

# Intro to AngularJS Lab

Now that we've got an idea of what Angular can do, let's install it as well as the other parts of its ecosystem.

## Installing NodeJS & npm

1. If Node is already installed you can skip this section. Look at the "Testing Node" section below to test if it is installed.
2. Open a browser and go to <http://nodejs.org>
3. Hit the big install button in the middle of the page. Allow it to download and run the installation program. Follow the prompts through.

## Testing Node

4. Open a bash window and type in "node". You'll see a command prompt.

5. Type in:

```
var x = "hello";  
console.log(x + " world");
```

It is normal to see "undefined" a few times. But you should see "hello world".

If so, NodeJS is installed and ready to go.

6. Hit ctrl-C to exit Node.

## Installing MongoDB

7. If Mongo is already installed you can skip this section. Look at the "Testing MongoDB" section below to test if it is installed.
8. Go to <http://mongodb.org>
9. Choose your OS and version and hit the download button. No need to join anything or provide any personal information.

10. Make a directory like this:

```
mkdir -p /usr/local/bin/mongodb
```

11. Unzip all the downloaded files to your new mongodb directory.

12. Put mongo in your path in .profile:

```
PATH=$PATH:/usr/local/bin/mongodb/bin
```

13. And make the data directory:

```
mkdir -p /data/db
```

That should be it! Let's test it out.

## Testing MongoDB

14. Open a new command prompt and type in *mongod*

It should show a bunch of messages to the command line and finally say that it is waiting for connections on a particular port. Keep it running in that window at all times throughout these labs.

15. In another window, type in *mongo*

16. From the Mongo prompt, type in *show dbs*

17. It should say there's a local database and give you a size. Verify that it does.

If these steps work, you've got MongoDB installed and running.

## Downloading and installing the lab files

18. Your humble instructor will give you the lab files on a USB drive or web site. Get that file and unzip it to any directory you like. This will be your lab project directory. Write it down here: \_\_\_\_\_
19. Notice that there is a directory called 'setup' with scripts to load your database with data and to download/install all the library files we'll need for the labs.
  - Get to a bash prompt in your project directory
  - `cd` to setup/bash
  - Run the script like this:  
`./installLab.bash`

## Testing the Lab Install

20. Let's make sure it worked. Do this from a command prompt:

```
mongo northwind
db.employees.find()
```

You should see some employees. If so, you're on the right track.

21. If you go back to the root of your project directory, you'll see a folder called *node\_modules*. Do a listing in that folder. You should see a few sub-folders like *express*, *passport*, and *gulp*.
22. Go back to the root and into the *app* folder. Hey, look! There's another *node\_modules* folder. It should have *bootstrap*, *jquery*, and *angular*.
23. Lastly, open a command prompt, `cd` to your lab project directory and go:  
`node northwindServer.js`  
It'll tell you that your web server is listening on a particular port.
24. Open your favorite browser (Chrome, Firefox, Edge, whatever) and point it to that address.
25. You should see the landing page of the web site. Have a look around the site. The data is all hardcoded, but the site should look good.

If all those things check out, we are ready to go. Let's learn how Angular works!

## Quick Start Lab

Let's see if we can get Angular working in our project.

1. Add a folder under app called "main".
2. Add a view in your app/main folder called "aboutUs.html". The contents aren't important as long as you have the following hardcoded data points in there:
  - Company name
  - Your company street address, city, region/state/province, zip/postal code
  - Your company phone number
  - A company contact email
  - Social media links (like Facebook, Twitter, LinkedIn, YouTube, etc).
  - Today's date
3. Add this in the <head> of your page:  
`<script src="/app/node_modules/angular/angular.js"></script>`
4. Make the body tag say:  
`<body ng-app="mainModule">`
5. Load the page and open developer tools to the Network or Sources tab and ensure that angular.js is being read.
6. Find all your data points from above and change the hardcoded values to Angular expressions. (Hint: use mustaches like so: `{{company.name}}`, `{{company.streetAddress}}`, `{{company.city}}` and so forth).
7. Reload the page. You should still see the mustaches.

### Adding a module

8. Create a new file in the /app/main directory called *mainModule.js*. In it, create a module like so:

```
var mod = angular.module("mainModule", []);
mod.controller("aboutUsController", function ($scope) {
  console.log("In the controller");
});
```

9. Add a script tag to the <head> of your page pointing to it.
10. Run the page and look at the console. Do you see your message? No? Good. You shouldn't. You haven't used that controller yet!

