Exercise 10 - Adding services to our application

Objective

To abstract business logic away from our components and encapsulate implementation into our services

Overview

Currently our application uses data declared in the components themselves. This isn't very effective as we are already repeating ourselves, it becomes difficult to change the implementation, and it is difficult to test. In this exercise, we are going to abstract the implementation to a service and then provide that service to the components that require it. The services will use JSON files as the end point of HTTP requests, provided by a neat npm package called json-server.

Instructions

1. Start this exercise by adding the import for `HttpClientModule` from `@angular/common/http` to **app.module.ts**
2. Create a **Course Service** within the **Courses** module.

   - You can do this manually or by using the *CLI generator* using: `ng g service courses` from within the courses folder, just be sure to inspect the resultant code when using the generator to ensure it is as expected

3. Import `HttpClient` from `@angular/common/http`
4. Set a `private readonly` **string** called `COURSESURL` to be:

```
http://localhost:3000/coursesResults
```

5. Inject a **private** `HttpClient` called `http` into the **constructor** of the service
6. Add a `getCourses()` function to the **CourseService** that:

   - Takes no arguments
   - Returns the result of a call to the **HttpClient** `get` method, using the URL specified as `COURSEURL` as the argument

Let's now use this service in the course-editor.

7. Inject the **service** into the **constructor** of the **course-editor** component - do this by adding it with an appropriate access parameter.
8. Remove the array from the `courses` property, but leave the declaration as an array of `Courses` in place.

9. In the `ngOnInit` method write a method body that:

   - **Subscribes** to the Course Service's `getCourses` method with 2 arguments
   - The first argument should be an anonymous function that takes the `response` of an argument and sets `courses` to the `response` as an *array of Courses*
   - The second argument should be an anonymous function that takes an `error`, in the form of an `HttpErrorResponse`
   - The *second argument's body* should check for an `instanceof Error` and alert the *error message* or alert the *error status* and *a message explain the data was not saved*

10. Import the `HttpErrorResponse` from `@angular/common/http` into the component

Use the service in the App Component

11. You need to refactor **app.component.ts** so that it makes use of this new service – it will need to set the `displayCourses` to the new value of `courses` once retrieved

- HINT: Essentially use similar code to that used in the Course Editor Component to inject the service into the component and retrieve the courses.

Checking the Application so far…

12. Before running your application, open a command-line/terminal and enter the following command:

```
npm run json-server
```

This will launch the json-server and give you access to the following resources:

```
Resources
http://localhost:3000/courseResults
http://localhost:3000/instructorResults
http://localhost:3000/locationsResults
```

json-server allows CRUD commands and documentation can be found by visiting: https://www.npmjs.com/package/json-server

13. Using a **new instance** of the command-line/terminal to run the `ng serve` command, test your application - the course-editor should continue to work as it did before

Send the form data back via the service

Once a user makes changes, it is usual that the data is sent back to the origin to update it. This part will demonstrate how to send the data. The idea is that you can provide data from your application for the backend service to consume.

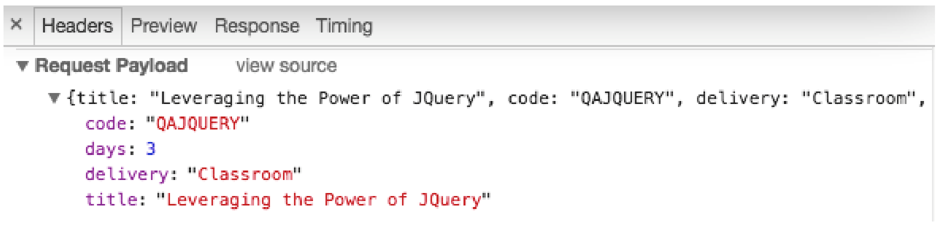
14. In the **Course Service**, add a method called `submitCourse()` that takes a `course` as an argument
15. Make the body of the method **return** a `PUT` request to: `COURSEURL` with a *sub path* of the **course's id** and a **payload** of the `course` supplied
16. In the **Course Editor Component**, edit the `onSubmit()` method replacing the code with:

```
this.courseService.submitCourse(this.selectedCourse).subscribe();
```

17. In the `subscribe` function make the `response` `alert` that the *data was saved*
18. As a second argument to the `subscribe` function, if the `error` is an `instanceof` `Error`, `alert` that *an error occurred* with *the error message*. Elsewise, `alert` that the *backend sent an error status with its valu*e and that the *data was not saved*.

Save all of your files and check that the application is functioning. Go through the motions to submit changes to a course. We can check that the PUT request delivered the correct payload by observing the alert message and then checking the output on the form.

Our update has been accepted and the current version of the courses array on json-server has been updated. We can check what was sent to the server by locating the request in the Network tab of the Developer tools and scrolling down to find the Request Payload:



```
▼ Request Payload
  ▼ {title: "Leveraging the Power of JQuery", code: "QAJQUERY", delivery: "Classroom",
      code: "QAJQUERY"
      days: 3
      delivery: "Classroom"
      title: "Leveraging the Power of JQuery"
```

If you have time

19. Our **instructor-gallery** is also guilty of embedding data into the component. Extract this data into its own **InstructorService** complete with `getInstructors()` method that returns the array of instructors as our `getCourses` method does and then use that in the component in place of the array.

The URL for the instructors is:

```
http://localhost:300/instructorResults
```