ISEL / ADEETC

Master in Communication and Multimedia Network Engineering

Interactive Multimedia Applications

Lab Work 4

Interactive Multimedia Applications

Favourite Movies

(Angular JS)

Rui Jesus

Introduction

This work aims to introduce the Angular JS framework which is a relevant part of the Ionic framework. The tutorial begins with the download and installation of the necessary tools. The following is to build a responsive website using the main features of the Angular JS framework.

The development of the Favorite Movies app covers the fundamentals of Angular JS framework. In this tutorial students will build an app where the user can make a list of his most favorite movies and watch their trailer. Students can choose another topic, for instance cars, cities or animals. However, the structure of the app should not be changed, only the content related to the topic should be changed.

This app has many of the features students would expect to find in a data-driven application. It acquires and displays a list of movies, edits a selected details of a movie, and navigates among different views of movies data.

The following parts of the Angular JS are used in the application:

(1) Angular **directives** to show and hide elements and display lists of movies data;

(2) Create Angular **components** to display movies details and show an array of movies;

(3) Use **one-way data binding** for read-only data;

(4) Add editable fields to update a model with **two-way data binding**;

(5) Bind component methods to user **events**, like keystrokes and clicks;

(6) Format data with **pipes**;

(7) Create a shared **service** to assemble the movies;

(8) Use **routing** to navigate among different views and their components;

(9) Use the **dependency injection system** for components communication;

(10) Add **observable data** for asynchronous operations;

(11) Add a **HttpClient** (Angular) for communicating with a remote server over HTTP.

The final application will look like the following:

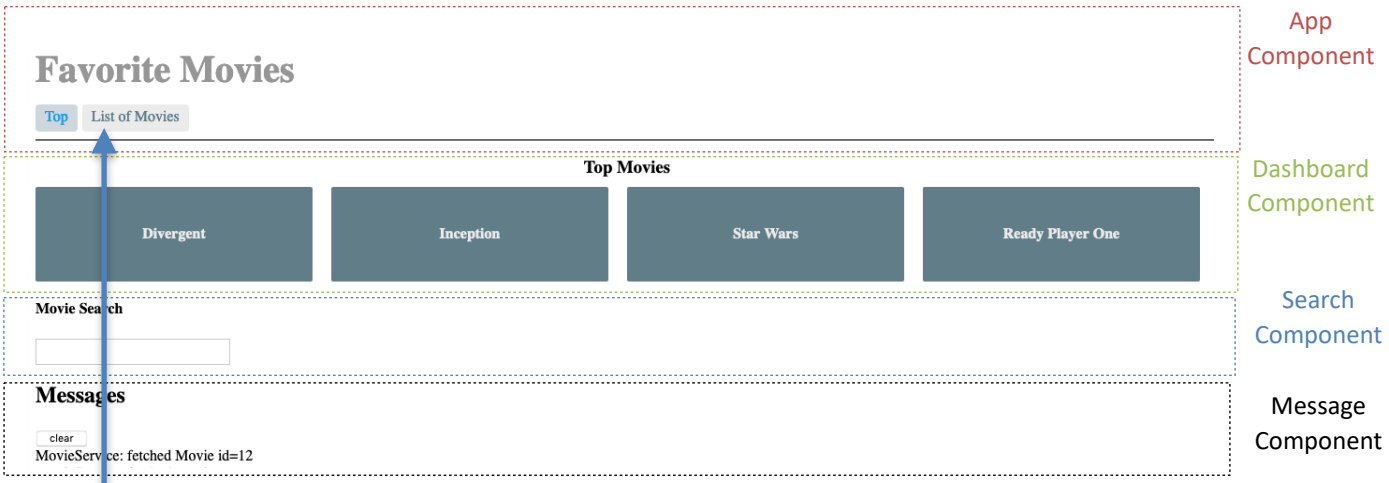

Figure 1. Main Layout composed by 4 Angular Components.

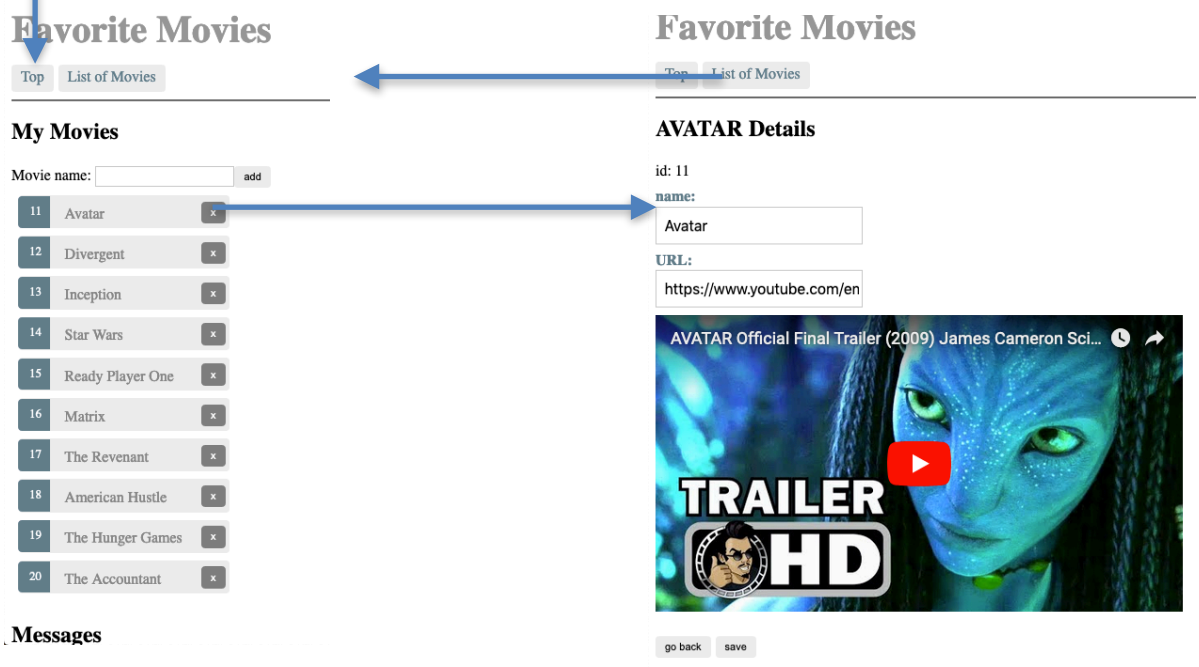

Figure 2. My Movies and Movie Details layouts.

Users can click buttons "Top" and "List of Movies" to navigate between the Main layout and My Movies layout. One click on a movie (List or Top) presents the user the Movie Details layout. The "go back" button on the Movie Details layout takes the user to the Main layout.

Note: this lab work should be done in class and the resulting code must be delivered through the Moodle platform until **May 16th**.

Laboratory Work

Download and installation of the tools

1. Angular requires **"Node.js"** and **"npm"** (Node.js package manager) package manager. Make sure your development environment includes these tools. If not, install them from https://nodejs.org. "Node.js" installation includes the "npm".