### Using the controller

11. Add `ng-controller="aboutUsController"` to the <body> tag.
12. Re-run and look at the console. You should see the message.
13. Now go back into the controller and create a JSON object like so:  
`$scope.company = { phone: "867-5309" };`
14. Re-run. Your phone number in the view should be populated from the controller.
15. Do the same for all the other properties of your made-up company object.

16. Add today's date like so:

```
$scope.today = new Date();
```

17. Run and test. If you like your new page, you can be finished. If not, change its format until you're happy.

## **Bonus!! Formatting the date**

18. If you have time, you may notice that the date is ugly. Let's fix that with a filter.

19. Find where the view is putting the date on the page. It should be in mustaches.

20. Inside the mustaches, add a pipe symbol with the word "date" after it. Like so:

```
{{ today | date }}
```

21. Run and test. Better?

22. Try a couple of these:

```
{{ today | date:"fullDate" }}
```

```
{{ today | date:"longDate" }}
```

```
{{ today | date:"shortDate" }}
```

```
{{ today | date:"mediumDate" }}
```

# Intro to Directives Lab

## Working with ng-app and ng-controller

1. In your *app* folder, you have a file called *index.html*. Edit it.
2. Make sure it has two attributes in the `<body>` tag, one for `ng-app="mainModule"` and one for `ng-controller="mainProductController"`.

These will wire up Angular to be in control of the page. You may already have the `mainModule`, but you probably don't have the `mainProductController`, so let's create that.

3. open your main module and add a new controller to the module called `"mainProductController"`. Just like before, have it `console.log()` something so we can be sure it is running.
4. Run and test until you see your console message. Do what you must to get it running.

## Working with ng-repeat

We've decided that in order to push certain products we're going to feature them on our front page. We'll eventually pull them from a database in our controller but we're not ready for that yet. We'll simulate it using a JSON array of products.

5. Look in the `/assets/js` directory for a file called `products.json.js`. Open it in an editor.
6. Copy the entire array and paste it in your controller.
7. Get the array into `$scope.topProducts`.

If they're part of `$scope`, we know they'll be available in the view. And since they're in an array, we should loop through them using `ng-repeat`.

8. In `index.html`, find where we should be displaying four products and replace them with this:

```
<div ng-repeat="product in topProducts" class="item text-center"
  ng-class="{ 'active': $first }">
  <a ng-attr-href="product/index.html/#{product.productID}">
    {{product.productName}}</a>
  </div>
```

9. Run and test. You should be able to see the products you put in the array.
10. Bonus! Note that the `ng-attr-src` could have been written as `ng-src` and that `ng-attr-href` could have been shortened to `ng-href`. Feel free to change them if you like.

## Adding a map

In case our customers want to find our brick-and-mortar location, let's give them map to us.

11. Open the about us page and add a `<section>`. In this section, put a `<h3>Map of our headquarters</h3>` and an `<img src="" />`
12. Set the src of the map to "  
`https://maps.googleapis.com/maps/api/staticmap?markers={{ company.street }} {{ company.city}} {{ company.region }} {{ company.postalCode }}`  
`&size=600x600&maptype=hybrid"`
13. Run and test. Look at your console and/or network panels. You should have some errors like 404s.

Those errors might worry us during testing. They're caused by the src which has mustaches in it. This won't be hidden from the browser unless we change it to `ng-attr-src`. That directive signals Angular to replace the mustaches with model values.

14. Change the src attribute to `ng-attr-src` and refresh the page.

Once you can see your map and have no http errors, you can be finished.

# Controllers Lab

At this point you have some good code written but it may not be organized in the best way. We should implement the best practices we learned in class.

## Creating the project structure

A large project will have multiple modules and a shared module. Might as well prepare ourselves.

1. Review the project directory structure. Make sure you have a *main* folder and a *shared* folder under "app".

## Splitting the module and controllers

2. You've currently got a file called "mainModule.js" with your module creation and at least two controllers. Split that into as many other files as you'll need. Here's a suggestion:
  - mainModule.js
  - mainProductController.js
  - aboutUsController.js
3. Make sure mainModule only contains the definition/creation of the module. It has no dependencies at this point.

mainProductController.js should have nothing but the controller in there. Same with aboutUsController.js.

