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 his/her 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 that service in every component and set a 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 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 values to whatever you like in the login method. You can use an Ajax call to get a user from /api/customers if you want to.
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 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 --module=app.module
```

8. Give that service a property called user. It doesn't need to have any properties here unless you want it to.

Note that this is a one-liner service so far. Super-simple.

Using the service

9. Inject the service into your login component. After the user has logged in, set this service's user property equal to the user entity you created in the component.
10. Now inject the service into your receive product component.
11. In the ngOnInit, set a private user property to the service's user property.
12. Go into the template and use interpolation to display the user's given name and family name.
13. 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.

14. Give the login service a method called authenticate. Move the logic for logging in from the login component to this method.
15. Back in the login component, call the authenticate() method from your LoginService.
16. Run and test. Can you log in?
17. 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. (Read more here if you are interested: https://martinfowler.com/eaaCatalog/repository.html). 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.

18. 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.
19. Give it a method called getOrder() which receives in an orderID and returns an observable of orders.
20. Write two methods called markAsShipped() and markAsTrouble(). Each should receive an orderID and return an observable.
21. Open the orders to ship component's TypeScript file. Inject your new OrdersRepository service.
22. Find in there 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.
23. Run and test. Adjust until you get your orders list working again.

Cool! You're using the repository pattern!

24. Open the ship order component. Inject the repository in there as well.
25. Find where 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.
26. Do the same where you're reading the order via Ajax.
27. Run and test.
28. 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.

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