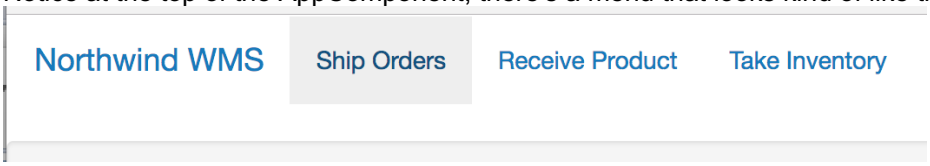
Routing Lab

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.

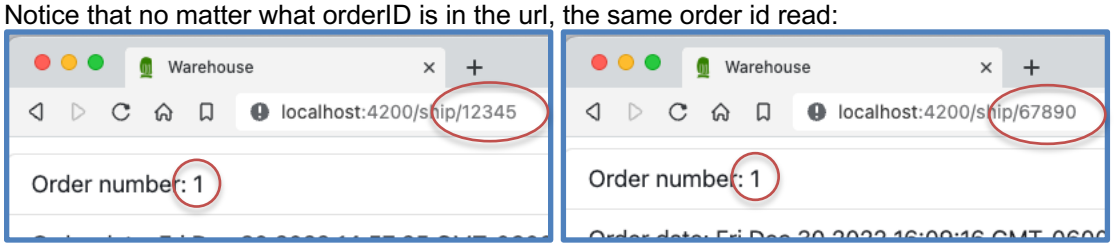
Programmatically activating a route

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!