Exercise 11 - Routing our application

Objective

To use the Angular Router to add multiple routes to our application.

Overview

Using Angular best practices, we will start by creating a simple routed application with two possible routes. We will use child routes to create a more modular design and then we will use route parameters and the `ActivatedRoute` module to add finer grain control over the behaviour of our routes.

Instructions

Part 1 - Adding routes to our application

In this first part we are going to create a routing module and add some initial routes to our application.

1. Before we go any further we need to refactor the Instructor module slightly to benefit from the routing we are going to add in. Create a component for the instructors module (instructors.component.[ts,html,css]) in the instructors folder (be sure to use the **--flat** flag here from the instructors folder too to avoid creating a sub-folder called instructors if you are using the CLI)
2. This instructors.component is going take over from app.component as the parent component of the instructors gallery:
    a. **Copy** the `ngOnInit` function and the `courses` and `displayCourses` properties from **app.component.ts** into **instructors.component.ts**
    b. **Move** the `doInstructorChange` function from **app.component.ts** into **instructors.component.ts**
    c. **Move** the `app-instructor-gallery` element from **app.component.html** into **instructors.component.html**
3. Now for the routing. Create an **AppRouting** module at the root of your application. If you use the CLI generator for this you can use the **--flat** flag to avoid creating a new sub-directory.
4. Within our new **app-routing.module** we are going to configure the `Router`.
    a. Import the class definitions for the components we will use: *instructors* and *course* editor.
    b. Define two `routes` in a `Routes` array:

```
let routes: Routes = [
    {path: 'instructors', component: InstructorsComponent},
    {path: 'editcourses', component: CourseEditorComponent}
]
```

5. We now need to add the `RouterModule`
    a. Add an *ES2015 import* for the `RouterModule` and `Routes` from `@angular/router`
    b. Within the *imports array* add a call to the `forRoot` method of the `RouterModule`, passing in the `routes` array from above
6. This has configured our `RouterModule`, so *re-export it from this module*
7. Import the `AppRouting` module you created above *into the root module*
8. We now need to update the view.
    a. Remove the references in **app.component.html** to the *course-editor* and *instructor-gallery* components
    b. Add a `router-outlet` element in their place
9. Test the application so far by manually visiting the /editcourses and /instructors routes. You should see the "Courses Available" grid at the top, and then the appropriate component underneath, depending on the route you visited.

- *Don't forget to do an* `npm run json-server` *from any folder within qa-app in a separate command line/terminal to enable to HTTP requests to work*

Part 2 - Adding child routes

Now we have a basic router in place, let's explore the child route functionality. This will enable us to control navigation for each feature module from within that module, loosening the coupling with the root module.

10. Using the skills you've learned so far, *create a new component* in the *courses module* called `course-browse` - this is going to be responsible for the section we see in the view headed "*Courses available at QA*"
11. *Move code from all files* that creates the "*Courses available at QA*" section from the *root module* into this new **course-browse** module.

- You should have moved code from:
    - **app.component.ts** (don't forget about imports)
    - **app.component.html**
    - **app.component.css**
- With regards to the template file: be sure to only remove the code that pertains specifically to this section. The header, footer and router-outlet should remain behind
- In the `ngOnInit` function, remove any reference to `displayCourses` and add the import for the `HttpErrorResponse`

12. Add *a new route* to the `Router` to display this component if we visit /courses

13. Check the application. We should have a largely blank page (except the header and footer) if no route is visited, and then 3 individual components viewable by visiting their routes /instructors, /editcourses or /courses

This is fine but our courses module's routes are now entangled with our root module's routes. This is going to get difficult to maintain as the application grows - so let's separate them out now.

