Angular Labs

Overview

Things the WMS can do

Note: until Ajax chapter, all server GETs are simulated with a hardcoded class variable.

1. AppComponent - The landing page - A dashboard
    - Links to all three of the other components. Will use routing eventually.
    - Dashboard
        - How many orders need to be picked.
        - Products we are low on??
2. Receive a product
    - Scan a tracking number to show the rest
    - Enter a product ID and quantity
        - product ID can be chosen from a lookup - As the user types, the list get more specific. If he/she clicks/taps on a product, it puts the product ID in the box and fills in the details onblur.
    - Ajax request to the server should bring back the details for this product.
    - Accept button -> Add it to inventory and assign it a location. Report back the location.
    - Location comes from server. It's random for now but server can make decisions later.
    - Note: no invoice/receiver
3. Products ready to ship
    - Server provides a list of orders ready to ship.
    - When clicked, each takes you to ship Product
4. Ship product
    - Server returns order (which includes product IDs and quantities and locations)
    - For each line, the user hits 'ready to pick' and system returns a location where it exists.
    - User marks that it was PPSd
    - User enters UPS shipping number
    - System reduces inventory by that amount in that location
    - User can hit a 'problem' button (We're too low on stock or something)
5. Take inventory
    - System provides a list of everything in stock with locations
    - Sort by location or by product

Things we can add later

6. Adjust inventory
    - From inventory page, click on a link taking you to this route. Product ID and Location are transferred.
    - Quantity field is populated with current quantity. User can change.
    - Commit writes it to inventory.
7. Receive a receipt (?)
    - User enters a receiver number (our PO or whatever)
    - Ajax request - server returns an array of product/quantities on that order
    - System shows products with details and asks user to type in quantity in shipment.
    - If expected != actual, system notes that and tells the user.
    - Calls Receive a Product from above.
8. adsf

Introduction Lab

Congratulations! You and your partners have just been hired by Northwind Traders, a restaurant supplier to build the warehouse management system (WMS) for their burgeoning business! They've been

1. keeping track of inventory,
2. shipping product,
3. and receiving new product into their small warehouse

all by hand! Their growth has taken them to the point where they need to automate.

Let's get started!

Make sure node and npm are installed

1. Open a terminal window and run this command:

```
node --version
```

2. Did you see a version number? If so, you have npm installed and you can skip to "Make sure Angular CLI is Installed". If not, go to the next step.
3. Point your browser at http://nodejs.org
4. Click the download button on the main page. Choose the one recommended for most users; it will be the stable release.
5. Follow the instructions to install it on your machine. This will install both npm and node.
6. Rerun your node --version command to make sure it is installed alright. If you get command not found, you should probably reboot so make sure your PATH has been reread.

Make sure Angular CLI is installed

7. Open a terminal window and run this command:

```
ng version
```

8. Did you see a version number? If so, you have the Angular CLI installed and you can skip to "Download the starter files". If not, go to the next step.
9. Open a command window as an administrator and run this command

```
npm install --global @angular/cli
```

10. Rerun your ng --version command to make sure it has installed properly.
11. If it does, skip to " Download the starter files ". If not, you'll need to add the installation directory to your PATH. That installation directory on Windows is *%APPDATA%/npm*. On MacOS, it is */usr/local/bin/ng*.

Download the starter files

12. Point your browser to the site your instructor gives you to download and install the starter files.
13. Take a look around the directory structure. You should see these directories:

instructions	Will eventually hold each lab's instructions
starters	Files and commands to load the database. Some code snippets.
server	The server-side code which is pre-written for you.

14. This is the root of your project. Please make a note of it here:

    _____

Starting the RESTful API server

15. Open a terminal window and cd to the server directory. Run these commands.

```
npm install
npm run start
```

It will tell you it is listening on a particular port, usually 3000.

16. Try a few of these addresses:

```
localhost:3000/api/products
localhost:3000/api/locations
localhost:3000/api/orders/10250
```

If you can see data, you are up and running! You can be finished with this lab.

Bonus!!

17. Make the order dates more current and realistic. Run this command:

```
npm run updateDates
```

Angular CLI Lab

Let's create the basis for our Angular-ized WMS. We'll start off by creating the bare-bones Angular app and then add some components to it.

Creating a starter site

1. Open a command window and cd to the root of your project.

```
ng new warehouse
```

   If it asks you about settings, answer like this:
   - Would you like to add Angular routing? - No
   - Which stylesheet format would you like to use? - CSS

   Note: this command may take a while to run mostly because of the installation of the supporting libraries. And it may complain about not knowing who you are on github, Both of these are normal.
2. Notice that there is now a new directory called 'warehouse'. cd to it. Look around at the files created.

Using the development server

Let's use the built-in server that comes with the Angular CLI.

3. Do this:

```
ng serve
```

4. Look at the message. Make sure it says that it is running a server at a particular port.
5. Point your browser to localhost at that port. Make sure your page comes up. You should see Angular's default page in the browser with "warehouse" near the top.
6. Keeping your browser open, edit the app.component.ts file in your favorite IDE.
7. Change the *title* variable to "Northwind Traders Warehouse Management". Hit save and watch the browser window. You should see your application reload without refreshing. This is the file watcher in action.
8. Go ahead and select everything in app.component.html and replace it with this:

```
<h1>
  {{title}}
</h1>
```

9. Once again, when you save it, you should see your new title appear in the browser but you've gotten rid of Angular's boilerplate content.

Working with angular.json

The Angular CLI makes some decisions for us that we may not agree with. We can change some of those by changing the config file called angular.json. Let's say we want to put our images in a different location than the default demands.

10. Delete favicon.ico in the src folder.
11. Look in the starters folder. Do you see the images folder? Move it and all of the contents into the warehouse/src/assets folder. You'll know you have it right when you have a directory called warehouse/src/assets/images and in there you should see favicon.ico along with some other images.
12. Edit angular.json. Find the line where the favicon is included in the build. Remove that line and make sure the JSON is valid.
13. Quit ng serve and run it again. Adjust angular.json until it validates.

14. Open index.html in your IDE and change the favicon line in the head to the proper folder (assets/images/favicon.ico). Keep adjusting until you see your new favicon. (Hint: you may need to hard reload your browser or open in incognito mode to avoid the browser cache).
15. Back in angular.json, look for the *prefix* key. It currently says "app". Change it to something related to this project. Suggestion: You could use "nw" which is short for "Northwind Traders", the name of our company. That's what we'll be using in these labs.

Adding sitewide styles

16. We've decided to use Bootstrap for styling. Install Bootstrap like so:

```
npm install bootstrap
```

17. This will have created a new folder called *bootstrap* under node_modules. See if you can find it. You'll need that in the next few steps.
18. Open the angular.json file. Find the *projects.warehouse.architect.build.options.styles* key. Notice that it is an array. Add Bootstrap's css file to that array as a string. (Hint: It might be located in something like "bootstrap/dist/css/bootstrap.min.css")
19. Find the *projects.warehouse.architect.build.options.scripts* key. Did you find it? In there add Bootstrap's JavaScript. (Hint: maybe "bootstrap/dist/js/bootstrap.min.js"?).
20. Stop and restart the development server so your changed angular.json file will be read.

If you see the font change to a sans-serif font, you've got it working properly.

Scaffolding components

21. Using the Angular CLI, create these four components (HInt: don't forget about the --dry-run option if you want to see what is about to happen).
    - Dashboard in a 'dashboard' directory
    - Inventory in a 'inventory' directory
    - ReceiveProduct in a 'receiving' directory
    - ShipOrder in a 'shipping' directory
    - OrdersToShip in that same 'shipping' directory

Hint: to make the last three, you'll need to do something like

```
ng generate component --flat shipping/shipOrder
```

22. Look in the three folders and examine the files created by this. You should see four new files for each component created.
23. Make the receiveProductComponent appear on the page by editing app.component.html and adding this to it

```
<nw-receive-product></nw-receive-product>
```

24. Run and test. When you see your new component appear on the main page, you know you did it right.

Giving the components a little shape

We've created some components but you have to admit, they're all pretty boring, right? Let's give them some content. But we know you'd rather focus on Angular instead of HTML so rather than you taking a lot of time to write these components by hand, we'll give you starters.

25. Look for a folder called starters/html. And in that folder you'll find some html files you can use for your components. They're full of hardcoded, fake data.
26. Edit receive-product.component.html. Replace its content with the receive-product HTML file in the starters directory.
27. Do the same for orders-to-ship.component.htm, ship-order.component.html and dashboard.component.html.
28. Give each of those a look by changing app.component.html so each gets a turn at being the startup component.

Once you've seen the four components, you can be finished.

Big Picture Lab

In this chapter we learned about the arcane booting process of Angular. Let's now look at the code and even make a few adjustments.

1. Open the angular.json config file in your IDE. Find the projects.warehouse.architect.build.options.index property. What is the index page file? _____.
2. Edit that file. Do you see the <app-root> tag? Add some text inside of that that says "Loading your warehouse management application. If this message doesn't go away, refresh the page. If it refuses to load, contact tech support."
3. Back in angular.json. Find the projects.warehouse.architect.build.options.main property. What is the main TypeScript file? _____
4. Find that file and open it in your IDE.
5. What module does it bootstrap? _____.
6. Looking above it, where is that module imported from? './app/_____'.
7. Now open that ts file and examine it. Note that the class in it is marked with a @NgModule decorator/attribute.
8. What component is being bootstrapped? _____
9. And like before, look above. Where is that file imported from? _____
10. Awesome! Keep following the breadcrumbs and open that file now. Do you see how the title was set? _____
11. What is the selector? _____
12. What is the template file? _____
13. Open that template file and look around.

Now that you know how the page is first drawn, you can probably guess how to make some changes to it. Let's do that now.

Change the boot component

The AppComponent traditionally serves as the landing page. On ours we're going to put a menu to access the other functions of our WMS and a dashboard for important information. Now, rather than having you type all that in, we've provided some more starter HTML that you can use.

14. Take a look at /starters/html/app.component.html. Copy all the contents of /starters /html/app.component.html into the app.component.html you created.
15. Replace the placeholder message inside <main> with <nw-dashboard></nw-dashboard>
16. Run and test. It should look pretty good but the links don't take you anywhere and the data is all fake and hardcoded. We'll fix these things later.

Hey, at least you now know how to change the main component's template.