2. Angular requires **Node.js** version 8.x or 10.x. To check the versions in a terminal/ console window run:

   ```
   node -v
   npm -v
   ```

3. To update (if necessary) the **npm** version, run in a terminal/console window:

   ```
   npm install npm@latest -g
   ```

4. It is recommended that developers use the **Angular CLI tool** to accelerate development and adhering to the style recommendations that benefit every Angular projet. Therefore, install the Angular CLI to create projects. In terminal/console terminal enter the following command:

   ```
   npm install -g @angular/cli
   ```

5. Angular bibliography recommends the use of the **Visual Studio Code** editor because it has the right properties for JavaScript/TypeScript editing and includes a terminal/ console window. It is suggested the use of this tool (you are free to use another one). Install it from the link https://code.visualstudio.com/.

First Angular app

6. The following setup will create Angular workspace an initial application. A workspace contains the files for one or more projects. A project is the set of files that comprise an app, a library, or end-to-end (e2e) tests. Run the CLI command **"ng new"** and provide the name my-app:

   ng new my-app

   Note: information about features to include in the initial app project will be asked. Accept the defaults by pressing the Enter or Return key. The **Angular CLI** installs the necessary **Angular npm packages** and other **dependencies**. This can take a **few minutes**.

   The CLI command "**ng new my-app**" creates the following workspace and starter project files:

   1. A new workspace, with a root folder named **my-app**;
   2. An initial skeleton app project, also called **my-app** (in the src subfolder);
   3. An end-to-end test project (in the e2e subfolder);
   4. Related configuration files;
   5. The initial app project contains a simple Welcome app, ready to run.

7. Angular includes a server to easily build and serve an app locally. In the terminal/console window go to the "my-app" folder and run the app:

   ng serve --open

   This command launches the server, watches your files, and rebuilds the app as you make changes to those files. The --open (or just -o) option automatically opens your browser to http://localhost:4200/.

Editing the Angular App

8. The page you in the browser is controlled by an Angular Component named AppComponent. **Components** are the fundamental building blocks of Angular applications. They display data on the screen, listen for user input, and take action based on that input. Click on the previous link or https://angular.io/guide/architecture to know more about the Angular **Components.**

9. Open the project folder in the **Visual Studio Code** and check the project file structure which is composed by 4 main parts:
    a. An initial skeleton app project, also called **project_name** (in the **src**/ subfolder).
    b. a folder with the node modules called **node_modules;**
    c. An end-to-end test project (in the **e2e**/ subfolder).
    d. Related configuration files.

At this stage, the **src/** subfolder is the most relevant part. To know more about the other parts follow the link https://angular.io/guide/file-structure. The **src/** subfolder

app/	Contains the component files in which your app logic and data are defined. **More details below**.
assets/	Contains image files and other asset files to be copied as-is when you build your application
index.html	The main HTML page that is served when someone visits your site. The CLI automatically adds all JavaScript and CSS files when building your app, so you **typically don't need to add any <script> or<link> tags here manually.**
main.ts	**The main entry point for your app**. Compiles the application with the JIT compiler and bootstraps the application's root module (AppModule) to run in the browser.
test.ts	The main entry point for your unit tests, with some Angular-specific configuration. **You don't typically need to edit this file.**

contains the source files (app logic, data, and assets), along with configuration files for the initial app. The main parts are described in the following table (Table 1):

Table 1. Main parts of the **src/** subfolder.

The **app/** subfolder as you should check is composed by the following files:

1. **"app.component.ts"** - defines the logic for the app's root component, named **AppComponent**. The **view** associated with this root component becomes the root of the **view** hierarchy as you add **components** and **services** to your app;

2. **"app.component.html"** - defines the HTML template associated with the root **AppComponent;**

3. **"app.component.css" -** defines the base CSS stylesheet for the root **AppComponent;**

4. **“app.component.spec.ts”** - defines a unit test for the root **AppComponent;**

5. **“app.module.ts” -** defines the **root module**, named **AppModule**, that tells Angular how to **assemble the application**. Initially declares only the AppComponent. As you **add more components to the app, they must be declared here**.

10. To finish this initial part, change the title of the app to “Favorite Movies”, the logo image to an image related to movies and the first link from “Tour of Heroes Tutorial” to “Angular Fundamentals” to jump to the page https://angular.io/guide/architecture. Run the app in the browser.

11. Add style below to the application. Run the app in the browser.

```
/* Application-wide Styles */

    h1 {
        color: #369;
        font-family: Arial, Helvetica, sans-serif;
        font-size: 250%;
    }

    h2, h3 {
        color: #444;
        font-family: Arial, Helvetica, sans-serif;
        font-weight: lighter;
    }

    body {
        margin: 2em;
    }

    body, input[type="text"], button {
        color: #888;
        font-family: Cambria, Georgia;
    }

    * {
        font-family: Arial, Helvetica, sans-serif;
    }
```

Movies Components

12. Create a new "Component" called **Movies** to display the movie information in the application shell (name given to the page controlled by the AppComponent). Using the Angular CLI, generate a new component named **Movies:**

    ng generate component movies

    A new folder is created in the **src/app/** subfolder called **Movies** with the same type of files as the AppComponent. Note the following in the MoviesComponent class file ("movies.component.ts"):

    1. Always import the Component symbol from the Angular core library and annotate the component class with @Component;

    2. @Component is a decorator function that specifies the Angular metadata for the component:
        a. selector - the component's CSS element selector;
        b. templateUrl - the location of the component's template file;
        c. styleUrls - the location of the component's private CSS styles.

    3. The **ngOnInit** is a lifecycle hook. Angular calls **ngOnInit** shortly after creating a component. It's a good place to put initialization logic;

    4. Always export the component class so you can import it elsewhere (like in the AppModule).

13. Add the "movie_name" property to the MoviesComponent for a movie named "Avatar" (before the Constructor):

    ```
    movie_name:string ="Avatar";
    ```

14. Show the name of the movie. Open the "movies.component.html" file. Delete the default text generated by the Angular CLI and replace it with a **data binding** to the new **movie_name** property:

    {{movie_name}}

15. To display the MoviesComponent, it must be added to be html template of the AppComponent. Replace all the code in the "app.component.html" with the following:

```
<h1>{{title}}</h1>
<app-movies></app-movies>
```

Note: Remember that app-movies is the element selector for the MoviesComponent.

Run the app in the browser. Assuming that the CLI "**ng serve"** command is still running, the browser should refresh and display both the application title and the movie name, after the changes are saved.

16. A movie is more than a name. Therefore, create a Movie class in its own file in the scr/ app folder:

```
export class Movie {
      id: number;
      name: string;
      url: string;
}
```

17. In the file "movies.component.ts" replace the movie_name property by a new property called "movie" which is an instance of the classe "**Movie**". Declare and initialize the property (before the **constructor**).

18. Show the properties "id" and "name", change the "movie.component.html" file to:

```
<h2>{{movie.name} | uppercase} Details</h2>
<div><span>id: </span>{{movie.id}}</div>
<div><span>name: </span>{{movie.name}}</div>
```

19. Users should be able to edit the movie name in an **<input>** textbox. The textbox should both display the movie name property and update that property as the user types. That means data flow from the **component class** out to the **screen** and from the **screen** back to the **class**.
    To automate that data flow, create a **two-way data binding** between the **<input>** form element and the **movie.name property:**

```
<h2>{{movie.name | uppercase}} Details</h2>
<div><span>id: </span>{{movie.id}}</div>
<div>
      <label>name:
      <input [(ngModel)]="movie.name" placeholder="name"/>
       </label>
```

```
</div>
```

Note: **[(ngModel)]** is Angular's two-way data binding syntax. Here it binds the **`movie.name`** property to the HTML textbox so that data can flow *in both directions:* from the **`movie.name`** property to the textbox, and from the textbox back to the **`movie.name`**.

