',
         'one'  : '1 restaurant found!',
         'other': '{} restaurants found!'}">
</ng-pluralize>
```

Extras

- Write a set of unit tests for the **RestaurantsController** that test the filtering of the restaurant list.

Step 9 - Create Restaurant Menu View

Now that we can filter and view the list of restaurants, we might like to look at their menu and maybe order some food.

Goals

- Create a menu route, parameterized by the restaurant id, which displays the menu for a restaurant.
- Refactor our controllers that rely on promises to use route **resolves** to unwrap promises for us
- Use ngPluralize to display the currency specific to our locale.

Topics

- Route Parameters
- Route Resolves
- Localization

Tasks

9.1. Add parameterized restaurant menu route

Change **js/app.js**:

```
.when('/menu/:restaurantId', {
  controller: 'MenuController',
  templateUrl: 'views/menu.html'
})
```

Note in the route path, we have **:restaurantId**, which is a route parameter. Any value which appears at this point in the route path will be extracted and added to the **$routeParams** service.

Change **index.html**:

```
<link rel="stylesheet" href="css/menu.css">
```

9.2. Refactor the restaurant data retrieval into a restaurantsPromise service, which is resolved in the restaurant list route

Change **js/restaurants.js**:

```
.factory('restaurantsPromise', ['$http', function($http) {
  return $http.get('data/restaurants.json').then(function(response) {
    return response.data;
  });
}])
...
.controller('RestaurantsController', ['$scope', 'customerInfo',
'$location', 'restaurants',
  function($scope, customerInfo, $location, restaurants) {
  ...
  $scope.restaurants = restaurants;
```

```
    ...
```

The **restaurantsPromise** service has been extracted from the **RestaurantsController**. The **RestaurantsController** now has a service called **restaurants** injected into it. This will come from the route **resolve**.

Change **js/app.js**:

```
$routeProvider
  .when('/', {
    controller: 'RestaurantsController',
    templateUrl: 'views/restaurant-list.html',
    resolve: {
      restaurants:
        ['restaurantsPromise', function(restaurantsPromise) {
          return restaurantsPromise;
        }]
    }
  })
```

The resolve property declares that there is a service that will return a promise and that the router should wait for this promise to resolve before changing to the new route. The resolved value from this promise will then be passed directly to the route's controller. This pattern is useful to keep our controllers synchronous and simpler.

9.3. Create a currentRestaurantPromise service, which is resolved in the menu route

Change **js/restaurants.js**:

```
.factory('currentRestaurantPromise',
    ['restaurantsPromise', '$route', '$routeParams',
    function(restaurantsPromise, $route, $routeParams) {

  return function() {
    return restaurantsPromise.then(function(restaurants) {
      for(var i=0; i < restaurants.length; i++) {
        if(restaurants[i].id === $route.current.params.restaurantId)
{
          return restaurants[i];
        }
      }
    });
  };
}])

.controller('MenuController', ['$scope', 'restaurant',
    function($scope, restaurant) {
  $scope.restaurant = restaurant;
}])
```

The **currentRestaurantPromise** service, combines the result of the promise from **restaurantsPromise** and the current **restaurantId** from the **$routeParams** to provide a promise to the current restaurant, to be used in the case of the menu view.

The **MenuController** will have this current **restaurant** injected into it as a resolve from the menu route.

Change **js/app.js**:

```
.when('/menu/:restaurantId', {
  controller: 'MenuController',
  templateUrl: 'views/menu.html',
  resolve: {
    restaurant: ['currentRestaurantPromise',
      function(currentRestaurantPromise) {
        return currentRestaurantPromise();
      }]
  }
})
```

9.4. Bind the current restaurant to the menu view

Change **views/menu.html**:

```
<div class="row fm-restaurant">
  <div class="col-md-2">
    <img ng-src="img/restaurants/{{restaurant.id}}.jpg"
         class="img-rounded">
  </div>
  <div class="col-md-10">
    <h3>{{restaurant.name}}</h3>
    <div class="row">
      <div class="col-md-2">
        <div>{{restaurant.address}}</div>
        <div>{{restaurant.rating | stars}}</div>
        <div>{{restaurant.price | dollars}}</div>
      </div>
      <div class="col-md-4">
        <div>{{restaurant.description}}</div>
      </div>
    </div>
  </div>
</div>

...