Bonus!! Examining all the parts of an Angular app

17. If you're finished early and want an extra challenge, look back at your course lecture material. Find the Big Picture diagram. Now look through the basic site that has been scaffolded for you and see if you can identify all the pieces from the diagram. Check these off when you find them:
    __ A file with the @NgModule annotation
    __ A list of the components in that file
    __ The startup component in that same file (Hint: Bootstrap)

__ A file with the @Component annotation
__ The template. Where is the template that component points to? _____
__ The class in that file
__ A property in that class
__ Where it is displayed in the template (Hint: Interpolation).

When you've located all these things, you should have a good idea of how they all fit together. But don't worry if you don't, we'll cover all that in later chapters.

TypeScript Lab

Let's exercise some TypeScript! We're going to ask you to create some TS types that'll be used in future labs. We're hoping you get a feel for the new demands that TS puts on you -- demands of safety, demands of precision. And while many people welcome those demands, others will resent them. So we warn you; this lab may be painful if you're in that latter category.

We're hoping that this experience will make you open to a sane balance between the speed & freedom of dynamic JavaScript vs the safety & protections of typed TypeScript. Some hints:

1. Read the entire lab before you begin. There are comments and suggestions offered after-the-fact that can help you to solve problems.
2. When creating types, you can create them as classes, interfaces, or types. Each have their own advantages. You have to decide which is best.
3. Class properties and method variables can be strongly typed. But the types can be omitted when it makes sense to. (This is hotly debated topic).
4. If you're finding TypeScript is too picky, remember that the tsconfig file has a "strict" flag that can be set to false. This is your way of saying "TypeScript, please chill for a minute".

With that said ... Get after it!

Making our types

We're going to be working with products, orders, and customers so it might be safer to create types to define the shape of the objects we'll be working with. We'll start by putting them in a shared folder.

1. Create a new folder under *src/app* called the *shared* folder.
2. In it, create five new files called customer.ts, location.ts, order.ts, orderLine.ts, and product.ts.
3. These will be our type files. In each, create one class, interface or type. You get to decide which one you want for each. Give them the following properties:

Customer

id	number
givenName	string
familyName	string
companyName	string
address	string
city	string
region	string
postalCode	string
country	string
phone	string
email	string
imageUrl	string
password	string

Order

id	number
customerID	number
status	number
orderDate	Date
shipVia	number
shipping	number
tax	number
shipName	string
shipAddress	string
shipCity	string
shipRegion	string
shipPostalCode	string
shipCountry	string
lines	Array<OrderLine>
customer	Customer

Product

id	number
name	string
description	string
price	number
imageUrl	string
featured	boolean

OrderLine

quantity	number
productID	number
locationID	string
price	number
picked	boolean
product	Product
location	Location

Location

id	string
description	string
productID	number
quantity	number

Note that Order uses OrderLine and Customer. Don't forget that you'll need to import these classes into Order.ts for this to work. This is true for Location and OrderLine as well.

Using the Order type in a component

The ShipOrderComponent's eventual purpose is to get an order ready to ship to the customer. So obviously it needs an order!

4. Edit shipping/ship-order.component.ts. Give the class a public property called *order* that is of type Order. (Yes, the one you just created). Don't forget you'll need to import it from Order.ts at the top.*
5. You should get an error at this point because the order property of the ShipOrderComponent class might be nullish and TS doesn't like properties accidentally becoming nullish. Discuss with your partner some ways that you might solve that.

One way to solve the problem is to make Order a **type** and make all its properties optional with the "?". But if you do that, you'll have to handle the possibly null properties throughout the html template. That's a viable alternative, but in this situation, there's a cleaner way with a lot less coding.

6. Make Order a **class** instead of a type. Then, declare all its properties with the exclamation point like so:

```
export class Order {
  id!: number;          // <-- Note the exclamation point
  customerID!: string;  // <-- Note the exclamation point
  etc. etc.
```

This says that the properties will exist, but it's okay for them to be null.

7. Now, back in ship-order.component.ts, instantiate an Order as soon as you declare it. You can do it like this:

```
order: Order = new Order();
```

8. We've provided a starter to create a fake order so you don't have to type every single property. Go look in /starters/codeSnippets/anOrderReadyToShip.js. Copy its contents into the ngOnInit() method of ship-order.component.ts.

Unless you wrote your types in a very particular way, you're encountering many compile errors. Fix those by making properties optional with "?" or "!". Or you could just make it really easy and use the "any" type.

Let's take a breather for a second. If you were using plain JavaScript or set "strict": false in tsconfig.json or if you took the easy way out and made everything an "any" type, you wouldn't have these problems. It's a lot simpler to code that way, but it's also more dangerous at runtime. You and your team will have to decide standards on how to handle these situations - quicker coding or safer at runtime?

---

* We're only going to remind you to add imports statements occasionally from now on so don't forget going forward.

Hey, don't be discouraged. You're not alone. TypeScript makes learning Angular more complicated. Once you get through the learning curve, you'll be writing safer and better code!

Let's do one that is a little easier.

9. Add this to dashboard.component.ts and orders-to-ship.component.ts

```
orders: Order[] = new Array<Order>();
```

This means that we need a list of Orders. Only an Order can be added to this list. We're also initializing it so it's not nullish.

10. And now let's do last one. If you look in the starters/codeSnippets folder, you'll see a file called SomeOrders.js. Open the file and copy its contents to the ngOnInit() method of the DashBoardComponent. Be ready to adjust the data or your business classes.
11. This will populate a property called "orders". Do a console.log() in ngOnInit() to make sure it is reading alright. We'll use this when we talk about directives soon. Stand by for that!

Bonus!! Exploring tree-shaking

If you're finished early and have extra time, let's explore how webpack and TypeScript interact to make your payloads smaller.

12. Open a command window and compile your site by typing in
`ng build`
or
`ng build --configuration=development`
13. Find the dist directory and look in there. You should see a main.*.js file. This is the file that will be served once we go live.
14. Go ahead and edit main.*.js. You and your partner look around in there. Do you see your ShipOrder component in there? _____ (You shouldn't unless you included it in app.component.html). How about your TypeScript classes? Are some there but not others? _____ Discuss with some of the students around you why this makes sense.

What's going on is this: TypeScript files are only included if they're used. If not, they're excluded from the bundle, keeping the size smaller.

Components Intro Lab

You've created some components, ReceiveProduct, OrdersToShip, ShipOrder, Inventory, and Dashboard. Let's practice with some interpolation and styling.

Running a component

You have a ship-order component that needs a little attention. Let's work on that one for a while.

1. Edit app.component.html. Change the component loaded to <nw-ship-order>.
2. Run and test your application.
3. You should now be seeing your ShipOrder component. Make adjustments until you do. (Hint: some new errors may now be showing up in the console. Open it up to see the details).

Interpolation

You can see your ShipOrder component but all the data is hardcoded. In the last lab you went to the trouble to set up a fake order. Let's see some of it on the page.

4. Edit ship-order.component.html
5. Find the hardcoded order ID and change it to

```
{{ order.id }}
```

6. Run and test. You should now be seeing the order ID from the component class.
7. Do the same for order.status, order.orderDate, order.shipVia, and the things in the shipping label at the bottom like order.shipName, order.shipAddress, order.shipCity, order.shipRegion, order.shipPostalCode, & order.shipCountry. All of these are in the template.

We know that we're still not showing the order lines yet. That'll happen in a later lab. Please don't show those just yet.

Styling the checkboxes

When the user picks an order line from its location in the warehouse, they will check the box to indicate that the item was picked properly. But it looks as though the checkboxes are not tap-friendly on a phone or tablet. Let's double the size of those checkboxes.

8. Open the ShipOrder component and locate the @Component decorator. Inside of that add a *style* property which is an array of strings. Add a style for a .bigCheckbox and set its transform property to scale(2).
9. Open the html template. Add the bigCheckbox class to the checkboxes.
10. Run and test. The checkboxes should be huge now.

Styling the photos

11. View the ReceiveProduct component in the browser (Hint: You'll change app.component.html again). Notice that the photos are a little large. We should resize them using a style.
12. Open receive-product.component.css. Add a style to it to set the height to 50px for the selector "#orderContents img"

Styling dashboard's orders to ship

In the dashboard, we have a list of orders but the list is kind of hard to read. It currently looks like this.

Order	Date	Products	Total
1	2017-06-09T07:26:21.032Z 123 Main Street, Anytown, ST, 01234	3	
2	2017-06-09T21:54:41.252Z 345 Elm Lane, Yourtown, ST, 98765 USA	5	
3	2017-06-10T10:54:24.384Z Carrera 52 con Ave. Bolívar #65-98 Llano Largo, Barquisimeto, Lara, Barquisimeto, Venezuela	2	

13. See if you can make it look kind of like this:

Order	**Date**	**Products**	**Total**
1	2017-06-09T07:26:21.032Z 123 Main Street, Anytown, ST, 01234	3	
2	2017-06-09T21:54:41.252Z 345 Elm Lane, Yourtown, ST, 98765 USA	5	
3	2017-06-10T10:54:24.384Z Carrera 52 con Ave. Bolívar #65-98 Llano Largo, Barquisimeto, Lara, Barquisimeto, Venezuela	2	

Hints:
- Make the header row bold and slightly larger than the data in the table
- Add a top border to each order
- Put a little padding in there

14. Eventually we will make each order clickable so that the user can click/tap on one and see its details. Add a **style** (not JavaScript) so that when the user hovers over a row, the cursor changes to pointer.

Bonus! Making a shipping label

15. Make ship-order component your startup component.

Notice that there's a bunch of information at the bottom. We put it all there as shipping label so that the shippers can just print this to a sticker and put it on the box. But it needs to be formatted.

16. Notice that in starters/html there's a file called ship-order.component.css. Copy that into your src/app/shipping folder.
17. Make sure that your ship-order.component.ts is making use of that file (hint: "styleUrls").
18. You'll know you've got it right when your shipping label looks kind of like this (yes, on its side and everything):

NORTHWEST TRADERS
656329919223
NORTHWEST TRADERS, LLC.
6717 LAKE SHORE DRIVE
ANYTOWN, AS 02334-2234

12 LBS

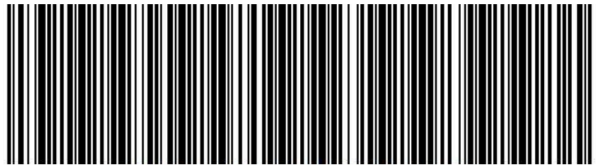
1 OF 1

SHIP TO:
SAVE-A-LOT MARKETS
656329919223
SAVE-A-LOT MARKETS
187 SUFFOLK LN.
BOISE, ID 83720
USA

ID 83720

2-DAY PRIORITY

TRACKING #: 9400 1057 3346 4567 2475 01



BILLING: P/P
ATTN DRIVER: SHIPPER RELEASE

Built-in Directives Lab

Alright, this is where the real fun with Angular begins! In this lab, we're going to iterate through arrays of orders, conditionally display messages and buttons, and conditionally set styles. Let's start with Ship Order Component.

