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!

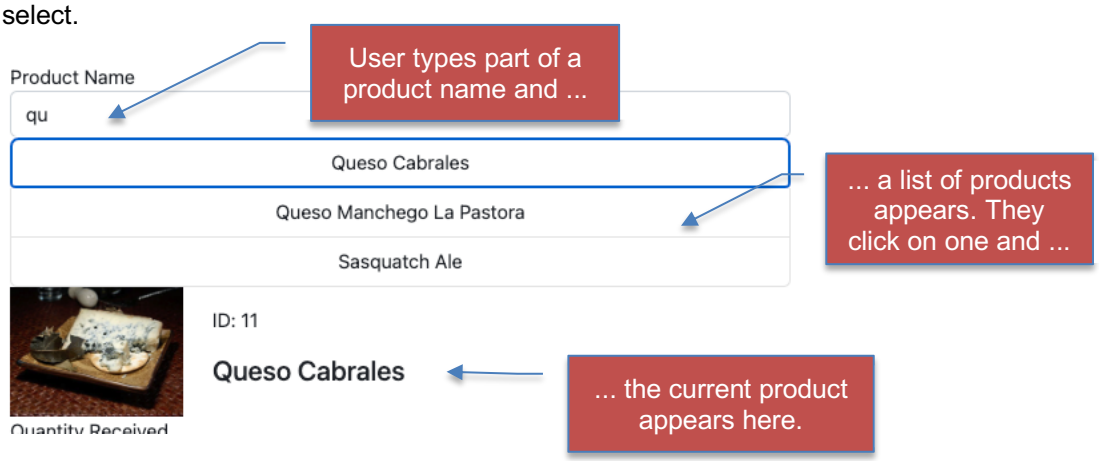
Ready? Here we go!

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.

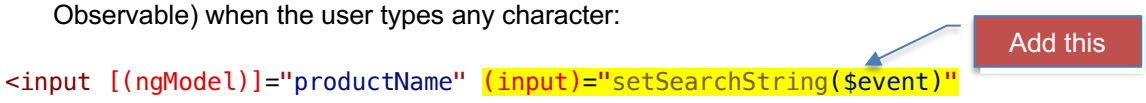
2. Next, add an observable to receive-product.component.ts:

```
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;
}
```

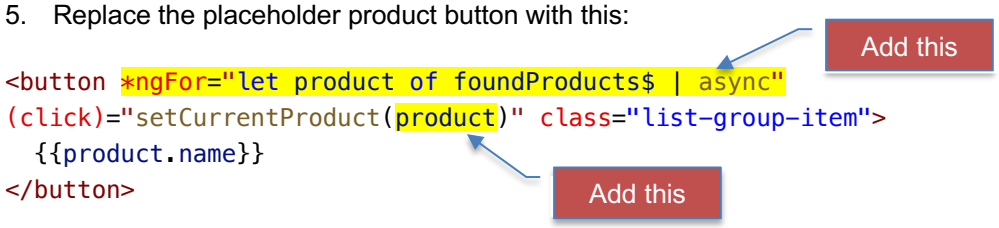
4. Now wire up the productName <input> box to reset the search string (and thus the Observable) when the user types any character:




```
<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