20. Although **ngModel** is a valid Angular directive, it isn't available by default. It belongs to the optional **FormsModule** and you must opt-in to using it. The Angular CLI generated an **AppModule** class in "src/app/app.module.ts" when it created the project. This is where you opt-in to the **FormsModule**. Run the code after including the following:

```
import { FormsModule } from '@angular/forms';

imports: [
      BrowserModule,
      FormsModule
],
```

Note: every component must be declared in exactly one NgModule. You didn't declare the MoviesComponent. Why did the application work? It worked because the Angular CLI declared HeroesComponent in the AppModule when it generated that component (check it in the "app.module.ts" file).

Display a List of Movies

21. To expand the Movies app to display a list of movies, and allow users to select a movie and display the movie's details, it is needed to create a database of movies. Create a file called "**movies.ts**" in the **src/app/** folder to simulate the database. Define a MOVIES constant as an array of ten movies and export it. The file should look like this:

```
import { Movie } from './movie';

export const MOVIES: Movie[] = [
{ id: 11, name: 'Avatar', URL:'https://www.youtube.com/embed/
6ziBFh3V1aM'},
{ id: 12, name: 'Divergent',URL:'https://www.youtube.com/embed/
sutgWjz10sM'},
    { id: 13, name: 'Inception', URL:'https://www.youtube.com/embed/
YoHD9XEInc0'},
```

```
        { id: 14, name: 'Star Wars', URL:'"https://www.youtube.com/embed/
FDXmcZ1_D-o'},
        { id: 15, name: 'Ready Player One', URL:'https://www.youtube.com/
embed/cSp1dM2Vj48'},
        { id: 16, name: 'Matrix', URL:'https://www.youtube.com/embed/
vKQi3bBA1y8'},
        { id: 17, name: 'The Revenant',URL:'https://www.youtube.com/embed/
LoebZZ8K5N0'},
        { id: 18, name: 'American Hustle', URL:'https://www.youtube.com/
embed/ST7a1aK_lG0'},
       { id: 19, name: 'The Hunger Games', URL:'https://www.youtube.com/
embed/mfmrPu43DF8'},
     { id: 20, name: 'The Accountant', URL:'https://www.youtube.com/embed/
DBfsgcswlYQ' }
    ];
```

22. Open the **MoviesComponent** class file and import the movies database ("movies.ts"). In the same file, define a component property called movies to expose MOVIES array for binding. Remove the property, the instance of Movie, added in point 17.

23. To list all the movies, open the **MoviesComponent** template (html file) and include the following at the beginning of the file:

```
<h2>My Movies</h2>
<ul class="movies">
      <li>
            <span class="badge">{{movie.id}}</span>
            {{movie.name}}
      </li>
</ul>
```

24. Now change the **<li>** to this:

```
<li *ngFor="let movie of movies">
```

Note: The ***ngFor** is Angular's repeater directive. It repeats the host element for each element in a list. Don't forget the asterisk (*) in front of ngFor. It's a critical part of the syntax.

Run the code in the browser.

25. The movies list should be attractive and should respond visually when **mouse is over** and the **user selects a movie** from the list. Add a click event binding to the **<li>** like this:

```
<li *ngFor="let movie of movies" (click)="onSelect(movie)">
```

The parentheses around click tell Angular to listen for the **<li>** element's click event. When the user clicks in the **<li>**, Angular executes the **onSelect(movie)** expression. **onSelect()** is a **MoviesComponent** method. Angular calls it with the **movie** object displayed in the clicked **<li>**, the same **movie** defined previously in the ***ngFor** expression.

26. Add the following **onSelect()** method, which assigns the clicked **movie** from the template to the component's **selectedMovie:**

```
selectedMovie: Movie;
onSelect(movie: Movie): void {
      this.selectedMovie = movie;
}
```

27. Rename the **movie** to **selectedMovie** in the component template (html file). Run the code in the browser. What happened?

When the app starts, the **selectedMovie** is undefined by design.
Binding expressions in the template that refer to properties of selectedMovie — expressions like {{**selectedMovie.name**}} - must fail because there is no selected **movie**.
Now, click one of the list items. The app seems to be working again. The **movies** appear in a list and details about the clicked **movie** appear at the bottom of the page.

28. The component should only display the selected **movie** details if the **selectedMovie** exists. Wrap the **movie** detail HTML in a ** **. Add Angular's ***ngIf** directive to the ** ** and set it to selectedMovie.

```
<div *ngIf="selectedMovie">
      <h2>{{selectedMovie.name | uppercase}} Details</h2>
      <div><span>id: </span>{{selectedMovie.id}}</div>
      <div>
            <label>name:
```

```
                    <input [(ngModel)]="selectedMovie.name"
                    placeholder="name"/>
                </label>
            </div>
        </div>
```

When **selectedMovie** is undefined, the ***ngIf** removes the **movie** detail from the DOM. There are no **selectedMovie** bindings to worry about.

When the user picks a **movie**, **selectedMovie** has a value and **ngIf** puts the movie detail into the DOM.

29. It is difficult to identify the selected **movie** in the list when all **<li>** elements look alike. The Angular class binding makes it easy to add and remove a CSS class conditionally. Just add **[class.some-css-class]="some-condition"** to the element you want to style. Add the following **[class.selected]** binding to the **<li>** in the **MoviesComponent** template (html file):

```
<li *ngFor="let movie of movies"
        [class.selected]="movie === selectedMovies"
         (click)="onSelect(movie)">
      <span class="badge">{{movie.id}}</span> {{movie.name}}
</li>
```

30. Add the following css style to the "**movies.component.css**" file:

```
.selected {
      background-color: #CFD8DC !important;
      color: white;
}
.movies {
      margin: 0 0 2em 0;
      list-style-type: none;
      padding: 0;
      width: 15em;
}
.movies li {
      cursor: pointer;
      position: relative;
      left: 0;
      background-color: #EEE;
      margin: .5em;
      padding: .3em 0;
```

```
        height: 1.6em;
        border-radius: 4px;
}
.movies li.selected:hover {
        background-color: #BBD8DC !important;
        color: white;
}
.movies li:hover {
        color: #607D8B;
        background-color: #DDD;
        left: .1em;
}
.movies .text {
        position: relative;
        top: -3px;
}
.movies .badge {
        display: inline-block;
        font-size: small;
        color: white;
        padding: 0.8em 0.7em 0 0.7em;
        background-color: #607D8B;
        line-height: 1em;
        position: relative;
        left: -1px;
        top: -4px;
        height: 1.8em;
        margin-right: .8em;
        border-radius: 4px 0 0 4px;
}
```

Run the code in the browser.

Movie Detail Component

31. At the moment, the **MoviesComponent** displays both the list of **movies** and the selected **movies's** details. Keeping all features in one component as the application grows will not be maintainable. Large **Components** should be divided into smaller sub-**components**, each focused on a specific task or workflow. Use the Angular CLI to generate new **Component** named **"movie-detail"** (repeat point 12).

32. Cut the HTML for the movie detail from the bottom of the **MoviesComponent** template and paste it over the generated boilerplate in the **Movie-DetailComponent** template.