Looping with a *ngFor

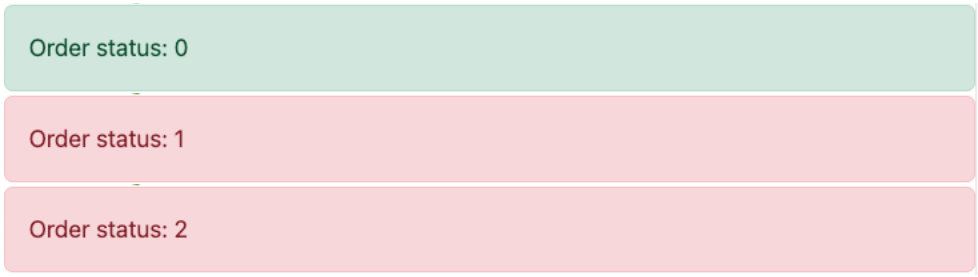
1. Edit your ship-order.component.html. Find where you currently have a hardcoded <tbody>
2. Delete all but one of the <tr>s in that <tbody>. On that remaining <tr>, add a *ngFor directive to loop through the order.lines collection that you created in a previous lab.
3. If you run and test right now, you'll see the same hardcoded line repeated. Give that a try.
4. Now go back in and change the hardcoded values to interpolated values. (Hint: use mustaches). You should see a different productid and picture on each line.

Setting conditional classes with ngClass

There are CSS classes that have been created for you in the Bootstrap CSS library. alert-success is for success messages. alert-danger is for error messages. Let's use them.

5. Take a look at the near the top of ship-order component that shows the order status.

That should have a class of alert-success if the order is ready to pick (order.status is 0) and a class of alert-danger if not. Like this.



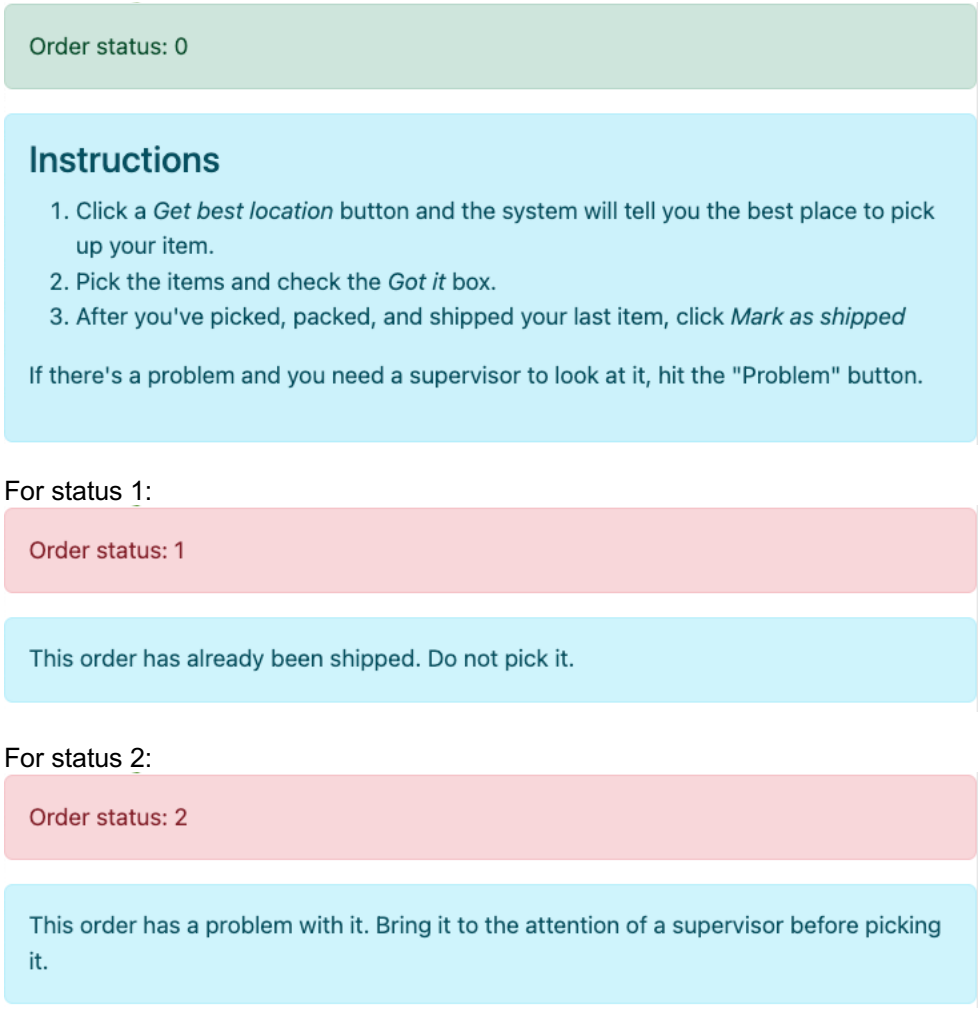
6. Change the template to change the CSS class based on the order status value. (Hint: [ngClass] takes and object in double-quotes. That object's keys are a class name like 'alert', 'alert-success', and 'alert-danger'.)

Using *ngSwitchCase

7. In ship-order.component.html find the instructions. They're in a <section>.

The existing message makes sense if the order status is "0" which means ready to pick. But not if it is "1" which means it has already been shipped or "2" which means that there is a problem with the order. We want the message to change with the status.

For status 0:



For status 1:

For status 2:

8. Each of the three messages should be wrapped in a . Add all three in individual s inside the <section>.
9. Put an [ngSwitch]="order.status" on the section itself.
10. For the first , put an *ngSwitchCase structural directive to say only show it if the order.status is 0.
11. For the second , put an *ngSwitchCase to say only show it if the order.status is 1.
12. Do the same for the third for when order.status is 2.
13. Run and test, changing the order.status in your TS class. When you're seeing the message change, let's work with those checkboxes.

Use a *ngIf with an else

Eventually our pick/pack/ship process will work like this: The picker sees the order and lines but the location isn't populated. Once they're ready to pick a line/product, they click a button to find the best location for that product and the system populates the locationID in real time.

So if the location exists on the line, we want the locationID to be seen but if not we show a button. Let's do that next.

14. Still in ship-order.component.html, find where you are currently showing a button that says "Get best location".

15. Move that button to a template:

```
<ng-template #getLocation>
  <button class="btn">Get best location</button>
</ng-template>
```

16. Inside the <td>, add a tag with the {{ line.locationID }} in it. Add a *ngIf directive to your tag to say when line.locationID is truthy, show it.
17. Run and test with one or more locationIDs nonexistent. You should see them when they're there and when not, that table cell should be empty.
18. Now add an else clause to your *ngIf. Show the button if line.locationID is falsy.
19. Run and test again. You should see a locationID for the lines that have it and a button for those that don't.

Alright, let's turn our attention to a different component.

Getting a list of orders to display

In the TypeScript lab we made a bunch of orders in the DashBoardComponent. Let's display them on the page.

20. Make DashboardComponent your starting component.
21. Open dashboard.component.html and find the s with the class of order-row. Notice that each of them holds one order. Delete all but one of those and add a *ngFor directive to the one remaining.
22. Replace each value in the s with mustaches/interpolation of the actual values from the order object in the component class.
    - Order column: {{ order.id }}
    - Date column: {{ order.orderDate }}
    - Products column: {{ order.lines.count }}
    - Total column: {{ getOrderTotal(order.lines) }}
23. See that getOrderTotal() method? Add that to dashboard.component.ts:

```
getOrderTotal(lines: Array<OrderLine>) {
  return lines.reduce(
    (p: number, ol: OrderLine) =>
      (p + (ol.price ?? 0) * (ol.quantity ?? 0)),
    0)
}
```

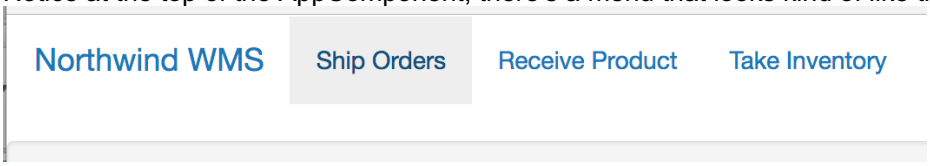
24. Run and test. You should be seeing a list of all the orders on the page with line totals.

Routing Lab

So far we have a few components but you'd be hard-pressed to call it a site or a single-page app. None of the components allow us to navigate to any of the others. We'll fix that in this lab. By the time we're through, our users will be able to visit any of our pages and get to any other.

Defining the routes

1. Notice at the top of the AppComponent, there's a menu that looks kind of like this:



2. Click on any of the three options. They don't do anything yet.

Creating the routing module

The first step would be to design the routes. In this case they're pretty obvious since so far we only have five places to go. So let's go with these routes:

Component	URL
Dashboard	/
Orders to ship	/ship
Receive product	/receive
Ship a single order	/ship/<orderID>
Take inventory	/inventory

3. Create a new file called app.router.ts.
4. Create an array of routes. Here's a start:

```
const routes = [
  {path: "ship", component: OrdersToShipComponent},
  // Okay, now you fill in the other four
];
```

5. Pass that array into .forRoot(). Something like this will work:

```
export const AppRoutingModule = RouterModule.forRoot(routes);
```

Adding your new routing module

Since this is a bona fide module, we need to *imports* it in our main module.

6. Open app.module.ts.
7. Find the imports array in the @NgModule annotation. Add your routing module to it.

Creating a place for the pages to go

The router now knows all your routes and their components. It just needs a place on the page to render them.

8. Remember where we had been hard-coding the sub-component in app.component.html? Let's replace those hardcodes with a <router-outlet>:

```
<main>
  <router-outlet></router-outlet>
</main>
```

9. Run and test. When you navigate to the root of your site, you should see AppComponent hosting your DashboardComponent.

You're probably thinking, "All that to get us back to where we started?!?" Be patient, the payoff is coming.

The payoff: Making all the routes work

10. Edit app.component.html and find the link for "Ship Orders". Make it look like this:

```
<a class="nav-link" [routerLink]="'ship'">Ship Order</a>
```

Basically you're just changing the href="#" to a [routerLink].

11. Use that same pattern for "/receive", "/" , and "/inventory".
12. Run and test. You should be able to use the nav menu to see the four components and this navigation menu appears at the top of every one.