14. Set up a child router for the courses module, with routes for the browse and editor components, these should be /courses/browse and /courses/edit
15. We can take another step to avoid such repetition and improve the structure of our routed application by having a child route with its own outlet.
    a. Create a **courses** component for the **courses module** which has a **router-outlet** of its own
    b. Update the `routes` array in the **courses-routing module** so that it has *one path for* `'courses'` and a `'children'` array with the paths `'browse'` and `'edit'`

```
let routes: Routes = [
    {
        path: 'courses', component: CoursesComponent,
        children: [
            {path: 'edit', component: CourseEditorComponent },
            {path: 'browse', component: CourseBrowseComponent }
        ]
    }
]
```

- Don’t forget to import your Routing Module for courses into the Course Module (and export it again!)

16. Modify the **app-routes.module** so that there is *no trace of any course-type route*

Part 3 - Navigating our application

In this part of the exercise you will use the `routerLink` directive to set up some navigation in the templates, then you will set up some programmatic navigation using the `Router` object and then finally you will retrieve the data passed from the `ActivatedRoute` object to populate the *edit-course* form.

17. Use the `routerLink` directive to hook up the navigation bar so that the **COURSES** link points to the route for **course-browse** and **INSTRUCTORS** links point to the **instructor route**.

18. Currently the **course-editor** component is only reachable through manual navigation. Let's hook up the browse component so that each course can be clicked on and will lead us to the editor component with the correct course pre-selected.
    a. Within the **course-browse** template add *a new list item with a button*
    b. Add a *click event handler* to call a function `navToEdit` and pass the course code as an argument

```
(click)="navToEdit(course.code)"
```

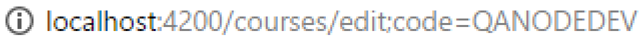
19. In the component class add the aforementioned `navToEdit` function, to handle the navigation you need to:
    a. Import the `Router` object and *inject it into the constructor*
    b. Call the router's `navigate` function to *navigate to the edit component*, passing the *course code* as a parameter

- In the previous step, you could have used the `relativeTo` option of the navigate function to make your code less brittle to changes down the line. If you didn't use it - try to refactor your code to include it now.

20. Add a **path** to the **children** property of the **course-routing.module** – it should add a **code parameter** to the end of **edit**

```
{path: "edit/:code", component: CourseEditorComponent}
```

21. Test your code in the browser, you should find that you can navigate to the course edit page and the URL contains the optional parameter we were after:


localhost:4200/courses/edit;code=QANODEDEV

Making the component look for the parameter

To achieve this goal, the **CourseEditorComponent** must retrieve and display the correct course, based on the code of the course that was clicked on in the previous page (or if the select input on this page is changed). As this requires subscriptions to 2 Observables (the return of `getCourses()` and the return of a call to the parameter map of the URL) that would be nested. A different approach needs to be used as this is considered bad practice (like the old callback *'Pyramid of Doom'*). Time for Observable operators to step-up, namely `flatMap`.

22. In **course-editor.component.ts**, insert a `pipe` before the call to `subscription`
23. Inside the `pipe` call, call `flatMap` with *an anonymous function argument* that:

    ▪ Takes `response` as an argument
    ▪ Sets `this.courses` to be the *array of courses* from the `response`
    ▪ Returns `this.activatedRoute.params`

24. Change the subscription so that the part before the error handling:

    - Takes `params` of type `Params` as an argument to the anonymous function
    - Sets `this.selectedCourse` to be the course found by looking for the course with the `code` supplied by `params` in the `courses` array

```
this.selectedCourse = this.course.find(course=>
   course.code === params['code']);
```

25. Add imports for:

    - `flatMap` from **rxjs/operators**
    - `ActivatedRoute` and `Params` from **@angular/router**

26. Inject the `ActivatedRoute` into the **constructor** as `activatedRoute`
27. Now visit the COURSES link from within your application, and select the EDIT button on any one of the listed courses. You should find that it navigates to the edit component, and once the data has been returned from the **courseService** the appropriate course will populate the form.

If you have time

28. Use named outlets to show the course-edit component in the same view as the course-browse component