33. The "**movie-detail**" component only receives a movie object (the selected one from the "**movies**" component) through its movie property and displays it. In the HTML, replace the "selectedMovie" by "movie" (property of the component "**movie-detail**"). This new **Component** can present any movie, not just a selected movie.

34. Add the "movie" property to the "**movie-detail**" component. Do not forget to import the "**Movie**" class. The movie property must be an Input property, annotated with the **@Input() decorator**, because the external **MoviesComponent** will bind to it like this. The **"movie-detail"** component will receive from an external component the **selected movie.** Include the following before the **constructor:**

```
@Input() movie: Movie;
```

35. The **MoviesComponent** is still a master/detail view. It used to display the **movie** details on its own, before you cut that portion of the template. Now it will delegate to the **MovieDetailComponent.** The two **components** will have a **parent/child** relationship. The **parent MoviesComponent** will control the **child MovieDetailComponent** by sending it a new **movie** to display whenever the user **selects** a **movie** from the list.

    The **MovieDetailComponent** selector is "**app-movie-detail**". Add an **`<app-movie-detail>`** element near the bottom of the **MovieComponent** template, where the **movie** detail view used to be. Bind the **MoviesComponent.selectedMovie** to the element's **movie** property like this:

```
<app-movie-detail [movie]="selectedMovie"></app-movie-detail>
```

    `[movie]="selectedMovie"` is an **Angular property binding**. It is a one way **data binding** from the **selectedMovie** property of the **MoviesComponent** to the **movie** property of the target element, which maps to the **movie** property of the MovieDetailComponent.

MovieService (Angular Dependency Injection)

36. The **MoviesComponent** is currently getting and displaying data. Components shouldn't fetch or save data directly. They should focus on presenting data (supporting the view) and delegate data access to a service. Therefore, we are going to create a **MovieService** that all application classes can use to get **movies**. Instead of creating a **service** with the **new** operator, we are going to use the Angular dependency injection system to inject it into the **MoviesComponent** constructor.
    Using the Angular CLI create a new **MovieService** class in the **src/app/movies** folder with this command:

        ng generate service movies/movie

    In the "**movie.service.ts**" file the **@Injectable()** is an essential ingredient in every Angular service definition. This marks the class as one that participates in the dependency injection system. The **MovieService** class is going to provide an injectable service, and it can also have its own injected dependencies.

37. Create the **getMovies()** method in the class **MovieService**:

```
getMovies(): Movie[] { return MOVIES; }
```

38. After create the **MovieService**, it is needed to make the **MovieService** available to the dependency injection system before Angular can inject it into the **MovieComponent**. This done by registering a **provider**. A **provider** is something that can create or deliver a service; in this case, it instantiates the **MovieService** class to provide the service.

    By default, the Angular CLI command "ng generate service" registers a **provider** with the **root** injector for your service by including provider metadata in the **@Injectable** decorator.

    When you provide the service at the **root** level, Angular creates a single, shared instance of **MovieService** and injects into any class that asks for it. The next step is the injection of service into **MoviesComponent.** Add a private **movieService** parameter of type **MovieService** to the constructor:

```
constructor(private movieService: movieService) { }
```

39. To have access to the **MOVIES**, change the code to use the **movieService** property:

```
movies: Movie[]; // instead of instance of the MOVIES property


getMovies(): void {
        this.movies = this.movieService.getMovies();
}
```

40. Instead of call "getMovies()" in the constructor, that's not the best practice, call it inside the ngOnInit lifecycle hook and let Angular call **ngOnInit** at an appropriate time after constructing a **MoviesComponent** instance.

    Reserve the constructor for simple initialization such as wiring constructor parameters to properties. The constructor shouldn't do anything. It certainly shouldn't call a function that could make HTTP requests to a remote server as a real data service would. Run the app in the browser.

MOVIES - Observable Data

41. The "MovieService.getMovies()" method has a synchronous signature, which implies that the **MovieService** can fetch movies synchronously. But soon the app will fetch **movies** from a remote server, which is an inherently asynchronous operation. The **MovieService** must wait for the server to respond, "getMovies()" cannot return immediately with movie data, and the browser will not block while the service waits. "MovieService.getMovies()" must have an asynchronous signature of some kind.

    It can take a **callback**. It could return a **Promise**. It could return an **Observable**.
    In this tutorial, "MovieService.getMovies()" will return an **Observable** in part because it will eventually use the **Angular HttpClient.get** method to fetch the **movies** and **HttpClient.get()** returns an **Observable**.
    Observable is one of the key classes in the RxJS library. Open the **MovieService** file and import the **Observable** and **of symbols** from RxJS:

```
import { Observable, of } from 'rxjs';
```

    Change the "getMovies()" method to return an **Observable**:

```
getMovies(): Observable<Movie[]> {
      return of(MOVIES);
}
```

42. Adjust the changes in **MoviesComponent**:

```
getMovies(): void {
      this.movieService.getMovies()
            .subscribe(movies => this.movies = movies);
}
```

This new version waits for the **Observable** to emit the array of **movies -** which could happen now or several minutes from now. Then **subscribe** passes the emitted array to the callback, which sets the component's **movies** property. This asynchronous approach will work when the **MovieService** requests **movies** from the server.

Messages Component

43. To display app messages at the bottom of the screen, create the **MessagesComponent**.

44. Modify the **AppComponent** template (html file) to display the generated **MessagesComponent**. Run the app in the browser. The app should display at the bottom "message works".

45. **Components** should only present data in the view. Use the CLI to create the **MessageService** in src/app.

46. Open **MessageService** and add to its contents the following properties and methods:

```
messages: string[] = [];

 add(message: string) {
      this.messages.push(message);
 }

clear() {
      this.messages = [];
}
```

The **service** exposes its cache of messages and two methods: one to "**add()**" a message to the cache and another to "**clear()**" the cache.

47. Inject the **MessageService** into the **MovieService** to display a message anytime data is delivered (repeat point 38). This is a typical "service-in-service" scenario: you inject the MessageService into the MovieService which is injected into the MovieComponent.

48. Modify the "getMovies()" method of the "**MovieService**" to send a message when the **movies** are fetched:

```
getMovies(): Observable <Movie[]> {
      this.messageService.add('MovieService: fetched movies');
      return of (MOVIES);
}
```

49. To display the message, open the **MessagesComponent** and import the **MessageService** to inject it in the constructor. Now, the **MessageService** instance injected must be public because the messages are going to be displayed in the template (html file) of the component.

50. Add the following code to the **MessagesComponent** template and run the app in the browser:

```
<div *ngIf="messageService.messages.length">
  <h2>Messages</h2>
  <button class="clear"
      (click)="messageService.clear()">clear</button>
  <div *ngFor='let message of messageService.messages'>
      {{message}}
  </div>
</div>
```

51. Add the private CSS styles to messages.component.css and run the app:

```
h2 {
      color: red;
      font-family: Arial, Helvetica, sans-serif;
      font-weight: lighter;
}
body {
      margin: 2em;
}
body, input[text], button {
```

```
        color: crimson;
        font-family: Cambria, Georgia;
}

button.clear {
        font-family: Arial;
        background-color: #eee;
        border: none;
        padding: 5px 10px;
        border-radius: 4px;
        cursor: pointer;
        cursor: hand;
}
button:hover {
        background-color: #cfd8dc;
}
button:disabled {
        background-color: #eee;
        color: #aaa;
        cursor: auto;
}
button.clear {
        color: #888;
        margin-bottom: 12px;
}
```

Routing Module

52. In Angular, the best practice is to load and configure the router in a separate, top-level module that is dedicated to routing and imported by the root **AppModule**. By convention, the module class name is **AppRoutingModule** and it belongs in the "app-routing.module.ts" in the **src/app** folder. Use the CLI to generate it:

    ng generate module app-routing --flat --module=app

    Note: --flat puts the file in src/app instead of its own folder.
    --module=app tells the CLI to register it in the imports array of the AppModule.