<li ng-repeat="menuItem in restaurant.menuItems">
  <a href>
    <span>{{menuItem.name}}</span>
    <span>{{menuItem.price | currency}}</span>
    <i class="glyphicon glyphicon-plus"></i>
  </a>
</li>
```

In this template, **restaurant** is the current restaurant resolved by the menu route from **currentRestaurantPromise**. We bind to the standard restaurant properties but then also iterate over the collection of **menuItems**.

9.5. Use the en-gb locale from angular-i18n to get pound signs for currency

```
<script src="bower_components/angular-i18n/angular-locale_en-gb.js">
</script>
```

By adding in a file from **angular-i18n**, we can get Angular automatically use different localization rules for things like numbers, dates and currency. You'll notice that before adding this file, all our bindings piped through the **currency** filter had a dollar sign. Now they should have a pound sign instead.

Extras

- Try out some different locales and see what difference it makes to your app. You can find them all in **app/bower_components/angular-i18n**.

Step 10 - Create a reusable fmDeliverTo component

Both the restaurant list view and the menu view display the same delivery information. This repetition of HTML is not good. We can build a reusable component directive to reduce this duplication.

Goals

- Create fmDeliverTo component directive.
- Use fmDeliverTo in the restaurant list and menu views.

Topics

- Component Directives
- Isolated Scope

Tasks

10.1. Create templated fmDeliverTo directive in new common/fmDeliverTo module

Create **js/common/fmDeliverTo/fmDeliverTo.js:**

```
angular.module('common/fmDeliverTo', [])

.directive('fmDeliverTo', function() {
  return {
    restrict: 'E',
    templateUrl: 'js/common/fmDeliverTo/fmDeliverTo.template.html',
    scope: {
      deliverTo: '='
    }
  };
});
```

This is a fairly simple component directive. It will insert the template specified by **templateUrl** into the HTML wherever it is used. It is restricted to being used only as an element (**restrict: 'E'**). Angular will create an isolated scope, and map the value in the **deliver-to** attribute on the directive's element to the **deliverTo** property on the isolated scope, which will be available to the template.

Create **js/common/fmDeliverTo/fmDeliverTo.template.html**:

```
<div class="row">
  <div class="col-md-12">
    <div class="breadcrumb fm-deliver-to">
      Deliver to: {{deliverTo.address}}
      <a href="#/customer-info" class="pull-right">Change</a>
    </div>
  </div>
</div>
```

Change **index.html**:

```
<script src="js/common/fmDeliverTo/fmDeliverTo.js"></script>
```

Change **js/app.js**:

```
angular.module('foodme', [
  'ngRoute', 'customer',
  'navigation', 'restaurants',
  'common/filters', 'common/fmDeliverTo'
])
```

10.2. Replace the inline HTML with the fm-deliver-to directive in the restaurant and menu views

Change **js/restaurants.js**:

```
.controller('MenuController', ['$scope', 'restaurant',
'customerInfo',
  function($scope, restaurant, customerInfo) {
    $scope.restaurant = restaurant;
    $scope.deliverTo = customerInfo;
}])
```

Change **views/menu.html** and **views/restaurant-list.html**:

```
<fm-deliver-to deliver-to="deliverTo"></fm-deliver-to>
```

Instead of the duplicate HTML we now just add one **<fm-deliver-to>** element, with the **deliver-to** attribute bound to the **deliverTo** value on the scope.

Step 11 - Implement a shopping cart in the menu view

Now that we can see what is on the menu, let's implement the shopping cart so we can choose what dishes we would like to order.

Goals

- Create a shoppingCart service that persists to localStorage and has methods for adding and removing menu items
- Bind the shoppingCart to the menu view

Tasks

11.1. Create a new shoppingCart service in a new shopping-cart module, which uses localStorageBinding

Create **js/common/alert.js**:

```
angular.module('common/alert', [])
.value('alert', window.alert);
```

Wrapping the global **alert()** function in a service makes it easier to mock out in tests.

Create **js/shopping-cart.js**:

```
angular.module('shopping-cart',['common/alert'])

.factory('shoppingCart', ['localStorageBinding', 'alert',
     function(localStorageBinding, alert) {
  var service = {
    add: function(choice, restaurant) {

      if ( !service.restaurant.id ) {
        service.restaurant = restaurant;
      }

      if ( service.restaurant.id !== restaurant.id ) {
        alert('You cannot mix items from different restaurant - clear
the shopping cart first.');
        return;
      }

      angular.forEach(service.items, function(item) {
        if (item.name === choice.name) {
          item.amount += 1;
          choice = null;
        }
      });

      if (choice) {
        service.items.push({
          name: choice.name,
          price: choice.price,
          amount: 1
        });
      }
    },

    remove: function(cartItem) {
      var index = service.items.indexOf(cartItem);
      if ( index !== -1 ) {
        service.items.splice(index, 1);
      }

      if (service.items.length === 0) {
        service.restaurant = {};
      }
    },

    total: function() {
      var sum = 0;
      angular.forEach(service.items, function(item) {
        sum += Number(item.price * item.amount);
      });
      return sum;
    },

    reset: function() {
      service.items = [];
      service.restaurant = {};
    },

