Services Lab

We haven't implemented security yet but in preparation, let's set up a few features. Let's say that once the user is logged in we want to display their name on each page. What are some ways this could be done?

Obviously, we don't want to have the user log in on every component. 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.

This is where services shine! If we were to inject a userService in every component and set its *user* property in a login component, it could be seen in all of them. Let's work on making that happen.

1. Create a new Angular component called Login. In the class, add string properties for username and password.
2. In the template, create:
    - inputs for username 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.
3. The login method will pretend to log them in and create a new user object. This object should hold whatever information you want it to, but at minimum it should have a userid, username, password, givenName, and familyName. Set these to any fake values in the login method.
4. 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.
5. 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.
6. While you're still running, navigate to the Receive Product component or the Orders Ready to Ship component.

Our goal is to make the user data appear in those other components.

Creating the service

7. In the Angular CLI, create a new service called login service. It should probably live in the shared folder. Something similar to this might do the trick:

```
ng generate service shared/Login
```

8. Give that service a property called loggedIn. Initialize it to false.
9. Also give it a property called user. It can have the same properties as in the Login component above (userId, username, password, etc.).

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._loginService.user;
```

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

```
this._loginService.loggedIn = true;
```

13. And set all of the user properties:

```
this.user.userid = 123;
this.user.username = "joKim";
this.user.givenName = "Jo";
this.user.familyName = "Kim";
```

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

14. Inject the LoginService into your Receive Product component.
15. In ReceiveProductComponent's ngOnInit, set a public user property to the service's user property. Maybe do something like this:

```
this.user = this._loginService.user;
```

16. Go into the template and use interpolation to display the user's given name and family name.
17. Run and test. Login and then navigate to the receiving component. Do you see your name? 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 data in this case but we can share methods as well.

18. Give the login service a method called *authenticate*. Move the logic for logging in from the login component to this method.
19. Back in the LoginComponent, call the authenticate() method from your LoginService.
20. Run and test. Can you still log in? If so, we've moved the work of logging in from a component into the LoginService where it really belongs.

21. Bonus!! Add that user service to all your other components and display the user's name in them.

Refactoring to the repository pattern

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 pull all of the data persistence out of our components and put it into a few services.

22. 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.
23. Give it a method called getOrder() which receives in an order ID and returns an observable of orders.
24. Write two methods called markAsShipped() and markAsTrouble(). Each should receive an order ID and return an observable.
25. Open the Orders To Ship component's TypeScript file. Inject your new OrdersRepository service.
26. 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.
27. Run and test. Adjust until you get your orders list working again.

Cool! You're using the repository pattern!

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

28. Open the ShipOrder component. Inject the repository in there as well.
29. 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.
30. Do the same where you're reading the order via Ajax.
31. Run and test.
32. 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.

33. Bonus!! If you have extra time, implement a repository for reading and writing product data.