4. You'll need to add references to these new files in your HTML files. Go ahead and do that.
5. Run and test everything. If it still works, you can proceed.

As you work with these files throughout these labs, make sure you're putting the right things in the right files. Just remember the Single Responsibility Principle--each file should contain only one thing (one controller, one module, etc).

## Encapsulating the functions

If we're not careful we could end up leaking objects onto the global namespace and get ourselves in trouble as the project grows. Let's make 100% sure that can't happen with IIFEs.

6. Edit each of your module/controller files and wrap each function in an IIFE.
7. Make sure all variables are declared with the *var* keyword, otherwise our IIFE won't do us any good.
8. Once again, run and test. Make sure we haven't broken anything.

## Services Lab

There are some activities which we may need on multiple pages and even in multiple sections of the site. Whenever we want to share activities, we should probably make a service out of them and put them under a shared folder.

1. If you don't already have a shared directory, create it.
2. Add a `sharedModule.js` and create the *sharedModule* ... uh ... module.

### Creating a category service

3. Create a new file under *shared* called `categoryService.js`. In it create a service that is part of the shared module.
4. The service should expose one function called `getAllCategories()`
5. `getAllCategories()` should return an array of category objects in JSON format. Feel free to copy those categories from your `productListController`.

### Using and running a service

6. To use the service you must include it. Open `productList.html` and add a link to this new script as well as the shared module.
7. Open your `productModule.js` file and add `sharedModule` as a dependency.
8. You'll also need to include it in the controller. Open the `productListController.js` and add `categoryService` as a dependency. Remember, it will need to be passed into the function as a parameter. You'll probably going to want to include it in the dependency array as a string to protect from minification.
9. Still in `productListController`, change the hardcoded category array to call your new service. Something like this might do the trick:  

```
$scope.categories = categoryService.getAllCategories();
```
10. Run and test. Make sure your list of categories still populates like before.

Now your controller is much cleaner and the list of categories is available to other modules and controllers. It only needs to be changed in one place ... the service.

### The productService

Looks like we have some more low-hanging fruit. Our product list is also repeated. Let's re-do the above steps with the product lists.

11. You can re-use the `sharedModule`. But create a new service called `productService`. Add all the trappings that will make this a service. You should know what to do by now. Go for it.
12. Go ahead and create these methods in `productService`:
  - `getAllProducts()` - Returns an array of all products



- `getFeaturedProducts()` - Returns all products for now. We'll change this in later labs
  - `getProduct(productId)` - Returns one product. You'll want to write it so that it searches through your array of products and only returns the one whose `productId` matches the one passed in.
13. Once these are all created, they can be used in `productListController`, in `productSearchController`, and on the main page, `index.html`.
  14. Find everywhere you're hardcoding a list of products in controllers. Change those to call `getAllProducts()`. Except in the `mainProductController`. Make that one call `getFeaturedProducts()` instead.
  15. Run and test. Make sure that your products appear as expected on all pages with products.

Once your product search page, product browse page, and main page are working properly you can be finished.

### **Bonus! Let's do one more**

If you have time, there's another opportunity for improvement. Notice that in the `shipToController`, you're still using a hardcoded local customer object. Change it to use a shared `customerService(customerID)` that will return a customer object given a `customerID`.

# Ajax with AngularJS Lab

Up to now we've been simulating reading from a server by using hardcoded files. In this lab we'll improve that. We're going to consume a web service with a database server behind it.

## Starting up the servers

1. Make sure MongoDB is running. You'll want to start the mongod server. Feel free to refer back to the first lab instructions to remind yourself of how to do that.
2. Start a nodeJS web server:  

```
cd <your student_labs folder>  
node northwindServer.js
```

A message will tell you on which port the server is running.
3. Point a browser at that site to ensure that it is working.
4. Get any of your web pages by browsing to `http://localhost:XXXX/app/YYYY.html` where XXXX is the port and YYYY is the page you want.

## Testing the RESTful requests

Getting pages is one thing but getting data is another. Let's make sure we can get data from the server now.

5. Open your favorite browser and point it to your site. Navigate to `"http://localhost:XXXX/api/product"`. You should see all of the products as JSON data.
6. Do the same with `"http://localhost:XXXX/api/product/featured"`. You should see only the products in the database marked as "featured".
7. Copy/paste one of the productIDs. Browse to `"http://localhost:XXXX/api/product/<the productID>"`. You should see one product as JSON data. Try two or three other IDs.
8. Try the same thing with category.