  };

  service.items = localStorageBinding('fmCartItems', []);
  service.restaurant = localStorageBinding('fmCartRestaurant');

  return service;
}]);
```

The **shoppingCart** service is mostly standard JavaScript, except that it uses the **localStorageBinding** service to persist the cart **items** and **restaurant** to **localStorage**.

Change **js/app.js**:

```
angular.module('foodme', [
  'ngRoute', 'customer',
  'navigation', 'restaurants',
  'shopping-cart', 'common/filters',
  'common/fmDeliverTo'
])
```

Change **index.html**:

```
<script src="js/shopping-cart.js"></script>
```

11.2. Bind the shoppingCart to the menu view

Change **js/restaurants.js**:

```
.controller('MenuController',
    ['$scope', 'restaurant', 'customerInfo', 'shoppingCart',
    function($scope, restaurant, customerInfo, shoppingCart) {
  $scope.restaurant = restaurant;
  $scope.deliverTo = customerInfo;
  $scope.shoppingCart = shoppingCart;
}])
```

Change **views/menu.html**:

```
<a href>
  <span>{{menuItem.name}}</span>
  <span>{{menuItem.price | currency}}</span>
  <i class="glyphicon glyphicon-plus"
     ng-click="shoppingCart.add(menuItem, restaurant)"></i>
</a>
...
<div class="fm-panel fm-cart">
  <div class="fm-heading">Your order</div>
  <div class="fm-content">
    <div class="fm-restaurant">{{shoppingCart.restaurant.name}}</div>
    <ul class="list-unstyled">
      <li ng-repeat="cartItem in shoppingCart.items">
        <a ng-click="shoppingCart.remove(cartItem)">
          <i class="glyphicon glyphicon-remove"></i>
          {{cartItem.amount}} &times; {{ cartItem.name }}
        </a>
      </li>
    </ul>
    <div class="pull-right">
      <a href="#/checkout" class="btn btn-primary">Checkout</a>
    </div>
    <div class="fm-total">
      Total: {{shoppingCart.total() | currency}}
    </div>
  </div>
</div>
```

Extras

- Try adding “Clear Cart” button to menu page.

Step 12 - Create Checkout View

Finally we should provide a checkout view where the user can enter their payment details and submit the order.

Goals

Create a checkout route, which displays the **shoppingCart** and a payment form

Create and use a dummy **submitOrder(deliverTo)** method on the shoppingCart, which returns a promise to an order id.

Redirect on a successful order submission to a thank you view.

Tasks

12.1. Create a CheckoutController in a new checkout module, which uses the shoppingCart service

Create **js/checkout.js**:

```
angular.module('checkout', [])

.controller('CheckoutController',
    ['$scope', 'shoppingCart', 'customerInfo', '$location',
    function($scope, shoppingCart, customerInfo, $location) {

  if ( !shoppingCart.restaurant ) {
    $location.path('/');
  }

  $scope.shoppingCart = shoppingCart;
  $scope.restaurantId = shoppingCart.restaurant.id;
  $scope.deliverTo = customerInfo;
  $scope.submitting = false;

  $scope.purchase = function() {

    if($scope.submitting) return;
    $scope.submitting = true;

    shoppingCart.submitOrder($scope.deliverTo).then(
      function(orderId) {
        $location.path('thank-you').search({orderId: orderId});
      },
      function(err) {
        alert('There was a problem submitting your order');
        $scope.submitting = false;
      }
    );
  };
}]);
```

We use the **$scope.submitting** flag here to prevent a double submission.

Change **index.html**:

```
<script src="js/checkout.js"></script>
```

12.2. Create checkout route and add checkout.css styles to the index page

Change **js/app.js**:

```
angular.module('foodme', [
  'ngRoute', 'customer',
  'navigation', 'restaurants',
  'shopping-cart', 'checkout',
  'common/filters', 'common/fmDeliverTo'
])

