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 and express.

Setup

1. Start a terminal window and cd to your warehouse project's folder.
2. We need to install something that will allow us to run things in parallel:

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

3. 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).
4. 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.
5. Try it out. Run "npm start". Once you see some success messages, browse to

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

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

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

7. 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.

8. Open dashboard.component.ts. Find where you're listing some hardcoded orders in ngOnInit(). Remove those orders.
9. Create a constructor. Make the signature read like so:

```
constructor(private _http: HttpClient)
```

10. Of course this won't compile because HttpClient isn't defined. import HttpClient from @angular/common/http.
11. Run and test. Oh no! Once again, it won't work. This time it is because the HttpClientModule is out of scope. Fix that by editing app.module.ts and adding HttpClientModule to the imports array. (Hint: Don't forget to JavaScript import it from @angular/common/http)
12. Create a method called getOrdersReadyToShip(). Have it console.log("hello world") for now.
13. Call getOrdersReadyToShip() from ngOnInit() .
14. Run and test. Make sure you can see your console message before you move on.
15. In getOrdersReadyToShip(), call the get method on this._http. Pass in the URL /api/orders/readyToShip.
16. Register a success function that will simply console.log the response.
17. Run and test again. Look in the browser console. You should see all the current orders in JSON format.
18. Now set this.orders to that response in your callback.
19. 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.
20. 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

21. In the DashboardComponent there is a badge saying how many orders need to be picked. Make that reflect the real number of orders.
22. 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.