These are the RESTful urls we can hit to populate our pages. Let's do that now.

## Get all the products!

Now that we know that the server side is running, the hard part is over. Let's try to read some data from it in Angular.

9. Open your ProductService.js file and add a dependency to \$http.

The `getAllProducts()` function is returning a hardcoded array of products. Let's change that to make an Ajax call to the server using `$http()`.

You are currently doing something like this:

```
var allProducts = [ { JSON object }, { another JSON object } ... ];  
return allProducts;
```

10. Change it to do something like this:

```
var promise = $http({url: '/api/product'});  
return promise;
```

Note that we changed what is coming back from the method. We're no longer returning a list of products. We're now return a *promise*. So obviously you'll need to change all controllers that use this service to handle the promise instead of the list.

11. Open productListController.js and find where we're calling getAllProducts().

You are currently doing something like this:

```
$scope.products = productService.getAllProducts();
```

12. Change it to do something like this:

```
productService.getAllProducts().then(function (response) {  
    $scope.products = response.data;  
});
```

13. Run and test by browsing to the productList.html page. You should see all the products from the database instead of your hardcoded products.

14. Once you've got that figured out, do the same for productSearchController. It'll be pretty much the same steps.

15. By now you're a pro! So here's a challenge. In the mainProductController you're calling productService.getFeaturedProducts(). Your mission is to change getFeaturedProducts() to make a RESTful call to '/api/product/featured'.

## Now the categories

You've converted all the product reading to well-designed RESTful calls. Nice. Let's do the same for categories.

16. Open the categoryService.js and change getAllCategories() to make a GET call to /api/category. Make it return the promise.

17. getAllCategories() is being called from productListController. Change it to process the promise rather than directly reading an array of categories.

Once you're reading real categories and real products from the RESTful services, you can be finished.

18. Bonus! If you have time and you did the bonus from the Services Lab, open your customerService and consume a customer object served from /api/customers/<customerID>. (Hint: try "BOLID" or "ALFKI" as example IDs).

# Capsules Lab

We know that any activities which we may be needed on multiple pages should probably be made a capsule and put under a shared folder. We did that with some services earlier. In this lab we'll get practice with making a factory.

## Creating a cart factory

1. Create a new file under *shared* called *cartFactory.js*. In it create a factory that is part of the shared module. Remember that a factory returns a JSON object. Make sure yours does so.
2. Load that returned JSON object with an empty array called *cart* and these methods:
  - `addToCart(product, quantity)`
  - `removeFromCart(product)`
  - `getCartTotal()`These can be method stubs\* for now that do nothing more than `console.log()` that they've been called.

## Making the factory work in controllers

They're not doing much yet but let's see how they'd work on a page.

3. Edit *productDetails.html* and wire it up for Angular.
4. Change all the hardcoded values to Angular expressions like `{{ product.productName }}` and `{{ product.unitPrice }}`
5. Create *productDetailController.js*. Add a dependency to *productService* and one to your new *cartFactory*.
6. Hardcode a variable called *productId* to any number between 1 and 50. Just pick one.
7. Make a call to `productService.getProduct()` so you can get the product data from the RESTful service.
8. In the success callback, populate `$scope.product`.
9. Add an `ng-click` directive to the *AddToCart* button, calling `cartFactory.addToCart()`; Don't forget to pass in the product and quantity from the view.
10. Run and test. Make sure it is console.logging the right product and quantity.
11. Open *checkoutController.js*. Also make it depend on *cartFactory*.
12. When *checkoutController* loads, you're populating `$scope.cart`. Change that to do this:

```
$scope.cart = cartFactory.cart;
```

This will allow us to tie the `$scope`'s cart to the shared cart.

---

\* A method stub is a function that exists, but doesn't really do anything. Developers use these as placeholders for real functionality later.

13. Reload and make sure it also is console.logging.
14. Find the `<a>` tag where we're removing from the cart. Make its ng-click action make a call to `cartFactory.removeFromCart` passing the product.

See how this is just the same as a service?

## Persisting the cart

Well, that was fun, wasn't it? We've created the factory and we're calling factory methods. But the methods don't do anything but console.log. Let's fix that by making use of a RESTful api service.

