Ajax Lab

Now the real fun begins! We've been working with fake data up to this point but in this lab we will begin reading real data from a RESTful service running via node, express, and mongoDB.

Setup

1. First we have to make sure mongod is running. In a window somewhere type in

```
mongod
```

   It should tell you that it is listening on port 27017 or something like that. Let this run at all times when running your server.
2. To make sure it is running, open a new command window and type in

```
mongo wms
```

3. Then type in

```
db.products.find()
```

4. You should see a list of a bunch of products. See them? Cool. Let's move on.
5. Next we need to install something that will allow us to run things in parallel:

```
npm install --save-dev npm-run-all
```

6. Next we must make sure Node/Express are running. Look in /setup/assets/codeSnippets for a file called proxy.conf.json. Copy that into your warehouse directory (The root of your Angular app).
7. Now edit package.json in that folder. Find where it says

```
"start": "ng serve",
```

   and replace that with the contents of /setup/assets/codeSnippets/runParallel.json
   Here you're telling it to route all requests through our web server that understands Ajax RESTful API requests.
8. Try it out. Run "npm start". Once you see some success messages, browse to

```
http://localhost:4200
```

9. You should see your Angular app. Then browse to

```
http://localhost:4200/api/products
```

10. You should see all those same products you saw in the database. Once you see them, you can move on to writing some Ajax requests.

Getting orders ready to be shipped

In our DashboardComponent, we have a hardcoded list of orders ready to be shipped and we have a count of those orders. Let's populate that with real data.

11. Set the DashboardComponent as the startup component. (HInt: app.module.ts, set bootstrap to DashboardComponent).
12. Open dashboard-component.ts. Find where you're listing some hardcoded orders in ngOnInit(). Remove those orders.
13. Create a constructor. Make the signature read like so:

```
constructor(private _http:Http)
```

14. Of course this won't compile because Http isn't defined. import Http from @angular/http. And while you're there import toPromise from rxjs also:

```
import 'rxjs/add/operator/toPromise';
```

15. Create a method called getOrdersReadyToShip(). Have it console.log("hello world") for now.
16. Call getOrdersReadyToShip() from ngOnInit() .
17. Run and test. Make sure you can see your console message before you move on.
18. In getOrdersReadyToShip(), call the get method on this._http. Pass in the URL /api/orders/readyToShip.
19. Convert it to a promise by calling .toPromise().

20. Register a success function that will simply console.log the response.
21. Run and test again. You should see all the current orders in JSON format.
22. Now set this.orders to that response in your callback.
23. Run and test. When the page is loaded, you should see actual orders whose status is zero from the database. Feel free to look in the database to verify that they match up.
24. Now do the same for the OrdersToShip component. It should be the same as for the Dashboard.

Bonus! Make the badge and the messages work

25. In the DashboardComponent there is a badge saying how many orders need to be picked. Make that reflect the real number of orders.
26. There are also two messages on the page. Only one of the two should ever be seen. Make one show when there are orders to be picked and the other show when there are none.