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).