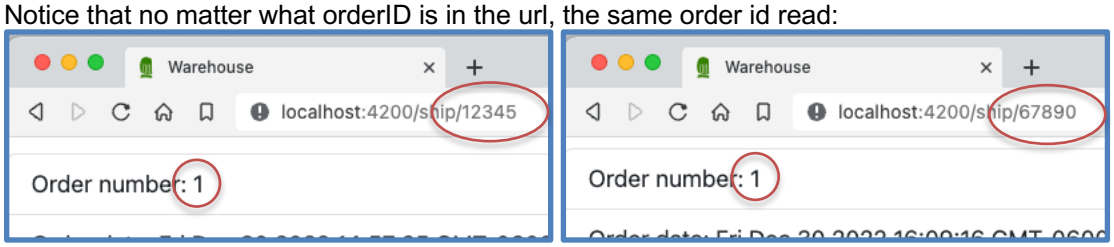
Programmatically activating a route

Remember that in DashboardComponent and OrdersToShipComponent we have lists of orders that need to be picked and shipped. As of now when you click on them, they do nothing. Let's fix that.

13. Open dashboard.component.html and add a [routerLink] to each order's . The url they're routing to should be 'ship/<orderID>'.
14. Run and test. You'll know you have it right when you can click on any order and get routed to the ShipOrderComponent with the proper orderID in the url.

You might wonder about the OrdersToShipComponent. Shouldn't we do the same with OrdersToShip as well? Ah! We have special plans for that one in a future lab, so be patient.

Notice that no matter what orderID is in the url, the same order id read:



That's because we've hardcoded it in orders-to-ship.component.ts. What we should do instead is read the parameter from the ActivatedRoute.

15. Open ship-order.component.ts. Look at the ngOnInit method. Add the code to read the order ID from the route parameters. (Hint: you will need to read _route.snapshot.params after having declared _route as an ActivatedRoute).
16. Go ahead and set order.id to the order ID read in as a route parameter.
17. Run and test. You should be able to click on any order in the DashboardComponent and see the order ID dynamically set on the page.

As of now, the only thing you'll see change on the page is the order ID but very soon we will be reading the order details live from a RESTful service via Ajax. Stay tuned for that!

Once you can route to all the components and see a route value passed from one to another you can be finished!

Event Binding Lab

When our customers order a product we have to pick the products, pack them, and ship them. Our pickers have to know where to get the product from our warehouse. Let's create a method for them to get the best location for that product.

The ship order component

1. Edit your ship-order.component.ts file. Add a new method called getBestLocation(orderLine). Note that it will receive an orderLine object. For now it should just set orderLine.locationID to "01A1A" and then console.log it.
2. Now edit your ship-order.component.html and find the 'Get best location' button. Add an angular click event handler to it. When fired, it should call the "getBestLocation()" method you just wrote. Don't forget to pass in the current orderLine.
3. Run and test. When you click your button, the button should be replaced by a location.

Cool! Do you see the "Mark as shipped" and "Problem" buttons? Let's work on them.

4. Wire up the "Mark as shipped" button to run a method called markAsShipped(order) which receives in an order object. This method should change the order's status from 0 to 1.
5. Wire up the "Problem" button. It should run a markWithProblem() method which also receives an order object. It should set the order's status to 2.
6. Run and test. You'll know it's working when the Order status and the instructions change as you hit the buttons.

Receive inventory

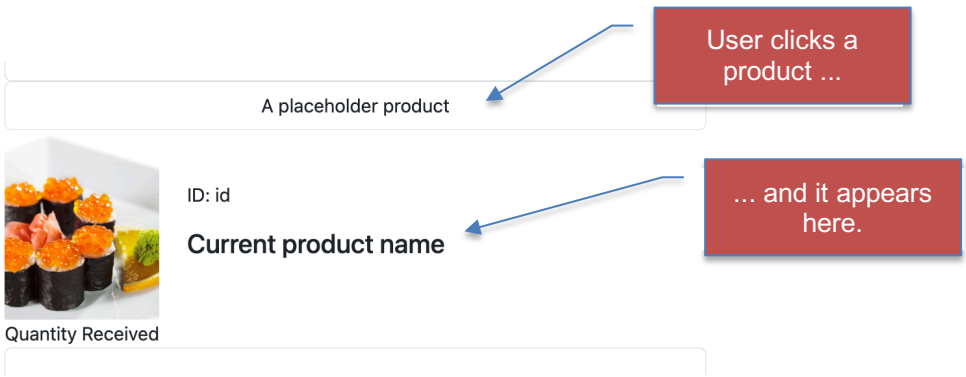
Let's turn our attention to the ReceiveProduct component. This is where the user receives a shipment of product into inventory.

The managers of the warehouse have decided that the users should enter the package tracking number before entering any product. Let's hide everything else on the page until the tracking number is entered.

7. In receive-product.component.ts, create a boolean property called showForm and a string property called trackingNumber.
8. In the template, add a structural directive (hint: *ngIf) so that everything below the tracking number section is gone until this showForm property is true.
9. Create an event handler for the button click. It should call a new method named saveTrackingNumber() which merely sets showForm to true. (Later on we can actually save the tracking number.)
10. Run and test. Did the rest of the component show when you click the button?

Setting the received product

On receive-product.component.html there's a button labelled "A placeholder product". This will eventually be one of many buttons, with each button representing a product. When the user clicks on one of those buttons, they will be "selecting" that to be the currentProduct. But until they select a product from the list, there should be no currentProduct. Let's work on that.



11. First, add a currentProduct property to the class.

```
currentProduct?: Product;
```

12. Next, conditionally show the section if a current product is truthy. (Hint: *ngIf="currentProduct")
13. Make the button's click event call a setCurrentProduct method:

```
setCurrentProduct(product: Product = { id: 0, name: "Fake Product" }) {
  this.currentProduct = product;
}
```

14. Run and test. The section is gone until you click the button.
15. In the currentProduct <section> use interpolation (hint: double curly braces) to display your fake currentProduct's image, id, and name.
16. Run and test one more time to make sure you're seeing your fake product details.

Two final buttons

17. Notice that there's a button labeled "Receive product". When the user clicks it, call a method named receiveProduct(). For now it can just console.log(). We'll make it do something meaningful later.
18. Run and test. Make sure that when you click the button something is logged to the console.
19. Do the same for the "Finished receiving" button. It should call a finishedReceiving() method which just logs something to the console.
20. Run and test that as well.

When you have your buttons working, you can be finished. See how simple event handling can be?

Forms and Two-way Binding Lab

An order is ready to ship when all of the items have been picked. So the "Mark as shipped" button should not be clickable until all the order lines are picked. We're going to disable the button until every line is marked as picked.

1. Edit ship-order.component.ts. Create a method called isReadyToShip(order). Note: It should receive an order object.

```
isReadyToShip(order: Order) {
  return order.lines.every((ol:OrderLine) => ol.picked)
}
```

2. Now edit ship-order.component.html. Find the button and do an Angular property binding. Bind the "disabled" property to the return value of the isReadyToShip(order) function. (Hint: use square brackets. Another hint: The "!" character makes a true false and a false true).
3. Run and test. Since none of the lines are picked yet, you'll see that the button is disabled.
4. Now edit ship-order.component.html. Find your "Got it" checkbox. Using banana-in-a-box, 2-way bind it to the order line's "picked" property.
5. Run and test. When you've marked every line as picked, the "Mark as shipped" button should enable. (HInt: if you get errors, remember that you have to import the FormsModule).

Receive Component

Let's move to the ReceiveProduct component. When the user enters a package tracking number, we want it bound back to the business class.

We already have an <input> in the html template for the tracking number. And we have a property in the TypeScript class for trackingNumber. We just need to bind them together.

6. Edit receive-product.component.html. Put a two-way binding on the <input> box to bind it to this.trackingNumber.
7. In the Events Lab you make the button click call a saveTrackingNumber() function. Now, have that function console.log() the tracking number just entered.

Now notice that there's a pattern attribute on the tracking number input box. If the entry matches the pattern, it is valid. We want the user to get immediate feedback if it is valid or not. You're going to add a Bootstrap class of text-bg-success if the number is valid and text-bg-danger if it is invalid.

8. Add a template reference on the <input> like this #tn="ngModel"
9. Add an [ngClass] directive to the <section> where the user enters a tracking number. If the tracking number is a valid format, make the background of the section turn green by applying the text-bg-success class. If it is invalid, apply the text-bg-danger class. It should look like this:

With an invalid tracking number

With a valid tracking number

10. Run and test with these valid numbers

FedEx	12345678901234
UPS	1Z1234590291980793
USPS	9400 1057 3346 4567 2475 01
USPS	9205 5000 3865 8954 7543 00

11. Bonus!! Add tn.touched to your [ngClass] directives to say to only add the classes after the user has had an opportunity to make a change.

Recording the items received

Let's allow the user to receive items. We want to read in the quantity and product ID and add it to the array of products being received when he/she hits the "Receive Product" button.

12. Add a public array called receivedProducts to the component class. This will be an array to keep track of all the products and quantities that are being received. Add a quantity and productName string.

```
receivedProducts:
  Array<Product & { quantity?: number, location?: Location }> = new Array();
productName?: string;
quantity?: number;
```

13. Notice that you have an <input> box for the productName being received. Bind that to productName.
14. There's another input box for the quantity. Bind that to quantity.
15. When the user hits the receive product button you're calling the receiveProduct() method. Modify the method to add the currentProduct and quantity to the receivedProducts array.
16. In the starter HTML you were given, there's a hardcoded <table> of all the items being received. Convert the <tr> to an *ngFor, iterating receivedProducts and show all the product IDs, names, and quantities in a table. Hint: Here's how two of the table cells might look

```
<td><img [src]="'assets/images/productImages/' + product.id + '.jpg'"
         [alt]="product.name" ></td>
<td>{{ product.id }}</td>
```

17. Run and test. Enter a fake product name into the productName <input>. Then click the "placeholder product" button. Then enter a quantity and click the "Receive product" button. You should see your new product/quantity added to the table. Cool, right?

18. Bonus!! Note that after you add your product to the array you still have values in the product name and quantity textboxes. Can you make the values clear? Go ahead and do that.

Composition with Components

The spirit of web components is to allow encapsulation so that our site is easier to maintain and easier to extend. And we have the opportunity to do that right now. If you look at the main page you'll see a link for "Ship Orders" but on our main page, we already have a list of orders to ship. Aha! We have an opportunity for re-use!