15. Open `cartFactory.js` in your editor.
16. In the main `cartFactory` function, create a private reference to the factory itself like this:

```
var self = this;
self.cart = [];
```

17. Then check to see if there is anything in `self.cart`. If not, get it from the RESTful API like so:

```
if (! self.cart.length)
  $http({url: '/api/cart'}).then(function (res) {
    res.data.forEach(function (line) {
      self.cart.push(line);
    })
  });
```

18. Lastly, locate where we're returning the object from the factory and instead of setting `cart` to an empty array, set it equal to `self.cart`.
19. `addToCart(product, quantity)` should make a POST request to `/api/cart`. It should pass a JSON object in the body with the product and the quantity. It should also push that product/quantity on to the `self.cart` array. It should return the promise. (Hint: a `.then()` also returns a promise).
20. `removeFromCart(product)` should make a -- you guessed it -- ajax DELETE call to `/api/cart`. It should pass a JSON object in the body with the product just like with `addToCart`. It also should remove that product from the `self.cart` array and return the promise. Note that since HTTP DELETES don't normally have a body, you'll have to add this to your request:

```
headers: {"Content-Type": "application/json;charset=utf-8"}
```

21. Run and test the checkout and productDetail pages, adding and removing quantities from the cart. Keep adjusting until it works.

## Bonus! Handling messages

22. You'll want to make adjustments to `checkoutController.js` to implement the callback to `getCart()`. (Hint: create a `.then()` and display a message of some kind). Same with the callback to `removeFromCart()`. Take care of both of those.
23. Do the same with `productDetailController.js`. When the `addToCart()` method is run, you may want to produce a message in a `.then()`.

## Templates Lab

It makes sense to reuse code rather than duplicate it. HTML pages are no exception. We should do that with sections that are common to more than one page. Let's start with the footer.

1. Surf through your site. Notice that the header is exactly the same on every page. And, hey! The footers are, too. Repetition is not good. Let's fix that.
2. Edit your main page, index.html.
3. Go to the bottom of the page and find the footer. Cut it and replace it with this:

```
<footer class="row" ng-include="'/app/shared/pageFooterPartial.html'">
```

4. Now paste it into a new file called pageFooterPartial.html.
5. Run and test. You should still be able to see the page footer.
6. Now reuse this page footer in all other pages on your site by using the ng-include directive.

### Replacing the header

You did it with the footer. Now let's do the same with the header.

7. Start with index.html. Find the bounds of the header and copy them into a new file called pageHeaderPartial.html.
8. Use ng-include to include pageHeaderPartial.html. But this time, also give it a controller called pageHeaderController.js.
9. The pageHeaderController doesn't need to do anything yet except console.log so you can see it running.
10. Do the same for all pages on your site and test them all out. You should see no change except the console.log() messages but now your pages are simpler.

### Actions in a template

Now let's make the pageHeader do something much more useful. Let's show the user how many products they have in their cart at all times.

11. Open pageHeaderPartial.html. Notice that it has a link to the checkout page with a cart icon and a badge. That badge is supposed to have a number that shows how many products are in the cart. Right now it only has a hardcoded "0" in it.
12. Change that "0" to an expression with the value of cart.length.

If you run it now, it won't work because \$scope.cart isn't a thing yet.

13. Give pageHeaderController a dependency on cartFactory. Remember that factories are singletons so if we set the cart length anywhere, it'll be reflected here as well.
14. At the top of the controller function, go  
`$scope.cart = cartFactory.cart;`
15. That should do it! Go through the exercise of adding things to your cart and removing things from your cart. That badge will change as you change the cart contents.

Once all your pages are sharing a common header and footer, you can be finished.

### **Bonus!! Totalling the cart**

You may have noticed that the cart has a subtotal but it isn't doing anything yet. We'll fix that in this bonus exercise.

The subtotal should be displayed in the view in an expression, populated in the controller based on the cart object, and calculated in the cartFactory. Let's do it!

16. Edit checkout.html. Find where the subtotal is being displayed and change it to an expression reflecting `cart.subtotal`.
17. Edit cartFactory.js. Add a method called `calculateSubtotal()` which will loop through the lines of the cart and sum up each `unit price * quantity`. Set `self.subtotal` to this value.
18. Each time we change the cart's contents this number should be recalculated. So call `calculateSubtotal()` in `AddToCart()`, `removeFromCart()`, and `checkout()`.
19. Run and test this. You should see that the total always reflects the current contents of the cart.