53. Generally we don't declare **components** in a **routing module,** therefore we can delete the **@NgModule.declarations** array and delete **CommonModule** references too.

Configure the router with **Routes** in the **RouterModule**, therefore, import those two symbols from the **@angular/router** library. Add an **@NgModule.exports** array with **RouterModule** in it. Exporting **RouterModule** makes **router** directives available for use in the **AppModule** components that will need them ("**app-routing.module.ts**" file):

```
import { NgModule }             from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

@NgModule({
      exports: [ RouterModule ]
})

export class AppRoutingModule {}
```

54. **Routes** tell the router which view to display when a user clicks a link or pastes a URL into the browser address bar. A typical Angular **Route** has two properties:
    a. **path**: a string that matches the URL in the browser address bar.
    b. **component**: the component that the router should create when navigating to this route.

    To navigate to the **MoviesComponent** the URL can be something like "localhost:4200/movies". Import the **MoviesComponent** in the **AppRoutingModule** to reference it in a **route**. Then, define an array of **routes** with a single **route** to that component:

```
import { NgModule }             from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import {MoviesComponent} from './movies/movies.component';

const routes: Routes = [
      { path: 'movies', component: MoviesComponent }
];

@NgModule({
       exports: [ RouterModule ]
})
export class AppRoutingModule {}
```

55. Initialize the **router** and start it listening for browser location changes. Add **RouterModule** to the **@NgModule.imports** array and configure it with the **routes** in one step by calling **RouterModule.forRoot()** within the imports array, like this:

```
imports: [ RouterModule.forRoot(routes) ],
```

56. Open the **AppComponent** template replace the **<app-movies>** element with a **<router-outlet>** element. Run the app in the browser.

    The element **<app-movies>** is removed because the **MoviesComponent** will only displays when the user navigates to it. The **<router-outlet>** tells the **router** where to display routed views.

    The **RouterOutlet** is one of the **router** directives that became available to the **AppComponent** because **AppModule** imports **AppRoutingModule** which exported **RouterModule**.

57. The browser should refresh and display the app title but not the list of movies. Look at the browser's address bar. The URL ends in /. The **route** path to **MoviesComponent** is **/movies**. Append **/movies** to the URL in the browser address bar.

58. Add a <nav> element and, within that, an anchor element that, when clicked, triggers navigation to the **MoviesComponent**. The revised **AppComponent** template looks like this:

```
<h1>{{title}}</h1>
<nav>
       <a routerLink="/movies">List of Movies</a>
</nav>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

Dashboard Component

59. Routing makes more sense when there are multiple views. Create a new **component** called DashboardComponent using CLI.

60. To look like as in Figure 1, add the following code to the component template:

```
<h3>Top Heroes</h3>
<div class="grid grid-pad">
      <a *ngFor="let hero of heroes" class="col-1-4">
            <div class="module hero">
                        <h4>{{hero.name}}</h4>
```

```
            </div>
      </a>
</div>
```

61. Add the following code to the DashboardComponent:

```
import { Component, OnInit } from '@angular/core';
import { Movie } from '../movie';
import { MovieService } from '../movies/movie.service';

@Component({
      selector: 'app-dashboard',
      templateUrl: './dashboard.component.html',
      styleUrls: [ './dashboard.component.css' ]
})
export class DashboardComponent implements OnInit {
      movies: Movie[] = [];

      constructor(private movieService: MovieService) { }

      ngOnInit() {
            this.getMovies();
      }

      getMovies(): void {
            this.movieService.getMovies()
                  .subscribe(movies => this.movies =
                  movies.slice(0, 4));
      }
}
```

62. Add the following code to the style file of the DashboardComponent:

```
[class*='col-'] {
      float: left;
      padding-right: 20px;
      padding-bottom: 20px;
}
[class*='col-']:last-of-type {
      padding-right: 0;
}
a {
      text-decoration: none;
```

```
}
*, *:after, *:before {
      -webkit-box-sizing: border-box;
      -moz-box-sizing: border-box;
      box-sizing: border-box;
}
h3 {
      text-align: center;
      margin-bottom: 0;
}
h4 {
      position: relative;
}
.grid {
      margin: 0;
}
.col-1-4 {
      width: 25%;
}
.module {
      padding: 20px;
      text-align: center;
      color: #eee;
      max-height: 120px;
      min-width: 120px;
      background-color: #607d8b;
      border-radius: 2px;
}
.module:hover {
      background-color: #eee;
      cursor: pointer;
      color: #607d8b;
}
.grid-pad {
      padding: 10px 0;
}
.grid-pad > [class*='col-']:last-of-type {
      padding-right: 20px;
}
@media (max-width: 600px) {
      .module {
            font-size: 10px;
```

```
                max-height: 75px; }
        }
        @media (max-width: 1024px) {
                .grid {
                        margin: 0;
                }
                .module {
                        min-width: 60px;
                }
        }
```

63. Add the dashboard route to the **AppRoutingModule.routes** array that matches a path to the **DashboardComponent** and run the code in the browser:

```
{ path: 'dashboard', component: DashboardComponent }
```

64. When the app starts, the browsers address bar points to the web site's root. That doesn't match any existing **route** so the router doesn't navigate anywhere. The space below the <router-outlet> is blank. Add a default **route** a run the code in the browser:

```
{ path: '', redirectTo: '/dashboard', pathMatch: 'full' }
```

65. The user should be able to navigate back and forth between the **DashboardComponent** and the **MoviesComponent** by clicking links in the navigation area near the top of the page. Add a dashboard navigation link to the **AppComponent** template, just above the **Movies** link and run the code in the browser:

```
<a routerLink="/dashboard">Top</a>
```

66. Change the style of the **AppComponent** to the following code:

```
h1 {
        font-size: 1.2em;
        color: #999;
        margin-bottom: 0;
}

h2 {
        font-size: 2em;
        margin-top: 0;
        padding-top: 0;
}
```

```
nav a {
      padding: 5px 10px;
      text-decoration: none;
      margin-top: 10px;
      margin-right: 5px;
      display: inline-block;
      background-color: #eee;
      border-radius: 4px;
}

nav a:visited, a:link {
            color: #607d8b;
}

nav a:hover {
      color: #039be5;
      background-color: #cfd8dc;
}

nav a.active {
      color: #039be5;
}
```

MovieDetail Component

67. The **MovieDetailsComponent** displays details of a selected **movie**. At the moment the **MovieDetailsComponent** is only visible at the bottom of the **MovieComponent.** The user should be able to get to these details in three ways:
    a. By clicking a movie in the **dashboard**.
    b. By clicking a movie in the **movies** list.
    c. By pasting a "deep link" URL into the browser address bar that identifies the movie to display.

    Delete movie details from **MovieComponent**.

68. An URL like **~/detail/11** would be a good URL for navigating to the Movie Detail view of the **movie** whose id is **11**. Add a parameterized **route** to the **AppRoutingModule.routes** array that matches the path pattern to the movie detail view:

```
{ path: 'detail/:id', component: MovieDetailComponent }
```

The colon (:) in the path indicates that :id is a placeholder for a specific **movie** id.

69. Change the movie links in the **DashboardComponent** template to:

```
<h3>Top Movies</h3>
<div class="grid grid-pad">
      <a *ngFor="let movie of movies" class="col-1-4"
      routerLink="/detail/{{movie.id}}">
            <div class="module movie">
                  <h4>{{movie.name}}</h4>
            </div>
      </a>
</div>
```

70. Change the movie links in the **MovieComponent** template to:

```
<h2> My Movies</h2>
<ul class="movies">
      <li *ngFor="let movie of movies">
                  <a routerLink="/detail/{{movie.id}}">
                        <span class="badge">{{movie.id}}</span>
                              {{movie.name}}
                  </a>
      </li>
</ul>
```

71. Change the style of the **MovieComponent** to the following code:

```
.movies {
      margin: 0 0 2em 0;
      list-style-type: none;
      padding: 0;
       width: 15em;
}

.movies li {
      position: relative;
      cursor: pointer;
      background-color: #EEE;
      margin: .5em;
      padding: .3em 0;
      height: 1.6em;
      border-radius: 4px;
```

```
}

.movies li:hover {
      color: #607D8B;
      background-color: #DDD;
      left: .1em;
}

.movies a {
      color: #888;
      text-decoration: none;
      position: relative;
      display: block;
      width: 250px;
}

.movies a:hover {
       color: #607D8B;
}

.movies .badge {
      display: inline-block;
      font-size: small;
      color: white;
      padding: 0.8em 0.7em 0 0.7em;
      background-color: #607D8B;
      line-height: 1em;
      position: relative;
      left: -1px;
      top: -4px;
      height: 1.8em;
      min-width: 16px;
      text-align: right;
      margin-right: .8em;
      border-radius: 4px 0 0 4px;
}
```

72. Remove the "**onSelect()**" method and **selectedMovie** property from the **MoviesComponent** class. They are no longer used. Run the code in the browse.

73. Previously, the parent **MovieComponent** set the **MovieDetailComponent.movie** property and the **MovieDetailComponent** displayed the **movie** details. **MoviesComponent** doesn't do that anymore. Now the router creates the **MovieDetailComponent** in response to a URL such as **~/detail/11**. The **MovieDetailComponent** needs a new way to obtain the **movie**-to-display. Get the **route** that created it, extract the **id** from the **route** and acquire the **movie** with that **id** from the server via the **MovieService.** First, made the following in the **MovieDetailComponent**:

```
import { MovieService } from '../movies/movie.service';

import { ActivatedRoute } from '@angular/router';
import { Location } from '@angular/common';

…
constructor(
      private route: ActivatedRoute,
      private movieService: MovieService,
      private location: Location
) {}
```

The **ActivatedRoute** holds information about the **route** to this instance of the **MovieDetailComponent**. This component is interested in the route's bag of parameters extracted from the URL. The "id" parameter is the id of the **movie** to display. The **MovieService** gets **movie** data from the remote server and this component will use it to get the **movie-to-display**.
The **location** is an Angular service for interacting with the browser. You'll use it later to navigate back to the view that navigated here.

74. Extract the id **route** parameter, following the code (**MovieDetailComponent**):

```
getMovie(): void {
      const id = +this.route.snapshot.paramMap.get('id');
       this.movieService.getMovie(id)
            .subscribe(movie => this.movie = movie);
 }
```

Do not forget to call the "**`getMovie()`**" method in the **ngOnInit()** lifecycle hook.

The **route.snapshot** is a static image of the **route** information shortly after the component was created.

The **paramMap** is a dictionary of **route** parameter values extracted from the URL. The "id" key returns the **id** of the **hero** to fetch.

Route parameters are always strings. The JavaScript (+) operator converts the string to a number, which is what a **movie** id should be.

The browser refreshes and the app crashes with a compiler error. **MovieService** doesn't have a "**getMovie()**" method.

75. Add the "**MovieService.getHero()**":

```
getMovie(id: number): Observable<Movie> {
      this.messageService.add(`MovieService: fetched movie id=$
      {id}`);
      return of(MOVIES.find(movie => movie.id === id));
}
```

Run the code in the browser.

76. By clicking the browser's back button, you can go back to the movie list or dashboard view, depending upon which sent you to the detail view. It would be nice to have a button on the MovieDetail view that can do that.

Add a "**go back**" button to the bottom of the component template and bind it to the component's "**goBack()**" method (html file):

```
<button (click)="goBack()">go back</button>
```

and add in the **MovieDetailComponent** the method:

```
goBack(): void {
      this.location.back();
}
```

Run the code.

77. Add the following style to the **MovieDetail** component to the following code:

```
label {
            display: inline-block;
             width: 3em;
            margin: .5em 0;
```

```
            color: #607D8B;
            font-weight: bold;
    }

     input {
            width: 400px;
            height: 2em;
            font-size: 1em;
            padding-left: .4em;
    }

    button {
            margin-top: 20px;
            font-family: Arial;
            background-color: #eee;
            border: none;
            padding: 5px 10px;
            border-radius: 4px;
            cursor: pointer;
            cursor: hand;
    }

    button:hover {
            background-color: #cfd8dc;
    }

    button:disabled {
            background-color: #eee;
            color: #ccc;
            cursor: auto;
    }
```

78. Add to the **MovieDetail** component template (html file) the remaining information about the movie:

```
    …
     <label>URL:
          <input [(ngModel)]="movie.URL" placeholder="URL"/>
    </label>
</div>
<div>
```

```
        <iframe
        [src]='sanitizer.bypassSecurityTrustResourceUrl(movie.URL)'
        width="560" height="315" frameborder="0" webkitallowfullscreen
        mozallowfullscreen allowfullscreen>
        </iframe>
    </div>
    …
```

Do not forget to import the **DomSanitizer** and to inject it in the constructor:

```
import { DomSanitizer } from '@angular/platform-browser';
…
constructor(public sanitizer: DomSanitizer, private route:
      ActivatedRoute, private movieService: MovieService,
      private location: Location)
{ }
```

79. Add a line (html element) after the <nav> element in the **AppComponent** template.

HTTP Services

80. In real apps the **movie** data, usually is obtained from a server data by **HTTP** requests. **HttpClient** is Angular's mechanism for communicating with a remote server over **HTTP**. To make **HttpClient** available everywhere in the app, open the root **AppModule,** import the **HttpClientModule** symbol from **@angular/common/http** and add it to the **@NgModule.imports** array.

81. To simulate communication with a remote data server, the **In-memory Web API** module is used. At this stage, is not so relevant to implement a real data server. After installing the module (**In-memory Web API**), the app will make requests to and receive responses from the **HttpClient** without knowing that the **In-memory Web API** is intercepting those requests, applying them to an in-memory data store, and returning simulated responses.

    Install the **In-memory Web API** package from **npm:**

    ```
    npm install angular-in-memory-web-api --save
    ```

82. Open the root **AppModul** and import the **HttpClientInMemoryWebApiModule** and the **InMemoryDataService** class, which you will create in a moment:

```
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-
web-api';
import { InMemoryDataService }  from './in-memory-data.service';
```

83. Add the **HttpClientInMemoryWebApiModule** to the **@NgModule.imports** array - after importing the **HttpClientModule**, - while configuring it with the **InMemoryDataService**:

```
…
HttpClientInMemoryWebApiModule.forRoot(
      InMemoryDataService, { dataEncapsulation: false }
)
```

The class "**src/app/in-memory-data.service.ts**" is generated by the following command:

```
ng generate service InMemoryData
```

84. The "**in-memory-data.service.ts"** file replaces "movies.ts" file. Update this file to:

```
import { InMemoryDbService } from 'angular-in-memory-web-api';
import { Movie } from './movie';
import { Injectable } from '@angular/core';

@Injectable({
      providedIn: 'root'
})

export class InMemoryDataService implements InMemoryDbService {
      constructor() { }

      createDb() {
      const movies: Movie[] = [
{ id: 11, name: 'Avatar', url:'https://www.youtube.com/embed/
6ziBFh3V1aM'},
{ id: 12, name: 'Divergent',url:'https://www.youtube.com/embed/
sutgWjz10sM'},
{ id: 13, name: 'Inception', url:'https://www.youtube.com/embed/
YoHD9XEInc0'},
{ id: 14, name: 'Star Wars', url:'https://www.youtube.com/embed/
FDXmcZ1_D-o'},
```

```
      { id: 15, name: 'Ready Player One', url:'https://www.youtube.com/
embed/cSp1dM2Vj48'},
      { id: 16, name: 'Matrix', url:'https://www.youtube.com/embed/
vKQi3bBA1y8'},
      { id: 17, name: 'The Revenant',url:'https://www.youtube.com/embed/
LoebZZ8K5N0'},
      { id: 18, name: 'American Hustle', url:'https://www.youtube.com/
embed/ST7a1aK_lG0'},
      { id: 19, name: 'The Hunger Games', url:'https://www.youtube.com/
embed/mfmrPu43DF8'},
      { id: 20, name: 'The Accountant', url:'https://www.youtube.com/embed/
DBfsgcswlYQ' }
    ];
    return {movies};
  }

      // Overrides the genId method to ensure that a movie always has an
id.
      // If the movie array is empty,
      // the method below returns the initial number (11).
      // if the movie array is not empty, the method below returns the
highest
      // movie id + 1.
      genId(movies: Movie[]): number {
            return movies.length > 0 ? Math.max(...movies.map(movie =>
            movie.id)) + 1 : 11;
      }
}
```

85. Open the file "movie.service.ts", import the following symbols:

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
```

86. Inject in **MovieService** class the **HttpClient** into the constructor in a private property called **http**.

87. Add a private method **log()** in the **MovieService** class:

```
private log(message: string) {
      this.messageService.add(`MovieService: ${message}`);
 }
```

88. Define the **moviesUrl** of the form **:base/:collectionName** with the address of the **movies** resource on the server. Here base is the resource to which requests are made, and **collectionName** is the **movies** data object in the "**in-memory-data-service.ts**" file:

```
private moviesUrl = 'api/movies';  // URL to web api
```

89. The current **MovieService.getMovies()** uses the **RxJS of()** function to return an array of **movies** as an **Observable<Movie[]>**. Convert that method to use **HttpClient**:

```
getMovies (): Observable<Movie[]> {
        return this.http.get<Movie[]>(this.moviesUrl)
}
```

Run the app in the browser.

Error handling

90. Things go wrong, especially when you're getting data from a remote server. The **MovieService.getMovies()** method should catch errors and do something appropriate. To catch errors, "**pipe**" the observable result from **http.get()** through an **RxJS catchError()** operator. Import the **catchError** symbol from **rxjs/operators**, along with some other operators that will be needed later:

```
import { catchError, map, tap } from 'rxjs/operators';
```

91. Extend the **observable** result with the **.pipe()** method and give it a **catchError()** operator:

```
getMovies (): Observable<Movie[]> {
  return this.http.get<Movie[]>(this.moviesUrl)
    .pipe(
      catchError(this.handleError<movie[]>('getMovies', []))
    );
}
```

The **catchError()** operator intercepts an **Observable** that failed. It passes the error an error handler that can do what it wants with the error. The following **handleError()** method reports the error and then returns an innocuous result so that the application keeps working.

92. The following **handleError()** will be shared by many **MovieService** methods so it's generalized to meet their different needs. Instead of handling the error directly, it returns an error handler function to **catchError** that it has configured with both the name of the operation that failed and a safe return value. Add the **handleError()** method to the **MovieService** class:

```
/**
* Handle Http operation that failed.
* Let the app continue.
* @param operation - name of the operation that failed
* @param result - optional value to return as the observable
* result
*/
private handleError<T> (operation = 'operation', result?: T) {
  return (error: any): Observable<T> => {
    // TODO: send the error to remote logging infrastructure
    console.error(error); // log to console instead

    // TODO: better job of transforming error for user
    // consumption
    this.log(`${operation} failed: ${error.message}`);

    // Let the app keep running by returning an empty result.
    return of(result as T);
  };
}
```

After reporting the error to console, the handler constructs a user friendly message and returns a safe value to the app so it can keep working.
Because each service method returns a different kind of **Observable** result, **handleError()** takes a type parameter so it can return the safe value as the type that the app expects.

93. The **MovieService** methods will **tap** into the flow of **observable** values and send a message (via **log()**) to the message area at the bottom of the page:

```
getMovies (): Observable<Movie[]> {
      return this.http.get<Movie[]>(this.moviesUrl)
        .pipe(
            tap(_ => this.log('fetched Movies')),
```

```
            catchError(this.handleError<Movie[]>('getMovies',
            []))
        );
}
```

94. Do the same for the **MovieService.getMovie()** method to make the movie request:

```
/** GET movie by id. Will 404 if id not found */
getMovie(id: number): Observable<Movie> {
  const url = `${this.moviesUrl}/${id}`;
  return this.http.get<Movie>(url)
   .pipe(
      tap(_ => this.log(`fetched movie id=${id}`)),
      catchError(this.handleError<Movie>(`getMovie id=${id}`))
   );
}
```

Update Movies

95. When a movie's name in the movie detail view is edited, as is typed, the movie name updates the heading at the top of the page. But when the "**go back button**" is clicked, the changes are lost. To make these changes persistent, it is needed write them back to the server. At the end of the movie detail template, add a "**save button**" with a **click** event binding that invokes a new component method named **save()**. Change the style in order to separate in 5px the "**go back button**" and the "**save button**".

96. Add the following **save()** method, which persists **movie** name changes using the movie service **updateMovie()** method and then navigates back to the previous view:

```
save(): void {
      this.movieService.updateMovie(this.movie)
            .subscribe(() => this.goBack());
 }
```

97. Add "**MovieService.updateMovie()**". The overall structure of the **updateMovie()** method is similar to that of **getMovies()**, but it uses **http.put()** to persist the changed movie on the server:

```
updateMovie (movie: Movie): Observable<any> {
```

```
        return this.http.put(this.moviesUrl, movie,
              httpOptions).pipe(
              tap(_ => this.log(`updated movie id=${movie.id}`)),
              catchError(this.handleError<any>('updateMovie'))
              );
}
```

The **HttpClient.put()** method takes three parameters:

(1) the URL;

(2) the data to update (the modified movie in this case);

(3) options.