1. Open a command window and use the Angular CLI to generate a new component called ListOfOrders. It should exist in the shipping folder. (Hint: use the --flat flag and specify the folder as part of the name ie. shipping/ListOfOrders)
2. Edit Dashboard.component.html and add a tag to show the list-of-orders component. Maybe put it just above the list-of-orders .
3. Run and test. If you can see your new component hosted inside the dashboard, you've just implemented composition.
4. Now edit dashboard.component.html and find the html that creates the list of orders. Cut it from there and paste it into your new list-of-orders.component.html.
5. You'll see various compile errors because properties and methods in the HTML are not in the list-of-orders.component.ts TypeScript class.
6. Fix those errors by copying/cutting from dashboard.component.ts and pasting them into list-of-orders.component.ts.
7.
8. Run and test. You won't see any orders lists because the "orders" property is defined in DashboardComponent but is trying to be displayed in ListOfOrders. We need to pass those orders into the ListOfOrders component. Let's do that in the next few steps. Read on!

Property binding between components

9. Open ListOfOrders and create a new property called orders. Mark it with an "@Input" annotation. (Hint: In order for this to work, you're going to have to import *Input* from @angular/core).
10. Now to pass it in the HTML. Open dashboard.component.html. Find where you've included the tag to list-of-orders. Add this property binding to it:

```
[orders]="orders"
```

    This says to take the orders property from the host component and pass it down into the inner component as orders. (They happen to have the same name but they could be different if you prefer).

11. Run and test again. You should see the list in your dashboard again.

Cool. What we've done will make our app more modular and easier to understand. But we now have an opportunity for re-use.

Reusing a component

Notice that the AppComponent has a menu choice at the top called "Ship Order". This component will present a list of orders that are ready to ship and then allow the user to click on one to see the details for that one order. Hey, that's like ListOfOrders! Let's reuse it.

12. Edit orders-to-ship.component.html and add a tag to display the list-of-orders component. (Hint: Don't forget your property binding.). Remove the hardcoded order-list .

13. Look in the DashboardComponent.ts in the ngOnInit method and copy the creation of this.orders to OrdersToShipComponent.ts.
14. Run and test. Once the list of orders is available, you've successfully used and re-used a component.

15. Bonus! Remember that you had previously created some styles for that list. They're in dashboard.component.css. If you move those styles to list-of-orders.component.css, they'll apply on both Dashboard and OrdersToShip. Go ahead and move them.

Extracting the shipping label

Take a look at an order in the browser. Remember that nifty shipping label at the bottom? That should probably be relocated to a new component. Let's extract it as well but this time you'll do it with fewer instructions.

16. Create a new component called shipping/ShippingLabel.
17. Include it at the bottom of ship-order.component.html. Make sure you pass the order as a property binding.
18. Move the shipping label's HTML into shipping-label.component.html.
19. Move the shipping label's CSS into shipping-label.component.css.
20. Edit shipping-label.component.ts and add the order as a property. Mark it with @Input.

You'll know you have the refactor correct when you can see the shipping label again.

Got it? Way to go!

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 "

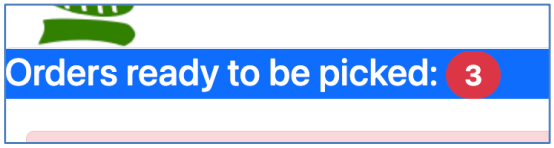
You should see the right order date, ship via, shipping address, the right products in the list, and the right shipping address in the shipping label at the bottom of the component.

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

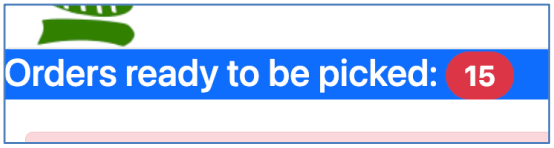
Bonus! Make the badge work

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



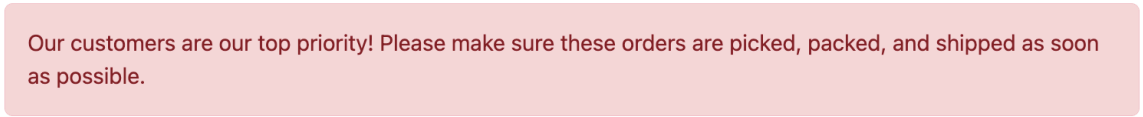
Yep! Reflects the correct number of orders

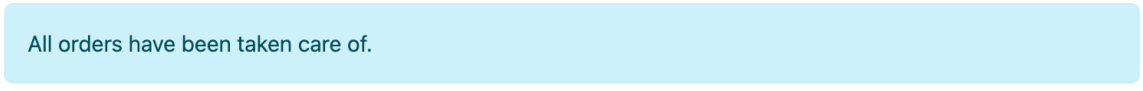


28. Make that reflect the real number of orders.

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)

Observables Lab

RxJS and Observables are maybe the most difficult thing to understand about Angular. They demand that you think differently. You have to use Reactive programming thinking which is very different than how most devs think currently. Don't be discouraged! It takes many, many reps to master it. Most devs have to work with it for months before they get comfortable.

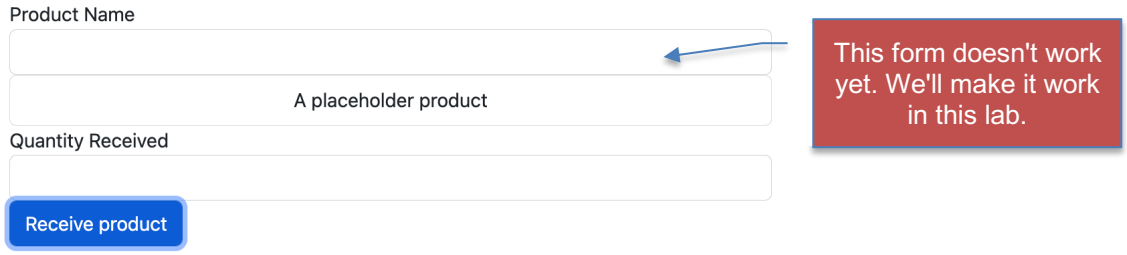
Because of all this, we're going to give you a lot more details in this lab. We're going to give you more pre-written code so you struggle less. If you don't like this, just don't peak at the answers we're giving. Try to do as much of it as you can without using our code. Write your own if you want to!

Ready? Here we go!

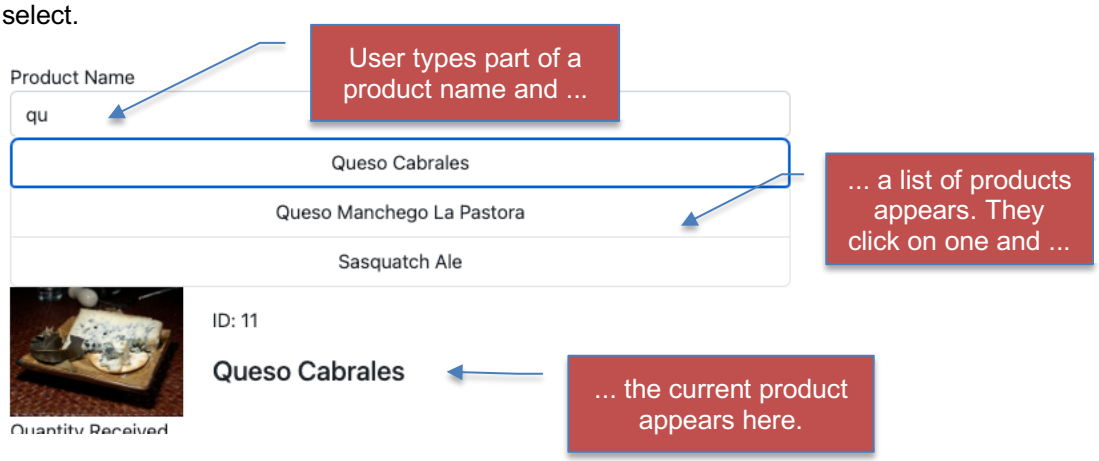
An autosuggest input

Our users receive new products into the warehouse using the ReceiveProduct component where they enter a product name and a quantity. Let's give them a list of products when they enter a product name.

Remember ReceiveProduct? There's an <input> marked "Product Name". Below it is a fake placeholder product.



As the user types into the <input> box, we're going to make an Ajax call to the server to get back all products whose name matches that. Then we'll put each product into the list for the user to select.



1. First, inject an HttpClient into the component's constructor.

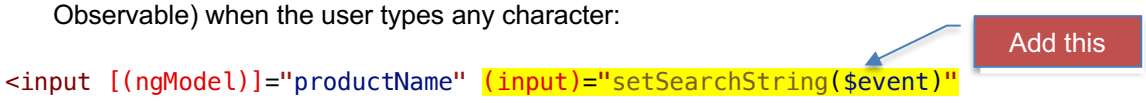
2. Next, add an observable to receive-product.component.ts:

```
foundProducts$?: Observable<Product[]>;
```

3. Create a function to grab the value from the <input> box and set the Observable to fetch from our server. Angular and TypeScript do not make this trivial, so here's a solution that will work:

```
setSearchString(event: Event) {
  const searchString: string | null = (event.target as HTMLInputElement).value
  this.foundProducts$ = searchString ?
    this._http
      .get<Product[]>(`/api/products?search=${searchString}`)
    : undefined;
}
```

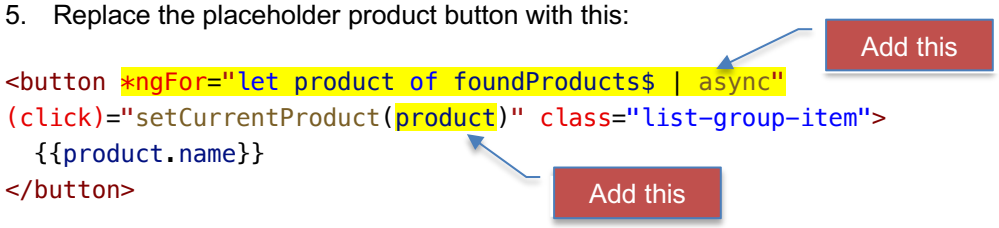
4. Now wire up the productName <input> box to reset the search string (and thus the Observable) when the user types any character:




```
<input [(ngModel)]="productName" (input)="setSearchString($event)"
class="form-control" id="productName" />
```

If you console.log(searchString) inside the setSearchString() method, you'll see your text get logged but no requests are sent yet. Why? Because Observables are lazy; we need to subscribe to them before they do their thing. We'll do that with an async pipe.

5. Replace the placeholder product button with this:




```
<button *ngFor="let product of foundProducts$ | async"
(click)="setCurrentProduct(product)" class="list-group-item">
  {{product.name}}
</button>
```

6. Run and test. Enter some text into the <input> box and watch the <button>s appear and disappear.

Getting locations to put the products away

Let's do one last thing on the ReceiveProductsComponent. Let's get the best location for each product.

