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}}">
    <img
    ng-attr-src="/assets/img/productImages/{{product.productID}}.jpg"
    class="center-block"
    ng-attr-alt="{{product.productName}}">{{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
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.

Naming the functions

This step helps with clarity of code and also in debugging the call stack. Currently we're passing anonymous functions into the module.controller() function as the second argument. Let's give them names.

9. Pull each function out of the controller() parameter list and give each a name like so:

```
function controllerName() { /* All your logic goes here */ }
```

10. Then put the controllerName as the second argument in your controller function parameter list.
11. Run and test again. Still working? Good!

Protecting from minification renaming

As we grow we're going to want to start minifying our code but if we do, our controller dependencies are going to fail. We'll fix that with explicit dependency declarations.

12. In both of the controllers, add an array as the 2nd argument between the two existing ones. Sort of like this:

```
angular.module('moduleName')
.controller('controllerName', ['$scope', 'otherDep', ctrlFuncName]);
```

13. Of course if you don't have a second dependency, you won't have 'otherDep' in the example above. So you may not need that second argument.
14. You know the drill by now. Run and test.

Adding a dependency

Just to get practice, we're going to add a dependency. You won't really need $http, but let's see how we would implement that dependency.

15. Open mainProductController.js. Add $http as an argument in the function declaration.
16. Now you'll have to match that in the explicit dependency array. Look at the array and add $http there as well. Make sure it is a literal string by putting it in quotes. And double-check that it is in the right place.
17. Run and test. Make sure everything still works after your refactor.

Views Lab

Improving performance

We're currently using binding in several places. This is great except when we know that those elements will never change. Seriously, do we expect the company address to change during the run of the program?

1. Open the aboutus.html page.
2. Find all places where we are referencing the company address. Change every one to a one-time binding. (Hint: Use {{ ::variable }}).
3. Run and test.
4. Do the same on the index page with the featured products. Those may change, but not while the page is loaded in the browser.

Wiring up the checkout page

Let's work with the page that will be used to checkout the user. It'll have a review of their cart contents, their name, address, and a place for a credit card.

5. Find the *ordering* folder. The separate folder is hinting that it might be a good idea to create a separate module.
6. Create orderingModule.js. In it put the declaration for a new module called orderingModule. Don't forget your IIFE and the other best practices.
7. Add a checkoutController.js file. Create your new controller in it using the best practices we learned earlier.
8. In that controller, add a hardcoded JSON object called cart which might look like this:

```
$scope.cart = [
    {
     "product":{ /* Copy a product from mainProductController.js */},
     "quantity": 4
    },
    {
     "product":{ /* Copy a different product */},
     "quantity": 2
    },
    /* And so on ... */
  ];
```

   Note: Look for a file under *assets* called cart.json.js with a hardcoded cart in it. Using this file will save you from typing in some JSON data.
9. Go ahead and fill your cart contents with as many items as you'd like.
10. Underneath the controller but inside the IIFE, add this function:

```
function getCartTotal(cart) {
  // Loop through your $scope.cart array and return a sum
  // of all quantities * prices
};
```

11. In the controller, add these methods:

```
$scope.$watch('cart', function () {
  $scope.cartTotal = getCartTotal($scope.cart);
}, true);
$scope.removeFromCart = function (product) {
  // Remove the entire line from the cart for the product passed in
};
$scope.processOrder = function () {
  // console.log each item in the cart and then clear out the
  // cart contents
};
```

12. Add a method called processOrder(). It should console.log the cart details and then clear out all the cart lines. . This will simulate our order submission.

Displaying them in the view

13. In your checkout.html view, add links to the above needed scripts and to Angular itself.
14. Find the <main> tag. Add an ng-controller directive pointing to your new checkoutController.
15. Display the cart contents in a table. (Hint: you'll use ng-repeat for each line in the cart.)
16. Look in the table footer (Hint: <tfoot>). Do you see where we're displaying the cart total? Change that to be an AngularJS binding to cartTotal.
17. Run and test. Make sure you can see your cart contents and total.
18. When the user hits the Place Order button, make sure processOrder runs. (Hint: ng-click).
19. Run and test the order submission. You should see the order details appear in the console and see the cart clear.

Altering cart items

The user may change his mind about an item. They may want to change the quantity of an item or delete it entirely.

20. Note the trash can icon to the right of each cart line. Add a click event to it that will call removeLine to remove just that line. (Hint: Angular keeps track of each line it adds, so you can just pass in *line.product* for the current product)
21. Since you already wrote removeLine(), you should be able to test it out. Click the trashcan icon and see if that line is deleted.
22. In your ng-repeat, find where you've bound quantity. Replace that expression with something like this:

```
<input type="number" ng-model="line.quantity" />
```

    where *line* is your range variable.
23. Run and test. This should work without *any* extra coding! Angular rocks!

Once you can submit your order and remove items from your cart, you can be finished.