## SPAs Lab

Let's say we've noticed that the productSearch page, the productList page, and the productDetail page all have some things in common; they deal with products and they have the same layout. We can take advantage of some efficiencies by combining all of these into a single page.

### Converting the shell page

One of the three needs to be the shell page. We'll arbitrarily pick productList.html.

1. Copy productList.html to index.html. (Hint: Make sure you're in your product subdirectory so you don't clobber your real index.html!)
2. Open this index.html and add this script tag to the document's <head>:

```
<script src="/app/node-modules/angular-route/angular-route.js">
</script>
```

3. Locate the content section. (Hint: That's the stuff that is different from the other two pages.)
4. Delete all of the content section. Make sure to leave the parts that are common to productSearch.html and productDetail.html.
5. Where the content used to be, make it say this:

```
<section ng-view></section>
```

The ng-view is the part that will make this work. If you were to test this out now, it wouldn't work because we haven't told Angular which partial to use in that view. That's next.

6. Rename productList.html to productListPartial.html. Remove all the parts that are already in its shell page (index.html from above).

### Setting up the routing

We need routing for a SPA but we haven't covered all the details yet. We'll explain more what all this stuff means later. Routing must be set up in a config.

7. Edit productModule.js. Add a new dependency to ngRoute and add a config. Do something like this in it:

```
angular.module("productModule", ['ngRoute', 'sharedModule'])
  .config(function ($routeProvider) {
    $routeProvider
      .when('/search', {
        controller: 'productSearchController',
        templateUrl: "/app/product/productSearchPartial.html",
        caseInsensitiveMatch: true
      })
      .when('/browse', {
        controller: 'productListController',
        templateUrl: "/app/product/productListPartial.html",
        caseInsensitiveMatch: true
      })
  })
```



```
.when('/', {  
  controller: 'productDetailController',  
  templateUrl: "productDetailPartial.html",  
  caseInsensitiveMatch: true  
});
```

8. As a test, you should be able to navigate to `<yourSiteRoot>/product/index.html#/browse` and see your list of products. Give it a try.

## Converting the other pages

Once our browse page is working, the other pages should be pretty simple to convert.

9. Rename `productSearch.html` to `productSearchPartial.html`.
10. Delete everything except the unique content just like you did with the `productListPartial.html` page.
11. Now do the same with `productDetail.html`.
12. Run and test. You should be able to see both of these pages at `<yourSiteRoot>/product/index.html#/search` and `<yourSiteRoot>/product/index.html#/.`

## Examining the architecture

13. In your browser, open the developer tools. Focus on the Net/Network tab.
14. Browse to `<yourSiteRoot>/product/index.html`. Look at the traffic in the tool.
15. Browse to `<yourSiteRoot>/product/index.html#/search`. Look at the traffic. You should see all the JavaScript, CSS, and HTML files loaded.
16. Now Browse to `<yourSiteRoot>/product/index.html#/browse`. What was loaded this time?

---

What you should see is that only the content was refreshed in the browser. This saves us strain on our server, traffic on our wires, UX time in reloading, and will provide a better experience for our user. Everyone wins!

## Routing Lab

In the SPAs lab we learned to create single-page apps. Let's expand that as we create route parameters and have a failover in case users request a bad address.

### Getting a single product

1. Currently your route to the productDetail view is probably at "/". Make a copy of that route. On the copy, add a route parameter for ':productId'. Don't forget the ":". This will allow us to scrape a productId from the URL.
2. Now let's read and use that parameter. Edit your productDetailController and add a dependency on \$routeParams. \$routeParams is a built-in provider that allows us to read any route parameters in our controllers.
3. Find where you're hardcoding the productId variable. Change it to read the productId from \$routeParams.
4. Run and test. Adjust until you're able to see one product at the location `http://<yourRoot>/app/product/index.html/#/XXXX` where XXXX is a product id. Any valid product ID should work. Try several. Each should show a different picture, description, and all other product details.

### Reordering the routes

5. Try to bring up the productList and the productSearch pages. Depending on where you have your new route, you'll either see these pages or they'll fail. If they've failed, see if you can fix the issue. (Hint: routes are evaluated top-down. The first matching route hit will be used and all others below it are ignored.)
6. Once that is fixed, you should also be able to click on any product in the productList and the productSearch pages and see the details for that product. Click on a few to make sure it works.
7. Bonus! The "/" route is now pointing to productDetail. Make it point to productList.