After a product is selected, we should reach out to the API server to get all its locations.
But there's a problem. That URL returns all the locations and we only want one. So we should process the observable to find the location with the fewest of that product in stock. Also, if this is a new product for us, there is no location already. So if there are no locations, we will tell the user "Any open slot".

7. Edit receive-product.component.ts. Create a new property to hold the location:

```
bestLocation?: Location;
```

8. Find the setCurrentProduct() method. In there, make an HTTP GET call to /api/locations/forProduct/<ProductID>. Go ahead and subscribe to it where you'll simply console.log() the response.
9. Run and test by selecting a product. Look in the console. You should see an array. It will have zero or more locations in it.

Remember, we need that observable to return one location or {id:"Any open slot"}.

10. Add a .pipe() to the observable before the subscribe.
11. In the .pipe(), put a tap(console.log). In fact, put as many tap()s as you like while you're debugging this to help you develop. Don't forget to remove them before you're finished.
12. In the .pipe(), test to see if the array of locations is empty. If so, return an array with one location whose id is "Any open slot". Otherwise, return the regular array:

```
mergeMap(locs => iif<Location[], Location[]>(() => locs.length == 0, of([{ id:
"Any open slot" }]), of(locs))),
```

13. Now let's add another pipeable operator to pick the array member with the smallest quantity.

```
map<Location[], Location>(locs => locs.reduce(
  (a, b) => (a.quantity ?? 0) < (b.quantity ?? 0) ? a : b
)),
```

14. Add one more pipeable operator to pluck out the "id" property.

```
map<Location, string | undefined>(loc => loc.id),
```

15. If you run and test at this point, any console.log()s should confirm that you're processing the locations. You'll get a location id or "Any open slot".
16. In the subscribe(), set this.bestLocation to the Location coming into the observer.
17. And here's the payoff, in the receiveProducts() method, you're .push()ing onto this.receivedProducts. We just need to add bestLocation to the line:

```
this.receivedProducts.push({ ...this.currentProduct, quantity: this.quantity,
location: this.bestLocation ?? {} });
```

18. Alright! Run and test. When you can see a good location.

Image	ID	Name	Quantity	Location
	51	Manjimup Dried Apples	12	Any open slot
	8	Northwoods Cranberry Sauce	67	07E0B
	34	Sasquatch Ale	42	07B6A

RAP: These can be put somewhere in an advanced lab or bonus or something

Getting locations for those products

Remember where we're hardcoding a location from which to pick? Let's get an actual location from the database.

RAP: TAKE THIS OPPORTUNITY TO DO THE OBSERVABLE PIPABLE OPERATORS

19. Still in ship-order.component.ts, find the getBestLocation method where you hardcoded the return.
20. Change that hardcode to hit the GET endpoint for /api/locations/forProduct/:productID. Of course you're passing the product ID in that productID parameter.
21. Note that what comes back from the API call is an array of Locations so you'll want to convert the response to Array<Location> using the map function.
22. Make sure you're subscribing a function. In that success callback, set your orderLine's locationID.
23. Run and test. When the button is pressed by the user, a real location should be displayed. You'll know you got it right when there's a different location for each product.

Set the shipped or problem flag

24. Still in ship-order.component, find the method you're running when the "Mark as problem" button is clicked.
25. Make that send a PATCH request to /api/orders/<orderID>/MarkAsProblem. (Note: the PATCH protocol requires a body but our service ignores it, so send an object -- even an empty one). We're doing nothing with the response so no need to map anything. Don't forget to subscribe() or the observable never gets run.
26. Run and test. Mark one as problem. Then look at your list of orders and you'll see that it is no longer ready to be picked because you've said it has a problem. When you can mark it as problem and see it removed from the list, you know you've got it working.
27. Do the same for "Mark as shipped"
28. Run and test them both. When you mark as shipped, you should see that the inventory amount is reduced by the amount shipped.

Remember, you can always look at an order and manually set the order status by directly editing database.json in the server folder. Just find your order and set status to 0.

Services Lab

Let's say that once the user is logged in we want to display their name on each page.

Obviously, we want to have the user log in to one component. Then we could pass the user's identity around using property binding, but that's a whole lot of properties. We'd have to do it in every component. Too much work spread around too many components. Gosh, if only we could somehow share properties and methods between components!

This is where services shine! If we were to inject an AuthService into a Login Component and set its *user* property and then inject that same AuthService into other components, its *user* property could be seen in all of them. Let's work on making that happen.

1. To prepare, create a new type in the shared folder called User.

```
export type User = {
  id?: number,
  username?: string,
  password?: string,
  givenName?: string,
  familyName?: string,
  isAdmin?: boolean
}
```

2. Create a new Angular component called Login. In the class, add a user property and string properties for message, username and password.
3. In the template, create:
   - inputs for email and password,
   - a button to log them in. The button's click event should run a method called login().
   - a to hold a success/failure message for their attempted login.
4. The login method will pretend to log them in and create a fake user object. This object should hold placeholder information for id, username, password, givenName, and familyName. Also have it set this.message to "Successfully logged in".
5. Now let's use the login component. Add a route to it in app.router.ts. Then add a router link to it in the menu at the top of App component.
6. Run and test. Make sure that you can get to the component and that 'logging in' (wink wink) will show a success message and set the user object's properties.

While you're still running, notice that you can navigate to the Receive Product component or the Orders Ready to Ship component. Our goal is to make the user data from Login appear in those other components.

Creating the service

7. In the Angular CLI, create a new service called Auth Service. Something like this might do the trick:

```
ng generate service auth/auth
```

8. Give that service a Boolean property called loggedIn. Initialize it to false.

9. Also give it an optional property called user.

Note that this is a super-simple service so far. It has no methods (yet).

Using the service

10. Inject the service into your Login Component.
11. In LoginComponent's ngOnInit, set its private user property to the injected service's user property. Something like this will do:

```
this.user = this._authSvc.user;
```

12. In LoginComponent's Login() method, pretend to authenticate. Set the service's loggedIn property to true:

```
this._authSvc.loggedIn = true;
this._authSvc.user = {
  // Whatever values you had in there before
};
```

Cool. It looks like the user is being logged in in the LoginComponent. Let's make sure in another component.

13. Inject the AuthService into your App Component as a **public** property.

14. Go into the template and use interpolation to display the user's given name and family name. Put this inside the navbar at the top:

```
<ul *ngIf="authSvc.loggedIn" class="navbar-nav d-flex justify-content-end">
  <a class="nav-link" [routerLink]="'account'">
    {{ authSvc.user?.givenName }} {{ authSvc.user?.familyName }}
  </a>
</ul>
```

15. Run and test. Login and then navigate to any component. Do you see your name appear? Cool! You've just shared data through a service!

Refactoring the service

We've just demonstrated that the purpose of a service is to share -- just properties so far but we can share methods as well.

16. Give the Auth Service a method called *authenticate*. Move the logic for logging in from the login component to this method. (Hint: this._authSvc.loggedIn becomes this.loggedIn and this._authSvc.user becomes this.user). These will still be hardcoded placeholder values.
17. Back in the LoginComponent, call the authenticate() method from your AuthService.
18. Run and test. Can you still log in? If so, we've moved the work of logging in from a component into the AuthService where it really belongs.

Making an HTTP call in the Service

Last step to make is more realistic. Let's use the login endpoint from our API server.

19. Edit the authenticate() method in the AuthService
20. Make sure it is receiving a username and password as parameters.

21. Make an HTTP GET call to `/login/${username}/${password}`[†]. The API will return an entire user object for a good username/password combination.
    Hints:
    - You can edit the database.json file in the server folder to get good username/password combinations. You can even add your own user records if you like.
    - You'll need to inject an HttpClient into the AuthService's constructor.
    - The authenticate() method should return an Observable so it can be activated in other components. ie. They subscribe when they're ready like on a button click.
    - If you still can't work it out without some more help, here's a way to do it:

```
authenticate(username?: string, password?: string): Observable<User> {
  return this._http.get<Array<User>>(`/api/login/${username}/${password}`)
    .pipe(
      map(users => users[0]),
      tap(user => this.user = user),
      tap((users) => this.loggedIn = true)
    )
}
```

22. You'll change your LoginComponent's login() method from this:

```
login() {
  this._authSvc.authenticate(this.username, this.password)
  this.message = "Successfully logged in";
}
```

to this:

```
login() {
  this._authSvc
    .authenticate(this.username, this.password)
    .subscribe({
      next: user => this.message =
        `Welcome, ${user.givenName}. You're successfully logged in`,
    });
}
```

23. Run and test. It should work for good credentials and should set the givenName and familyName from the database instead of your hardcoded values.

[†] The way this authentication is set up is not secure. You'll never actually make a GET call to really authenticate. In the real world it'll be a POST with the credentials in the body. But we're trying to keep this simple for you.

Bonus!! Refactoring to the repository pattern

Only do this if you have plenty of extra time and want a deeper challenge. There is a design pattern called the repository pattern that allows a developer to decouple database logic from their business logic[‡]. Let's use a form of it in our application and refactor all of the data persistence out of our components and put it into a few services.

24. Using the Angular CLI, create a new service called OrdersRepository. In it, write a new method called getOrdersReadyToShip(). It should return an observable array of orders.
25. Give it a method called getOrder() which receives in an order ID and returns an observable of orders.
26. Write two methods called markAsShipped() and markAsTrouble(). Each should receive an order ID and return an observable.
27. Open the Orders To Ship component's TypeScript file. Inject your new OrdersRepository service.
28. Find in OrdersToShip where you're making an Ajax call to get the orders that are ready to ship. Cut that logic and instead make a call to your service's getOrdersReadyToShip() method.
29. Run and test. Adjust until you get your orders list working again.

Cool! You're using the repository pattern!

30. Open the ShipOrder component. Inject the repository in there as well.
31. Find where you're marking the order as shipped and as in trouble. Move the Ajax calls to your service methods and call them instead of making the Ajax calls directly.
32. Do the same where you're reading the order via Ajax.
33. Run and test.
34. Edit the dashboard component. Use your repository to read orders ready to ship in there as well.

Now if the logic to read orders changes, we can adjust in the repository instead of in both components. That's just one of the major benefits of the repository pattern.