Forms Intro Lab

Angularizing the login and register pages

1. Look in the auth folder. You'll see two pages, login.html and register.html.
2. In this folder, add an authModule.js file which defines the authModule.
3. Add <script> tags to the tops of login.html and register.html to your new authModule.
4. Add ng-app="authModule" to the <body> tag.
5. Add two new controllers called loginController.js and registerController.js. They should both be created using our best practices and should both have $scope injected.
6. Open both HTML files and find the <main> tag in each. Put ng-controller attributes on them so Angular will use our controllers. Don't forget the <script> tags to include your respective controllers.

Wiring up the login page

Now that angular is in charge of these pages, let's bind each to its model.

7. Add this to your loginController.

```
$scope.login = function (user) {
  $scope.successMessage = "You are now logged in as " + user;
};
```

8. Open login.html. Note that there's a <p class="alert alert-success"> at the top. Give that a one-way binding to successMessage. Make it show only when successMessage is set. (hint: ng-show)
9. Find the <form>. Give it a name. Remove action and method. Make it call your login function above when the user submits the form.
10. Find the inputs. Bind each to a model variable.
11. Run and test. When you click the button you should be able to see the username in the login message.

Wiring up the registration page

12. Add a method to your registrationController called $scope.register which is similar to your login method from above. It should set a successMessage.
13. The success message is at the bottom of register.html this time. Bind it to successMessage $scope variable.
14. Wire up the form submit to $scope.register(), removing action and method.
15. Connect all inputs to $scope models.
16. Feel free to put testing values in as mustaches. Run and test. Remove the testing mustaches before moving on.

Now let's tame our checkout page! Look in the *ordering* folder for all of the below.

Wiring up the shipping costs

17. Add a new controller called shipViaController as part of the orderingModule.
18. Add this to the controller:

```
$scope.shipViaOptions = [
  {id: 1, name:'Next day', price:100},
  {id: 2, name:'Two day', price:50},
  {id: 3, name:'Ground', price:0}
];
```

19. Open checkout.html and make sure that controller file is included in a <script> tag.
20. Find the <section> where the checkout page offers shipping options. Add an ng-controller directive to it pointing to your new shipViaController.
21. Populate that dropdown using ng-options. The shipVia name should be displayed in the box.
22. Don't forget to add an ng-model='shipVia' to bind the option chosen at run time.
23. Add this temporary test expression anywhere in the section:

```
{{ shipVia }}
```

24. Run and test. Once you're convinced that it is working, remove your test expression.
25. Bonus!! See if you can make the name and the price show up in the dropdown. (Hint: use string concatenation with the "+" sign in ng-options).

Wiring up the ship-to section

Now let's do the same with the ship-to section.

26. Add a new controller called shipAddressController.js.
27. Add a fake logged-in customer to it:

```
$scope.customer = {
  "companyName": "Bottom-Dollar Markets",
  "contactName": "Elizabeth Lincoln",
  "address": "23 Tsawassen Blvd.",
  "city": "Tsawassen",
  "region": "BC",
  "postalCode": "T2F 8M4",
  "country": "Canada"
};
```

28. Add an ng-controller directive to the section tag pointing to your controller.
29. Find the <form> tag in that section and remove the action and method attributes.
30. Add a name attribute to the form and make sure each <input> has a name and an ng-model so that the customer shows up in the form.
31. Run and test. Feel free to put some dummy mustaches on the page to debug your values but if you do, don't forget to remove them.

Once you have well-behaving forms, you can be finished.

Forms Validation Lab

Form validation properties

1. Open checkout.html. Scan through the <input>s for validation properties like required, pattern, type=number, type=email and so forth. Get an idea of what each should do.
2. Note that some of the fields have maxlength="XX". Run and test. Try to put more than the max number of characters in that field. Can you? _____
3. Change them all to ng-maxlength. Try again. Can you now? _____
4. Find every <input> in this page and add this after each one:

```
<span class="glyphicon form-control-feedback">*</span>
```

5. Add an ng-show directive to show each of these spans only if the field is invalid. (Hint: formName.fieldName.$invalid).
6. Run and test. Every one should show the moment an error occurs.
7. It does seem a little rude to show fields in error before the user gets a chance to change them. change your ng-show expression to fix that. (Hint: formName.fieldName.$invalid && formName.fieldName.$touched)
8. Bonus!! If you have time, do the same for the login.html and register.html files.

More specific help

We're telling the user when a validation problem has happened but we're not telling them what the problem is nor how to solve it. We've got to do something about that.

9. Notice that the shipName input is required and has a max length and has a pattern. Add these below it:

```
<div class="alert alert-danger">Your name is required.</div>
<div class="alert alert-danger">That has a weird character.</div>
<div class="alert alert-danger">Too many characters. Shorten it.</div>
```

10. Now, they will always show but we only want them to show when there is a particular problem. So add a ng-show directive to the first that will only be true when the input has been $touched and formName.fieldName.$error.required is true.
11. Run and test.
12. Do the same for the second field but with a pattern error.
13. And again for the third with a maxlength error.
14. Copy and paste these for each field in your form. Of course, change the message to fit the field and eliminate messages that don't fit the situation.
15. Bonus!! Do the same for the login and register pages.