.config(['$routeProvider', function($routeProvider) {
  $routeProvider
    ...
    .when('/checkout', {
      templateUrl: 'views/checkout.html',
      controller: 'CheckoutController'
    })
    ...
```

Change **index.html**:

```
<link rel="stylesheet" href="css/checkout.css">
```

12.3. Bind the CheckoutController to the checkout view

Change **views/checkout.html**:

```
<div class="col-md-6">
  <h4>Deliver To:</h4>
  <div>{{deliverTo.name}}</div>
  <div>{{deliverTo.address}}</div>
  <a href="#/customer-info">Change</a>
</div>
...
<tr ng-repeat="item in shoppingCart.items">
  <td>
    <input type="number" class="form-control" ng-model="item.amount">
  </td>
  <td>{{item.name}}</td>
  <td>{{item.price | currency}}</td>
  <td>{{item.amount * item.price | currency}}</td>
  <td>
    <a ng-click="shoppingCart.remove(item)">
      <i class="glyphicon glyphicon-remove"></i>
    </a>
  </td>
</tr>
<tr>
  <th></th>
```

```
    <th></th>
    <th>Total:</th>
    <th>{{shoppingCart.total() | currency}}</th>
    <th></th>
  </tr>
  ...
  <div class="row">
    <div class="col-md-12">
      <div class="pull-right">
        <a href="#/menu/{{restaurantId}}" class="btn btn-default">
          Back to Menu</a>
        <a ng-click="shoppingCart.reset()" class="btn btn-danger">
          Clear Cart</a>
        <button class="btn btn-primary">Purchase</button>
      </div>
    </div>
  </div>
```

12.4. Implement a submitOrder() method on the shoppingCart service

Change **js/shopping-cart.js**:

```
.factory('shoppingCart',
    ['localStorageBinding', 'alert', '$q',
    function(localStorageBinding, alert, $q) {
  ...
  submitOrder: function(deliverTo) {
    return $q.when(Math.floor(Math.random()*100000));
  }
  ...
```

We are simulating a post to a server here, and creating a random order id as a response. Since we want to return a promise for our response, we inject and use the **$q** service.

12.5. Bind the purchase()method to the Purchase button in the checkout view

Change **views/checkout.html**:

```
<button ng-click="purchase()" class="btn btn-primary">
  Purchase
</button>
```

Step 13 - Create a Thank You View

We have submitted our order and we have our dummy order id. All that is left is to display this to the use with a thank you message.

Goals

- Create a thank-you route, which extracts the order id from the URL query.

Tasks

13.1. Create a thank-you route, with a ThankYouController, which extracts the orderId from the route

Change **views/thank-you.html**:

```
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <div class="fm-panel">
      <div class="fm-heading">Thanks for the order</div>
      <div class="fm-content">
        <p>Our chefs are getting your food ready. It will be on its
way shortly.</p>
        <p>Your order ID is {{orderId}}.</p>
      </div>
    </div>
  </div>
</div>
```

Create **js/thank-you.js**:

```
angular.module('thank-you', [])

.controller('ThankYouController', ['$scope', '$routeParams',
function($scope, $routeParams) {
  $scope.orderId = $routeParams.orderId;
}]);
```

Change **js/app.js**:

```
angular.module('foodme', [
  'ngRoute', 'customer',
  'navigation', 'restaurants',
  'shopping-cart', 'checkout',
  'thank-you', 'common/filters',
  'common/fmDeliverTo'
])
...
.when('/thank-you', {
  templateUrl: 'views/thank-you.html',
  controller: 'ThankYouController'
})
```

Step 14 - Filter by Cuisine

We would also like to filter by cuisine.

Goals

- Create a new fmCheckboxList directive that allows us to generate a dynamic list of checkboxes from data and uses ngModel to bind the checkboxes to an array of chosen values.
- Use fmCheckboxList to filter the restaurants by cuisine

Topics

- NgModelController

Tasks

14.1. Create fmCheckboxList directive

Create **js/common/fmCheckboxList.js**:

```
angular.module('common/fmCheckboxList', [])

.directive('fmCheckboxList', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attr, ngModel) {
      // model -> view
      ngModel.$render = function() {
        var values = ngModel.$viewValue || [];
        angular.forEach(elm.find('input'), function(input, index) {
          input.checked =
            values.indexOf(input.getAttribute('value')) !== -1;
        });
      };

      // view -> model
      elm.on('click', function(e) {
        if (angular.lowercase(e.target.nodeName) === 'input') {
          scope.$apply(function() {
            var values = [];

            angular.forEach(elm.find('input'), function(input) {
              if (input.checked) {
                values.push(input.getAttribute('value'));
              }
            });

            ngModel.$setViewValue(values);
          });
        }
      });
    }
    };
  });
```

Change **index.html**:

```
  <script src="js/common/fmCheckboxList.js"></script>
```

Change **js/app.js**:

```
  angular.module('foodme', [
    'ngRoute', 'customer',
    'navigation', 'restaurants',
    'shopping-cart', 'checkout',
    'thank-you', 'common/filters',
    'common/fmDeliverTo', 'common/fmCheckboxList'
  ])
```

Create **test/unit/common/fmCheckboxList.spec.js**:

```
  describe('fmCheckboxList directive', function() {
    var inputs, scope, element;

    beforeEach(module('common/fmCheckboxList'));

    beforeEach(inject(function($compile, $rootScope) {
      scope = $rootScope;
      scope.filter = {};
      scope.OPTIONS = { a: 'A', b: 'B', c: 'C', d: 'D' };

      element = angular.element(
        '<div fm-checkbox-list ng-model="filter.cuisine">' +
          '<label ng-repeat="(name, title) in OPTIONS"
  class="checkbox">' +
            '<input type="checkbox" value="{{name}}"> {{title}}' +
          '</label>' +
        '</div>');

      document.body.appendChild(element[0]);

      $compile(element)(scope);
      scope.$apply();

      inputs = element.find('input');
    }));

    afterEach(function() {
      element.remove();
    });

    var triggerClickOn = function(elm) {
      var event = document.createEvent('MouseEvents');
      // https://developer.mozilla.org/en-
  US/docs/DOM/event.initMouseEvent
      event.initMouseEvent('click', true, true, window);
      elm[0].dispatchEvent(event);
    };


    it('should update the view on model change', function() {
      scope.$apply(function() {
        scope.filter.cuisine = ['a', 'b'];
      });

      expect(inputs.eq(0).prop('checked')).toBe(true);
      expect(inputs.eq(1).prop('checked')).toBe(true);
      expect(inputs.eq(2).prop('checked')).toBe(false);
      expect(inputs.eq(3).prop('checked')).toBe(false);
    });


    it('should update the model on view change', function() {
      triggerClickOn(inputs.eq(0));
      scope.$digest();
      expect(scope.filter.cuisine).toEqual(['a']);

      triggerClickOn(inputs.eq(2));
      expect(scope.filter.cuisine).toEqual(['a', 'c']);

      triggerClickOn(inputs.eq(0));
      expect(scope.filter.cuisine).toEqual(['c']);
    });
  });
```

14.2. Use fmCheckboxList to filter by cuisine

Change **js/restaurants.js**:

```
  $scope.cuisine = [];

  $scope.CUISINE_OPTIONS = {
    african: 'African',
    american: 'American',
    barbecue: 'Barbecue',
    cafe: 'Cafe',
    chinese: 'Chinese',
    'czech/slovak': 'Czech / Slovak',
    german: 'German',
    indian: 'Indian',
    japanese: 'Japanese',
    mexican: 'Mexican',
    pizza: 'Pizza',
    thai: 'Thai',
    vegetarian: 'Vegetarian'
  };
  ...
  function filterRestaurants() {
    $scope.filteredRestaurants = [];
    angular.forEach($scope.restaurants, function(restaurant) {
      if ( ( !$scope.rating || restaurant.rating >= $scope.rating ) &&
           ( !$scope.price || restaurant.price <= $scope.price ) &&
          ( !$scope.cuisine.length ||
              $scope.cuisine.indexOf(restaurant.cuisine) !== -1) )
      {
        $scope.filteredRestaurants.push(restaurant);
      }
    });
  }

  $scope.$watch('restaurants', filterRestaurants);
  $scope.$watch('rating', filterRestaurants);
  $scope.$watch('price', filterRestaurants);
  $scope.$watchCollection('cuisine', filterRestaurants);
```

Change **views/restaurant-list.html**:

```
<h5>Cuisine</h5>
<div fm-checkbox-list ng-model="cuisine">
  <div ng-repeat="(name, title) in CUISINE_OPTIONS">
    <label class="checkbox">
      <input type="checkbox" value="{{name}}"> {{title}}
    </label>
  </div>
</div>
```

Change **test/unit/restaurants.js**:

```
it('should filter by cuisine', function() {
  expect(scope.filteredRestaurants.length).toBe(5);

  scope.$apply(function() {
    scope.cuisine = ['german'];
  });

  expect(scope.filteredRestaurants.length).toEqual(1);
  expect(scope.filteredRestaurants).toEqual([
    jasmine.objectContaining({id:'esthers'})
  ]);

  scope.$apply(function() {
    scope.cuisine = ['african', 'german'];
  });

  expect(scope.filteredRestaurants.length).toEqual(2);
  expect(scope.filteredRestaurants).toEqual([
    jasmine.objectContaining({id:'esthers'}),
    jasmine.objectContaining({id:'khartoum'})
  ]);
});
```

Step 15 - Pretty Rating Chooser Directive

The original mock ups had interesting widgets for filtering on rating and price. Up to now we just used a select box instead. Let's implement this more attractive widget.

Goals

- Create fmRating directive, on which we can specify the symbol to display, whether it is readonly and the maximum rating allowed.
- Use fmRating to select the rating and the price for filtering restaurants

Tasks

15.1. Create fmRating directive

Create **js/common/fmRating/fmRating.js**:

```
angular.module('common/fmRating', [])

.directive('fmRating', function() {
  return {
    restrict: 'E',

    scope: {
      symbol: '@',
      readonly: '@',
      max: '@',
      rating: '='
    },

    link: function(scope, element, attrs) {

      scope.$watch('max', function(value) {
        // Recreate the array of styles for each rating symbol
        var max = parseInt(value || 5, 10);
        scope.styles = [];
        for(var i = 0; i < max; i ++) {
          scope.styles.push({ 'fm-selected': false, 'fm-hover': false
});
        }
      });


      scope.enter = function(index) {
        // Mouse is over a symbol so update the styles
        if (scope.readonly) return;
        angular.forEach(scope.styles, function(style, i) {
          style['fm-hover'] = i <= index;
        });
      };

      scope.leave = function(index) {
        // Mouse is no longer over a symbol so update the styles
        if (scope.readonly) return;
        angular.forEach(scope.styles, function(style, i) {
          style['fm-hover'] = undefined;
        });
      };


      scope.select = function(index) {
        if (scope.readonly) return;
        scope.rating = (index === null) ? null : index + 1;
      };

      scope.$watch('rating', function(value) {
        updateSelectedStyles(value - 1);
      });


      function updateSelectedStyles(index) {
        if (index === null) index = -1;
        angular.forEach(scope.styles, function(style, i) {
          style['fm-selected'] = i <= index;
        });
      }

    },

    templateUrl: 'js/common/fmRating/fmRating.template.html'
  };
});
```

Create **js/common/fmRating/fmRating.template.html**:

```
<ul class="fm-rating" ng-class="{'fm-rating-pointer': !readonly}">
  <li ng-repeat="style in styles"
      ng-class="style" ng-click="select($index)"
      ng-mouseenter="enter($index)" ng-mouseleave="leave($index)">
    {{symbol || '\u2605'}}
  </li>
</ul>
<a ng-hide="readonly" ng-click="select(null)">clear</a>
```

Change **index.html**:

```
<script src="js/common/fmRating/fmRating.js"></script>
```

Change **js/app.js**:

```
angular.module('foodme', [
  'ngRoute', 'customer',
  'navigation', 'restaurants',
  'shopping-cart', 'checkout',
  'thank-you', 'common/filters',
  'common/fmDeliverTo', 'common/fmCheckboxList',
  'common/fmRating'
])
```

15.2. Use fmRating directive in menu and restaurant list views

Change **views/menu.html**:

```
<div>
  <fm-rating rating="restaurant.rating" readonly="true"></fm-rating>
</div>
<div>
  <fm-rating rating="restaurant.price" symbol="$" readonly="true">
  </fm-rating>
</div>
```

Change **views/restaurant-list.html**:

```
<div class="fm-content">
  <h5>Rating</h5>
  <fm-rating rating="rating"></fm-rating>
  <h5>Price</h5>
  <fm-rating rating="price" symbol="$"></fm-rating>

  ...

<tr ng-repeat="restaurant in filteredRestaurants">
  <td>
    <a href="#/menu/{{restaurant.id}}">
      <img class="img-rounded pull-left"
           ng-src="img/restaurants/{{restaurant.id}}.jpg">
        <b>{{restaurant.name}}</b>
    </a>
    <p>{{restaurant.description}}</p>
  </td>
  <td>
    <fm-rating rating="restaurant.rating" readonly="true"></fm-
rating>
  </td>
  <td>
    <fm-rating rating="restaurant.price" symbol="$" readonly="true">
    </fm-rating>
  </td>
</tr>
```