35. Extra bonus!! If you have extra time, implement a repository for reading and writing product data.

---

‡ Read more about the repository pattern if you are interested: https://martinfowler.com/eaaCatalog/repository.html

Pipes Lab

Using built-in pipes

1. No offense, but your dates are ugly. Look at your list of orders either in the dashboard or in orders to ship. You can also see some ugly dates in ship order.
2. All of these can be improved by piping the date through the date pipe. Go ahead and do that now, choosing whatever formats you think are best.
3. Run and test. When the dates are looking good, you can move on.

A custom location pipe

To make it easier on our new and temp warehouse workers we should decipher the locationID. We're going to create a custom pipe to do that. The locationID has the format **AABSI** where

AA	is the	Aisle
B	is the	Bay
S	is the	Shelf
I	is the	Bin

First, let's see how the pipe will be used before we create it.

4. Open your ShipOrderComponent in an IDE. In the template, find where you're displaying the location. Add a pipe right after it:

```
{{ line.locationID | location }}
```

5. Run and test. Angular complains about the location pipe not existing.
6. Create a new pipe called "location"

```
ng generate pipe shared/location
```

7. In that pipe you have a transform function that will receive a string ... the locationID. You want it to return a string in this format:

Aisle AA, Bay B, Shelf S, Bin I

8. Write that, then tune it until it is working.
9. And once you've got it working well, let's reuse it! Open your receive-product.component.html page and implement it there also.
10. Oh darn! You'll notice that if the locationID is "Any open slot" that it gets messed up. Change your location.pipe.ts to only transform the ID if it matches the format above.

When you've got it working in all cases, you can move on.

Make a ship via custom pipe

Notice that on the ShipOrder component there is a ship via property being bound. We'd like that to display the name of the shipper rather than the raw number.

11. Create a new pipe called shipVia. It should translate as follows:

Ship via	Shipper name
1	Federal Express
2	UPS
3	US Postal Service

12. Your goal will be to do an interpolation like this {{ order.shipVia | shipVia }} and have the name of the shipper be displayed.

13. Go ahead and implement it in the ship order component. Once you can see a different shipper for various shipvia values you can be finished.

Modules Lab

Note: Not yet finished.
Goal: Too much stuff in the main module. Split it into shipping module, a receiving module, an inventory module and a shared module. Give detail on what to remove from sections and add to others.

1. Using the Angular cli, generate four new modules:
- shared
- shipping
- receiving
- inventory

This may be more than we need but as long as we're creating modules, might as well go all the way with it.

2. Double-check to make sure that the modules are all in folders with the same name. ie. shared.module.ts is in the shared folder, shipping.module.ts is in the shipping folder, etc.

Reallocating a pipe

The LocationExpandPipe is being used by components in multiple locations so maybe we should put it in a shared module.

3. Open shared.module.ts and find the declarations key in the @NgModule directive. Go ahead and add LocationExpandPipe to it. Are you seeing an error in your IDE? Don't forget to import the LocationExpandPipe class.
4. Open app.module.ts and remove all references to LocationExpandPipe.
5. Run and test. It should no longer work. Why? Because LocationExpandPipe is now part of the shared module but your components which are part of AppModule can't see SharedModule. Let's fix that.
6. In AppModule, add SharedModule to the imports array. This will allow things in the SharedModule to be seen in the AppModule.
7. Run and test again. It still doesn't work. Why? Because we need to mark the pipe as a candidate to export.
8. Find the exports array and add the LocationExpandPipe to it.
9. Now run and test. It should be working again. If not, massage the files until it does.
10.
11. Do the same with your XXX service. You'll want to remove it from AppModule and put it in the SharedModule. It will need to be listed in the providers section.
12.

Doesn't need to be a long lab. There are no exciting features here.
stuff in shipping goes into a shipping module.
Stuff in inventory into a inventory module
stuff in receiving into a receiving module
stuff in shared into a shared module
Dashboard fails b/c list-of-orders is in shipping module. Move it to shared.

Component Lifecycle Lab

Goal:

Routing Observables Lab

Goal:

In the

Controlling Observables Lab

Goal:

In the

Zip Code Assessment

This assessment is designed to be written in two hours or less by an individual developer. It should take no special knowledge other than the things you've learned in this Angular course and your prerequisite knowledge of HTML, CSS, and JavaScript.

You are free to refer to your notes, the class materials. You are free to use the Internet to look things up but not to collaborate with another developer.

In this final assessment, your mission will be to create a new website from scratch that will allow your users to look up zip codes given a city and vice-versa.

Setting up the site and form

1. Create a new project called angular-project. Choose "no" for routing and "CSS" for styles.
2. Create a new component called CityLookup. Show this component when the project starts up.
3. Add a form to the component. It should have three <input>s, one each for "city", "state", and "country". Also give it a submit button.
4. When the user submits the form or clicks the submit button, your component should read in those three values and console.log them.
5. Next, add a <h3> below the form that says "Information for (city), (state)" where city and state are the actual values that the user entered in the form.
6. Make this <h3> only show after the user has submitted the form. (hint: create a property in the class called "haveData". When true, show the <h3>).
7. Bonus points for making it look nice with Bootstrap or any other CSS technique. But be careful to not spend too much time on cosmetics. Getting it functional through the last step is much more important.

Getting zip code data from the Internet

Your user can now enter a city, state, and country. Let's show them the data for that city. There is a service available called Zippopotam.us (http://Zippopotam.us) that allows you to send a GET request to http://api.zippopotam.us/country/state/city and get back a list of zipcodes for that city.

8. First, get familiar with the service by making a test request. Open your browser and type in http://api.zippopotam.us/us/md/columbia. Examine the returned JSON data.
9. Do the same for http://api.zippopotam.us/us/md/baltimore. See how they differ? Study the output for a minute. Maybe even keep this open for reference for future steps.
10. Instead of a simple console.log() when the form is submitted, make an Ajax call to Zippopotam.us to retrieve the data for that city.
11. Now display that data on the page below the <h3> you added above.
    - Hint 1: You'll want to iterate the data in the template and show it in s or maybe <tr>s.
    - Hint 2: Use interpolation in the iterator to put each data point in your or <td> or whatever you chose above
    - Hint 3: Zippopotam.us gives us a property unfortunately named "place name" (with the space in it). To interpolate this, use "*place['place name']"* instead of *"place.place name"*. Do the same with *"post code"* also. (Sheesh!)

Getting city data

12. Create a new component called ZipCodeLookup.
13. Make this the startup component.
14. It should have two <input>s for country and zip code. Also give it a button to submit the data. Feel free to copy/paste from your other page if you want to.
15. When the button is pressed, it should make a call to Zippopotam.us with the city and zip code like this http://api.zippopotam.us/country/zipcode. Again, read the response and display it on the page. (Hint: This endpoint, like the one above, returns an array so it should be iterated).
16. Run and test.

Setting up routing

Now that we have two components, let's make it a cohesive site by allowing navigation between them.

17. First, add a file called app.router.ts. (Hint: no need for ng generate. Just do it manually).
18. In that file create your routing array. (Hint: it's a JavaScript array of objects, each with a path and a component). Process it with RouterModule.forRoot().
19. Import this new configured router module into the AppModule. Also don't forget to import RouterModule.
20. Change your ZipCodeLookup component. The user should be able to click on the city name and navigate to the CityLookup component.
    - Hint 1: use a routerLink
    - Hint 2: They call it "place name" instead of "city"
21. Change your CityLookupComponent. When the user clicks on the zip code, it should navigate to the ZipCodeLookupComponent.
    - Hint: They call it "post code" instead of "zip"
22. Run and test. Make sure it is working before you go on.
23. Now change both of those links to include route parameters in the URL.
    - Clicking on the city name in ZipCodeLookup should also add the country, the state and the city to the url.
    - Clicking on the zip code in CityLookup should also add the country and the zip code in the url.
24. Again, run and test. Make sure they appear in the URL and navigation still works.
    - Hint 1: For navigation to work for these two new routes, you're going to have to add to your routing table.
    - Hint 2: The new routes will need route parameters
25. Change both components to read the route parameters from the url and pre-fill the textboxes that are already on the page.
    - Hint: you'll need to use dependency injection and put a couple of lines of code in either the constructor or ngOnInit
26. Once your textboxes are being filled from the route parameters, change the ngOnInit() method to check that the values are indeed being supplied and if so, go ahead and make the respective Ajax calls to fetch the data.

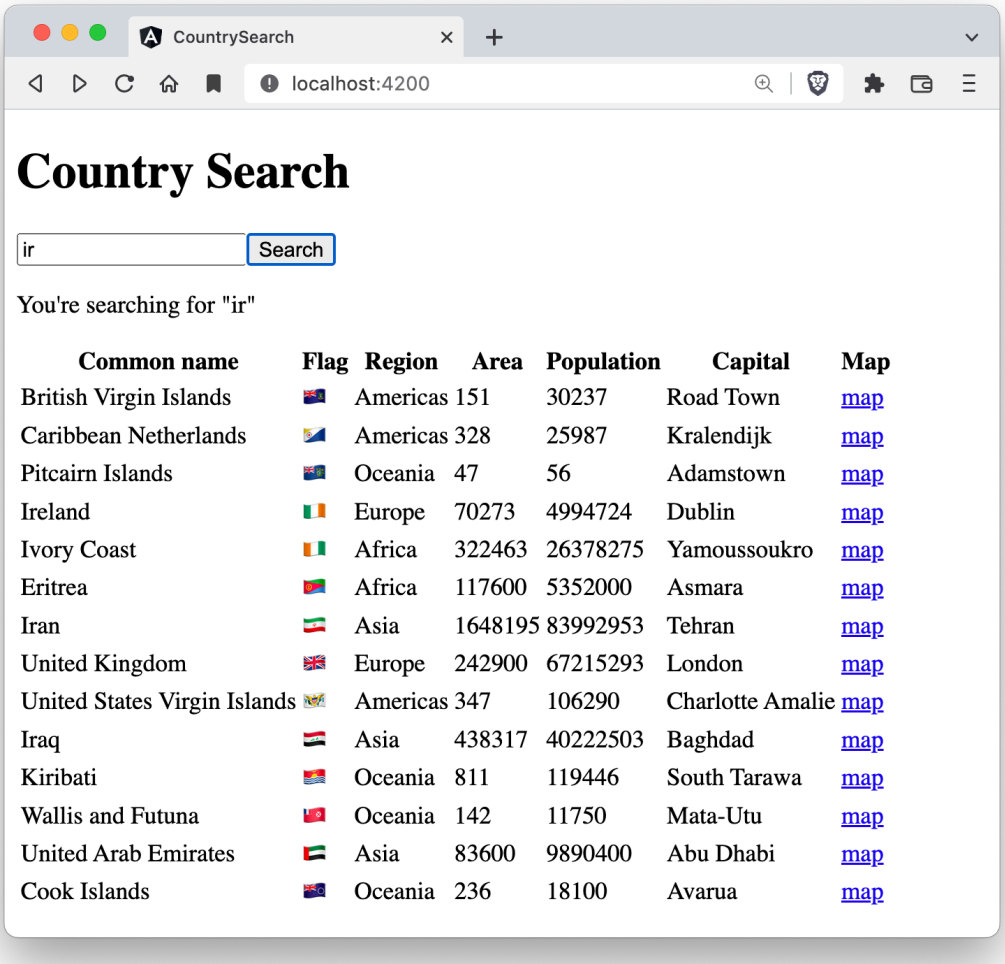
You'll know your site is working properly when you can fill in the data on either page, click the button to fetch data from the API server, then click on the results and navigate to the other page which then shows the proper data without you having to click any buttons.