The URL is unchanged. The movies web API knows which movie to update by looking at the movie's id.

The movies web API expects a special header in HTTP save requests. That header is in the httpOptions constant defined in the **MovieService**:

```
const httpOptions = {
      headers: new HttpHeaders({ 'Content-Type': 'application/
      json' })
};
```

Run the app in the browser.

98. To add a new **movie** insert the following into the **MoviesComponent** template, just after the heading:

```
<div>
      <label>Movie name:
            <input #movieName />
      </label>

      <label>Movie url:
            <input #movieUrl />
      </label>
      <button (click)="add(movieName.value, movieURL.value);
      movie.value=''">
            add
      </button>
</div>
```

Add the "**add()**" method in the class:

```
add(name: string, url: string): void {
      name = name.trim();
      url = url.trim();
      if (!name || !url) { return; }
      this.movieService.addMovie({ name, url } as Movie)
      .subscribe(movie => {
            this.movies.push(movie);
      });
}
```

99. Add the following addMovie() method to the **MovieService** class:

```
addMovie (movie: Movie): Observable<Movie> {
  return this.http.post<Movie>(this.moviesUrl, movie, httpOptions)
   .pipe(
      tap((newMovie: Movie) => this.log(`added movie w/ id=$
      {newMovie.id} id=${newMovie.url}`)),
      catchError(this.handleError<Movie>('addMovie'))
   );
}
```

MovieService.addMovie() differs from **updateMovie** in two ways:

(1) it calls **HttpClient.post**() instead of put().

(2) it expects the server to generate an id for the new **movie**, which it returns in the **Observable<Movie>** to the caller.

Run the app in the browser. Add new movies.

100. To delete movies, each movie in the movies list should have a delete button. Add the following button element to the **MoviesComponent** template, after the movie name in the repeated **<li>** element:

```
<ul class="movies">
      <li *ngFor="let movie of movies">
            <a routerLink="/detail/{{movie.id}}">
                  <span class="badge">{{movie.id}}</span>
                  {{movie.name}}
            </a>
            <button class="delete" title="delete
            movie" (click)="delete(movie)">x</button>
      </li>
</ul>
```

101. Add the following style to the **MoviesComponent**:

```
button {
      background-color: #eee;
      border: none;
      padding: 5px 10px;
      border-radius: 4px;
      cursor: pointer;
      cursor: hand;
      font-family: Arial;
}

button:hover {
      background-color: #cfd8dc;
}

button.delete {
      position: relative;
      left: 194px;
      top: -32px;
      background-color: gray !important;
      color: white;
}
```

Run the code in the browser.

102. Add the "**delete()**" method to the **MovieComponent**:

```
delete(movie: Movie): void {
      this.movies = this.movies.filter(m => m !== movie);
      this.movieService.deleteMovie(movie).subscribe();
}
```

103. Add the **"deleteMovie()"** method to the **MovieService** class:

```
deleteMovie (movie: Movie | number): Observable<Movie> {
      const id = typeof movie === 'number' ? movie : movie.id;
      const url = `${this.moviesUrl}/${id}`;

      return this.http.delete<Movie>(url, httpOptions)
        .pipe(
            tap(_ => this.log(`deleted movie id=${id}`)),
```

```
    catchError(this.handleError<Movie>('deleteMovie')));}
```

Note that, it calls **HttpClient.delete**. The URL is the **movies** resource URL plus the id of the **movie** to delete. Data is not send with put or post.

Search by Movie Name

104. The following operation consist of adding a movies search feature to the **Dashboard**. As the user types a name into a search box, repeated HTTP requests will be made for movies filtered by that name. The goal is to issue only as many requests as necessary. Start by adding a **searchMovies** method to the **MovieService:**

```
searchMovies(term: string): Observable<Movie[]> {
   if (!term.trim()) {
     // if not search term, return empty movie array.
     return of([]);
   }
   return this.http.get<Movie[]>(`${this.moviesUrl}/?name=$
   {term}`)
     .pipe(
       tap(_ => this.log(`found movies matching "${term}"`)),
       catchError(this.handleError<Movie[]>('searchMovies',
       []))
     );
 }
```

105. Add search element <app-movie-search> to the bottom of the **Dashboard** component. Create a **MovieSearchComponent** with the CLI.

106. Replace the generated **MovieSearchComponent** template with a text box and a list of matching search results like this:

```
<div id="search-component">
      <h4>Movie Search</h4>
      <input #searchBox id="search-
      box" (input)="search(searchBox.value)" />

      <ul class="search-result">
            <li *ngFor="let movie of movies$ | async">
                  <a routerLink="/detail/{{movie.id}}">
                        {{movie.name}}
```

```
                </a>
            </li>
        </ul>
    </div>
```

As the user types in the search box, an **input** event binding calls the component's search() method with the new search box value. As expected, the ***ngFor** repeats movie objects. Look closely and you'll see that the ***ngFor** iterates over a list called **movies$**, not movies. The **$** is a convention that indicates movies$ is an **Observable**, not an array. The ***ngFor** can't do anything with an **Observable**. But there's also a pipe character (|) followed by async, which identifies Angular's **AsyncPipe**. The **AsyncPipe** subscribes to an **Observable** automatically so you won't have to do so in the component class.

107. Replace the generated **MovieSearchComponent** class and metadata as follows:

```
import { Component, OnInit } from '@angular/core';
import { Observable, Subject } from 'rxjs';
import { debounceTime, distinctUntilChanged, switchMap} from
'rxjs/operators';

import { Movie } from '../Movie';
import { MovieService } from '../movies/movie.service';

@Component({
  selector: 'app-movie-search',
  templateUrl: './hero-movie.component.html',
  styleUrls: [ './hero-movie.component.css' ]
})
export class MovieSearchComponent implements OnInit {
  movies$: Observable<Movie[]>;
  private searchTerms = new Subject<string>();

  constructor(private movieService: MovieService) {}

  // Push a search term into the observable stream.
  search(term: string): void {
    this.searchTerms.next(term);
  }

  ngOnInit(): void {
    this.movies$ = this.searchTerms.pipe(
```

```
      // wait 300ms after each keystroke before considering the
term
      debounceTime(300),

      // ignore new term if same as previous term
      distinctUntilChanged(),

      // switch to new search observable each time the term
changes
      switchMap((term: string) =>
this.movieService.searchMovies(term)),
    );
  }
}
```

108. Add the following style to the **MovieSearchComponent:**

```
  .search-result li {
    border-bottom: 1px solid gray;
    border-left: 1px solid gray;
    border-right: 1px solid gray;
    width: 195px;
    height: 16px;
    padding: 5px;
    background-color: white;
    cursor: pointer;
    list-style-type: none;
  }

  .search-result li:hover {
    background-color: #607D8B;
  }

  .search-result li a {
    color: #888;
    display: block;
    text-decoration: none;
  }

  .search-result li a:hover {
    color: white;
  }
  .search-result li a:active {
```

```
  color: white;
}
#search-box {
  width: 200px;
  height: 20px;
}


ul.search-result {
  margin-top: 0;
  padding-left: 0;
}
```

109. Finally, run the code in the browser and enjoy the app.