### Creating a catch-all

8. Type a nonsense url into the address bar in your browser like `http://<yourRoot>product/index.html/#/some/bogus/address`. What do you see? \_\_\_\_\_
9. Instead, when the user types in nonsense, let's have him/her go to the product list. Add an otherwise route that sends the user to the productList.
10. Try your nonsense url again. In fact, try several. Make sure they all take you to the product list.

## Getting a querystring

This last part has nothing to do with routing, but we really should know how to get querystrings. On our search page, we currently allow the user to type a searchString into the input box. We then use that searchString to limit our list via a filter. Let's improve on this. Let's allow the user to pass an initial searchString to our page via a querystring. Then we'll read that querystring in the controller and apply the filter immediately.

11. Edit the productSearchController. If you don't already have a dependency to \$route, add one.

12. Read the queryString into the model like so:

```
$scope.searchString = $route.current.params.searchString;
```

It should now be available to the view immediately when it is run.

13. Test it out by navigating to

`http://<yourRoot>/app/product/index.html/#/search?searchString=queso`

You should see only those products with 'queso' in the description.

14. Now, if you'll notice, in our pageHeaderPartial.html, the search form makes a GET request to "/search" when submitted. This means that it will tack on a searchString. So by reading \$route.current.params.searchString above you just made the search box work on every page. To test that, go to any page on the site and type in a search term. Make sure that works.

Once it does, you can be finished.

## Intro to Custom Directives Lab

Better watch out! Our UX experts and marketing types have been talking. This always means more work for developers. And sure enough, they've come up with a way to increase sales and to make it easier on our users. Who couldn't get behind that? They've gotten feedback that our users would love to add products to their cart directly from the search results and from the browse page. As of now users must click through to the product details page to add to the cart. So let's roll up our sleeves and get to work!

1. Open the `productSearchPartial.html` and `productListPartial.html` files in your IDE. Note that they both present lists that are identical.

This is not DRY. A template sounds like a good idea at first. But as you re-think, you remember that we're now adding behavior as well as presentation. So a custom directive would be a better idea. Let's do that! We'll start with the presentation which is a template.

### Creating the template

2. Copy the inner contents of the `ng-repeat` into a new file called `nwProductOverviewPartial.html`. (Note: you don't want the `ng-repeat` div itself, just the things inside of it.)
3. Replace those lines in `productSearchPartial.html` and in `productListPartial.html` with this:

```
<nw-product-overview></nw-product-overview>
```

### Creating the custom directive file

4. Create a new file called `nwProductOverviewDirective.js`
5. Don't forget to...
  - Use an IIFE,
  - Add a reference to the `productModule`,
  - Call the `".directive()"` function, passing the name of the directive (`nwProductOverview`) and the directive function,
  - Return a DDO from the custom directive which has ...
    - a `restrict` property
    - a `template` property that says `"<p>product overview</p>"`
    - a `controller` property pointing to a function with one line:  
`console.log("product overview directive controller")`.
6. Include the `nwProductOverviewDirective.js` file in a `<script>` tag.
7. Run and test. Eliminate any JavaScript errors you see in the console.
8. At this point you should see the directive being repeated. The page should say "product overview". You should also see "product overview directive controller" in the console once for each item.

## Showing the actual product

9. Open the directive function again. Change your template property to be a templateUrl property and point it to your nwProductPartial.html file.
10. Run and test. You should now see all of your products just as before.

## Adding the new functionality

Let's add the input box for them to specify quantity and a button to add it to the cart.

11. In your directive's partial, add something like this:

```
<div class="col-sm-2">
  <input ng-model="quantity" type="number" class="form-control" />
  <button ng-click="addToCart(product, quantity)" class="btn btn-
primary btn-sm glyphicon glyphicon-shopping-cart"></button>
</div>
```

All those classes are there just to make it look good.

12. Bonus: Change the numbers in col-sm-\* to add up to 12 and they'll all fit in on a row.
13. Open your productDetailController.js and find the addToCart function. Copy it and paste it in your directive's controller.
14. Note that it requires a few services like \$scope and cartService. Make sure to add those to your controller as injected dependencies.
15. Run and test.

Your users should now have the ability to add products to their cart directly from the browse page, the search page, and the details page.