Setting styles

Now we're going to add some CSS to draw focus to invalid field. You can do this by just setting. But if you are using Bootstrap, this is pretty slick.

16. First, open site.css and add this:

```
input.ng-invalid.ng-touched { background-color: #FF9494; color: white;}
```

17. Run and test. You will see all fields turn red when in error.
18. Bootstrap gives us some even cooler options. So remove that style from site.css.
19. Then, notice that each field has generally this shape:

```
<div class="form-group has-feedback">
  <label for="shipName" class="control-label">Name to ship to:</label>
  <input ng-model="shipName" name="shipName" class="form-control" />
</div>
```

20. add something like this to the form-group :

```
ng-class="{'has-success': shipTo.shipName.$valid, 'has-error':
shipTo.shipName.$invalid}"
```

21. Then run and test. Bad data will cause it to turn red.
22. Bonus!! Add a $touched requirement to it so it starts out neutral and only turns green or red when the user blurs from the field.
23. Extra bonus!! Add this to all the fields on checkout.html.
24. Super bonus!!! Add this to all the fields in login.html and checkout.html

Preventing submission

Take a look at the register.html page. As of now, your user could submit the form even when there are validation problems. We can fix that easily enough.

25. Find the submit button on your form. Add something like this to it:

```
ng-disabled="formName.$invalid"
```

26. Run and test.

Once you have a well-behaving and validating form, you can be finished. Take a break. You've earned it!

Filters Lab

Formatting data

You are displaying some prices on your site. If you haven't already formatted them, they look fairly ugly.

1. Open checkout.html and find all prices. Where you're displaying any price, go ahead and run it through the currency filter.
2. Since we have to price it in US dollars, add a prefix of "US$" as a parameter. If we don't, the currency prefix will be localized for our users. Yikes!
3. Run and test. Once working, you can move on.

Creating a catalog of products

Let's create a way for our wonderful users to browse through all of our products.

4. If you look in the app/product directory you'll see a file called productList.html. Open that file in your editor.
5. Convert the hardcoded category dropdown to an Angular select list. You can do that kind of like this:

```
<select ng-model='category' name="category" id="category"
        ng-options="category.categoryName for category in categories">
  <option value="">All categories</option>
</select>
```

6. Now clearly we'll need to populate those categories in the controller. So create a controller called productListController.js. Make it a controller in the *productModule* module using all the best practices we've learned so far.
7. Add some categories like this:

```
$scope.categories = [
{"categoryID" : 2, "categoryName" : "Condiments",
  "description" : "Sauces, relishes, spreads, and seasonings"},
{"categoryID" : 4, "categoryName" : "Dairy Products",
  "description" : "Cheeses"},
{"categoryID" : 1, "categoryName" : "Beverages",
  "description" : "Soft drinks, coffees, teas, beers, and ales"},
... and so on
]
```

   Note: look for a file under assets called categories.json.js with some categories. If you copy from it you may save yourself some typing.
8. Make sure everything is wired up properly and test out your view. Can you see your list of categories? Good!
9. In the controller, create a bunch (10 or more) products in an array. Feel free to copy and paste from mainProductController,js.
10. Open your view again and display your $scope's products using ng-repeat. (Note that there are already hardcoded products. Convert that.)
11. Run and test, making sure you can see all 10 items.

Limiting by category

Here comes the payoff. Our goal is that when the user selects a category, he/she only sees the products in that category. To do that we'll employ a filter.

12. Add a "filter" filter to the ng-repeat expression. It should limit the results to only those products whose categoryID is the same as the one the user chooses. It should look kind of like this:

```
| filter: {categoryID: category.categoryID}:true
```

13. The *categoryID* says to match each product's categoryID. the *category* is the one chosen from the dropdown. The *true* means it should be an exact match.
14. Run and test again. Adjust your filter until you get it working.

Adding search ability

We also want our users to be able to search for products on our site. Let's give them a search page. We'll start with the controller.

15. Open the app/product directory and create a new controller file called productSearchController.js.
16. As before create an array of products. Go ahead and copy/paste.
17. You should see a productSearch.html page in that same app/product directory. Go ahead and open that in your editor.
18. Add links to your controller file and whatever else you'll need.
19. Find the input box type="search". Its ng-model should be "searchString". No need for a submit button.
20. Below that, display your $scope's products using ng-repeat.
21. Run and test, making sure you can see all 10 items.

Making the search work

We have a page with 10 products, but the search box does nothing. We can make it work with just a little Angular magic.

22. Add a *filter* filter to the ng-repeat expression. You want to limit the results to only those items that match the search string entered in your search box.
23. Run and test again.

Once you can type into the search box and see your limited list in real time, you can be finished.

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 " product overview "
        - 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.