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 our 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 /starters/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 this:

```
"startApi": "npm run start --prefix  ../server",
"startSite": "ng serve --proxy-config proxy.conf.json",
"start": "npm-run-all --parallel startApi startSite",
```

Here we're telling it to route all requests through our RESTful API server that understands Ajax 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 currently have a hardcoded list of orders ready to be shipped. Let's populate that with real data.

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

```
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. You shouldn't see any difference but it should also not error.
    - Hint: If it errors in the browser remember that the HttpClientModule needs to be imported. Fix that by editing app.module.ts and adding HttpClientModule to the imports array.

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 instead of console.log()ging the orders, set this.orders to that response in your callback.
19. Run and test. When the page is loaded, you'll see actual orders whose status is zero from the database. Feel free to look in the database to verify that they match up.

Don't forget to unsubscribe!

As you can see it's working great, but there's a risk of a memory leak. If we want to program cleanly we'll unsubscribe.

20. First, create a class variable to store the subscription:

```
ordersSub?: Subscription;
```

21. Next, when subscribing, save the subscription.
22. And lastly, create an onDestroy Lifecycle method where you'll unsubscribe.

```
ngOnDestroy() {
  this.ordersSub?.unsubscribe();
}
```

Orders To Ship

Remember, OrdersToShip works with the same data as Dashboard. Let's re-do all this for OrdersToShip. Try to do as much from memory as possible. Try not to look at the notes to accomplish this. It'll be a great learning experience.

23. All the subscribing and unsubscribing you did to Dashboard, do the same for the OrdersToShip component.

Using the async pipe

Let's try a different approach to the whole Ajax thing. You may like it better.

If you run your application and look at either the dashboard or the orders to ship component then click on an order, you'll be sent to http://localhost:4200/ship/<id>. Then look at the order ID in the url and on the page. Do they match? _____ They should. Now look at the order date, the ship_via, the shipping label at the bottom and the list of products. All of those are hardcoded so they will always stay the same. Let's get some real data from our Ajax API.

24. Open ship-order.component.ts. Prepare it to make Ajax calls by importing and injecting HttpClient. (Hint: This is what you did in the last sections and you should be familiar with it by now).

In ngOnInit, you're properly getting the order ID from the URL. But then you create fake hardcoded data that looks like an order.

25. Edit ngOnInit. Delete the code where you're creating that fake order. Make your GET call but don't process the observable. Just store the observable like this:

```
const orderId = this._route.snapshot.params['orderId'];
```

```
this.order$ = this._http.get<Order>(`/api/orders/${orderId}`);
```

26. This won't compile because this.order$ hasn't been declared. Add it to the class and delete the this.order declaration. Like this:

```
//order: Order = new Order();   // <-- Remove this declaration
order$?: Observable<Order>;     // <-- Add this one
```

We're removing this.order because we need to make "order" a local variable in the next steps.

27. Edit ship-order.component.html. Wrap the entire thing in a :

```
<div *ngIf="order$ | async ; let order">
```

This says "observe order$. As soon as it is settled, put its result in a local variable called 'order' and show the "

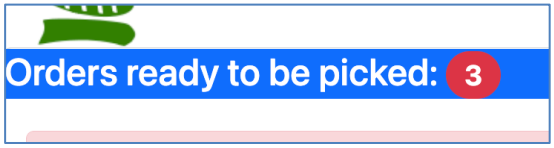
You should see the right order date, ship via, shipping address, the right products in the list, and the right shipping address in the shipping label at the bottom of the component.

What do you think? It's certainly a lot less coding and more abstract but also more arcane.

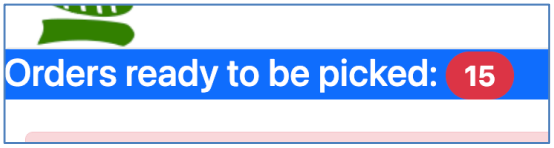
Bonus! Make the badge work

If you get finished early, this will be fun and easy. In the DashboardComponent there is a badge saying how many orders need to be picked.

Nope! Hardcoded number



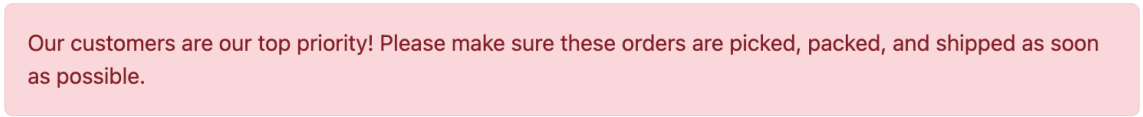
Yep! Reflects the correct number of orders

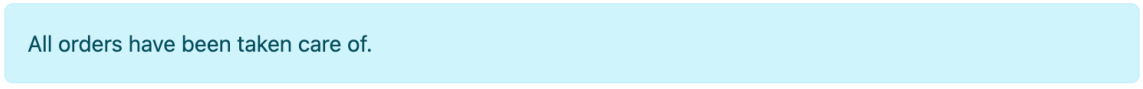


28. Make that reflect the real number of orders.

Extra bonus! Show one message

29. Also in Dashboard, there are two messages on the page. Only one of the two should ever be seen:





30. Make the first one show when there are orders to be picked and the other show when there are none. (Hint: *ngIf)