Countries Assessment

This assessment is designed to be written in three hours or less by an individual developer. It should take no special knowledge other than the things you've learned in this course and your prerequisite knowledge of HTML, CSS, and JavaScript.

You are free to refer to your notes, the class materials. You are free to use the Internet to look things up but not to collaborate with another developer.

In this final assessment, your mission will be to create a new Angular web app from scratch that will allow high school students to look up country data. They'll enter a country name or partial country name and your app will show them countries that match their search criteria. It'll look kind of like this:



Common name	Flag	Region	Area	Population	Capital	Map
British Virgin Islands		Americas	151	30237	Road Town	map
Caribbean Netherlands		Americas	328	25987	Kralendijk	map
Pitcairn Islands		Oceania	47	56	Adamstown	map
Ireland		Europe	70273	4994724	Dublin	map
Ivory Coast		Africa	322463	26378275	Yamoussoukro	map
Eritrea		Africa	117600	5352000	Asmara	map
Iran		Asia	1648195	83992953	Tehran	map
United Kingdom		Europe	242900	67215293	London	map
United States Virgin Islands		Americas	347	106290	Charlotte Amalie	map
Iraq		Asia	438317	40222503	Baghdad	map
Kiribati		Oceania	811	119446	South Tarawa	map
Wallis and Futuna		Oceania	142	11750	Mata-Utu	map
United Arab Emirates		Asia	83600	9890400	Abu Dhabi	map
Cook Islands		Oceania	236	18100	Avarua	map

Creating the app and search form

1. Create a new Angular project called country-search. Choose "no" for routing and "CSS" for styles.
2. Create a new component called CountrySearch.
3. Edit AppComponent. Remove the boilerplate. Instead, show your new CountrySearch component.
4. Run your project's auto-created unit tests by running *ng test*. Expect some of the AppComponent tests to fail. Delete any tests from app.component.spec.ts that no longer make sense. We just want you to start with 100% passing tests.

Now let's do a little TDD.

Unit testing your component

Remember, TDD starts with you writing a failing unit test. Then you write code until the test passes. Let's write that first unit test.

5. Edit country-search.component.spec.ts. Write a unit test for 'displays the search criteria to the user'. It will make sure that whatever the user puts in the input box is echoed back out to them on the page. This unit test should ...
   - Locate the input box by its id 'searchString'.
   - Assert that it is not null.
   - Put a value in the input box – any value you like.
   - Locate the #feedback paragraph
   - Assert that it exists.
   - Assert that its textContent contains the value you entered in the textbox.
6. Make the test pass by adding a <form>, <input> and <p> to the HTML file and adding a searchString property to the TypeScript file.
   Hints:
   - You'll need to imports FormsModule/ReactiveFormsModule in app.module.ts,
   - You'll need to imports FormsModule/ReactiveFormsModule in country-search.component.spec.ts in the TestBed.configureTestingModule()
   - Remember about someNativeElement.dispatchEvent() and fixture.detectChanges()

Got it passing? Cool.

Next we want to make an Ajax GET request from the service at http://RESTCountries.com. We could make our Ajax call in the component class, but that's not a very clean solution. Instead we're going to ask you to create a *service*.

Creating the CountryService

7. Generate a new service called CountryService.
8. Re-run your unit tests again. You should see an automatically-created smoke test for this CountryService.
9. Edit country.service.spec.ts. Make its TestBed config look like this:

```
TestBed.configureTestingModule({
  imports: [HttpClientModule]    // Needed to make HTTP calls in the unit test
});
```

10. Write a new failing unit test. Here's a start for you. Copy this into country.service.spec.ts.

```
it('should retrieve countries', (done) => {
  service.getCountries("Tuvalu")
    .subscribe({
      next: (countries) => {
        // TODO: Assert that the length of the countries array is 1
        const [country] = countries;
        const { area, name: { common }, region, subregion } = country;
        // TODO: Assert that the area is 26
        // TODO: Assert that the common is "Tuvalu"
        // TODO: Assert that the region is "Oceania"
        // TODO: Assert that the subregion is "Polynesia"
        done();
      }
    });
});
```

11. In that code above, there are TODOs. Follow their instructions.

Got those lines written? Good. You have yourself a failing test.

12. Now write the code in country.service.ts to make it pass.
    Hints:
    - Don't forget to imports HttpClientModule into the app.module.ts
    - You'll need to inject and HttpClient into the constructor of your CountryService.
    - Make sure your service returns an Observable.

Once your test is passing, you can move on.

Using your new service in the component

13. Edit your country-search.component.ts file. Use dependency injection to get an instance of your CountryService and call it _countryService.
14. Behind the <button>'s click event or the <form>'s submit event, call this._countryService.getCountries(searchString).
15. Subscribe to the observable returned from the service to retrieve all the country data.
16. If there's an error, it should be console.error()ed.
17. For each country returned from the service, put a row in the table. Use the above screenshot for a guideline.

Zooming into the flag image

18. The flag may be hard to see. So when the user hovers over it with their mouse, increase the size of the flag using a CSS transform property. Make the flag at least three times its current size.
19. We want the increase of size to happen gradually over one second. (hint: use a transition)
20. If you mouse over the flags, they're zooming in and out. It may look too busy. So delay the transition for one quarter second before increasing its size.

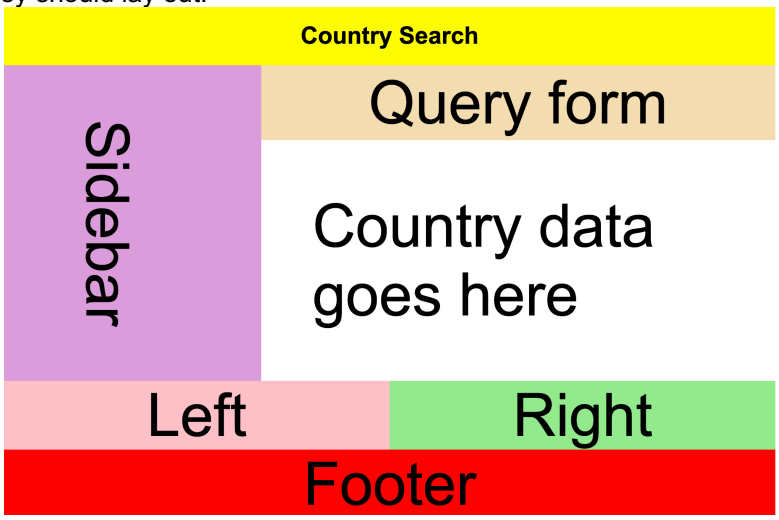
Page layout with a grid

If you haven't already done so, it's probably time to lay out the page. We're going to do that with a grid.

21. Make sure you have a <main> which wraps everything else. This will be your grid container.

22. The grid items will be
    - The <section> with the data table.
    - The <section> with the query form.
    - A new sidebar <section> (placeholder section with a purple background)
    - A page header (yellow background)
    - A page footer (placeholder section with a red background)
    - And two sections for ... other things. They're just above the footer and should take about half the window each. They're pink and light green below.

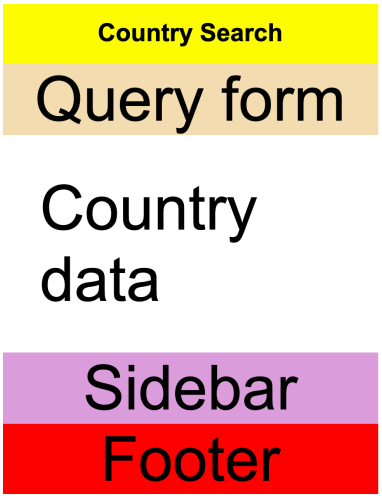
    Here's how they should lay out.



Responsive design

We have way too much data to show on a phone. Let's make the component responsive.

23. First, let's change the layout itself. If you're on a viewport that is smaller than 600 pixels, Make the layout look like this:



    Specifically,
    - The sections we're calling Left and Right are not displayed
    - Sidebar goes below country data
    - All sections take up the entire width

Once you've got the layout changed, we should probably do something similar with the country table itself. There's just too much stuff to look good on a small screen.

24. If the viewport is more than 600px show the entire table. But if the viewport is less than that, don't display the region or capital columns.

25. Bonus!! If you have extra time after finishing the last section, come back to this and adjust the widths and columns shown to tune your look and feel.

Mocking the service

Let's do one last unit test.

26. In country-search.component.spec.ts, write a test to make sure that if the user enters "Tuvalu" in the search input box and then clicks the button, the region "Oceania" is displayed on the page. Here's one way to make that work:

```
it('displays the country fetched', waitForAsync(() => {
  //Arrange
  const inputBox = fixture.nativeElement.querySelector('#searchString')
  const button = fixture.nativeElement.querySelector('button');
  const table = fixture.nativeElement.querySelector('table')
  inputBox.value = "Tuvalu";
  //Act
  inputBox.dispatchEvent(new Event('input'));
  fixture.detectChanges();
  button.dispatchEvent(new Event('click'));
  fixture.whenStable().then(() => {
    fixture.detectChanges();
    //Assert
    expect(table.innerHTML).toMatch(/Oceania/);
  })
}));
```

27. Get your test to pass.

Here's a problem, though. If Tuvalu ever changes their region, your test will suddenly start failing and you'll have no reason why. The poor dev who checks their code and sees this test breaking will think their code caused the problem! Your code is dependent on geopolitics!

We need to solve this problem with a mock.

28. Write a mock service that always returns a given region like "Oceania" given "Tuvalu" in the textbox. Heck, use a fake country with a fake region if you like.
29. In your unit test, where you had been providing an instance of the real service, now provide an instance of the mock service.
30. Adjust until your unit test passes again.

If you get to this point, congratulations! You've finished the assessment. I wish you the best of everything in your life and career!