# Workshop - Angular with TypeScript

with ***Andrei Antal (contact me at: antal.a.andrei@gmail.com)***

## 0. Introduction

The starter project we're going to use was created using the Angular CLI (https://cli.angular.io/). This tool allows us to quickly scaffold an Angular project that's ready to build upon. Please take a moment to familiarize yourself with the CLI since we'll be using it to generate the required elements in the project.

All the sample code will be available on github, so feel free to consult it whenever you're stuck. Also don't hesitate to ask me about anything that's unclear but be patient :)

Angular is an awesome framework so let's get to work. And besides everything, let's have fun!

### Initial setup

#### 1. Environment

Make sure you have Node installed on your machine (Node version >= 8.0 and NPM >= 5).
If not, install node and follow the instructions from their homepage: https://nodejs.org/en/download/

#### 2. Installing the Angular CLI

Open your terminal/command prompt application and install the Angular CLI globally because it will allow us to access the **ng** command from anywhere.

```
# install the Angular CLI
$ npm install -g @angular/cli
```

#### 3. Get the code and install the dependencies

```
# navigate to your work folder and clone the git repository
$ git clone https://github.com/IncrementalCommunity/Angular2_Workshop_Voxxed.git

# navigate to the project folder
$ cd Angular2_Workshop_Voxxed

# install project dependencies (make sure you're in the project directory)
$ npm install
```

```
# start the application
$ ng serve
```

Open the browser and enter the following address: http://localhost:4200. If you see the welcome message, congratulations, you're all set!

# 1. Application bootstrap

**1.1)** Open src/index.html. Notice the <app-root> element. This is where our Angular application will be rendered.

```
// src/index.html
13   <body>
14     <app-root></app-root>
15   </body>
```

**1.2)** Open src/main.ts. This is the entry point for our application. In our application we are using the runtime (JIT) compiler so we imported **platformBrowserDynamic**. **bootstrapModule** takes the root module (**AppModule**) to kick start bootstrap. We need to import the **AppModule** in order to pass it to bootstrap.

```
 2   import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
 4   import { AppModule } from './app/app.module';
11   platformBrowserDynamic().bootstrapModule(AppModule);
```

**1.3)** Open src/app/app.module.ts. **AppModule** is our root module. Take a first look at the module annotation (**@NgModule**). We pass a configuration object in order to set up our application module. Here are the code highlights:

- Within declarations (line 9) we set application wide components, directives and pipes.
- We add the modules we want to import like **BrowserModule** (line 13).
- Finally we indicate the root component that will be used during bootstrap (line 18).

**1.4)** Open src/app/app.component.ts. **AppComponent** is our root component. The bootstrap process will use the CSS selector '**app-root**' to locate the DOM element where we instantiate the component. This selector must match the one used in src/index.html (line 13). For components we use the **@Component** annotation. Notice also how we used export in front the component (ES6 class) to make it available to the import in src/app/app.module.ts (line 6).

The Angular bootstrap process will vary depending on your setup (ES5/ES6/TypeScript) or if you want to use the Ahead of Time compilation. In all cases, our Application starts with a call passing in the **root module (AppModule)** that sets the global dependencies and the **root component** (**App**).

Read more:
https://angular.io/guide/bootstrapping

# 2. Components

We will start to build our movies application. This section focuses on building your first components, learning about the component architecture and the data flow between them.

**2.1) Create your first component.**

- In the `src/app` folder create a new folder called `components`
- Using the terminal, navigate to this directory
- Create a new component using the angular-cli (make sure you have it globally installed):

```
$ ng g component MovieCard
```

- Check out all the generated files and notice that the component is already registered in the **`AppModule`** (in declarations, line 12). One of the main advantages of the CLI is that it takes care of both component scaffolding (creating the typescript file, the template and style files) and registering the component in the module. This saves you a lot of time. Also, any element created with the help of the CLI (component, pipe, service etc.) adheres to the Angular Style Guide (https://angular.io/guide/styleguide)
- Open the newly created component in `src/app/components/movie-card.component.ts` and have a look around. Notice that:
    - The selector for the component is **`app-movie-card`** (line 4). That's how you will reference it in other component templates
    - The template file and the styles file can be found in the same folder as the component at the specified locations (line 5 and 6)
- Create the following properties on the **`MovieCardComponent`** class and initialize them with the following values:

```
poster = 'https://images-na.ssl-images-
amazon.com/images/M/MV5BMjEwMzMxODIzOV5BMl5BanBnXkFtZTgwNzg3OTAzMDI@._V1_SX300.jpg';
title = 'Rogue One';
year = 2016;
duration = '133 min';
genre = 'Action, Adventure, Sci-Fi';
plot = 'All looks lost for the Rebellion against the Empire as they learn of the
existence of a new super weapon, the Death Star. Once a possible weakness in its
construction is uncovered, the Rebel Alliance must set out on a desperate mission to
```

```
steal the plans for the Death Star. The future of the entire galaxy now rests upon its success.';
```

- Open the template file in `src/app/components/movie-card.component.html` and display all the properties (`poster`, `title`, `year`, `duration`, `genre` and `plot`) of the component using the `{{}}` binding (eg **`<p>{{title}}</p>`**).

**2.2) Display the component**

- In order to see and use our new component, we need to reference it in some other template.
- Open src/app/app.component.html and replace everything with the tag:

```
<app-movie-card></app-movie-card>
```

- Check out the result in the browser. You should see the movie details.

2.3) **Displaying the movie poster** - binding to image src

- In order to display a proper image we need to add an <img> element to the **`MovieCardComponent`**
- Bind the src attribute to the poster property: **`<img [src]=”poster">`.** The [] creates a relationship between the src property on the img element and the poster property on the component. Whenever the class property will change, the element property will change as well.
- Check out more about the template syntax here: https://angular.io/guide/template-syntax

**2.4) Displaying a list of movie cards**

- Before we display a list of movie cards take the time to apply a minimalist style to the card so that we can distinguish information. In order so set up styling for this example up, you can follow the information in **Annex 1 - Styling the app**. But also feel free to implement any design you see fit.
- Using the terminal, navigate to the components directory
- Create a new component using the angular-cli (make sure you have it globally installed):

```
$ ng g component MovieList
```

- Check out all the generated files and notice that the component is already registered in the AppModule (in declarations)
- Create a property on the **`MovieListComponent`** component class, called **`movieList`** and assign it the following JSON:

```
movieList = [
      {
        title: 'Rogue One',
        year: '2016',
        duration: '133 min',
        genre: 'Action, Adventure, Sci-Fi',
```

```
        plot: 'All looks lost for the Rebellion against the Empire as they learn of
the existence of a new super weapon, the Death Star. Once a possible weakness in its
construction is uncovered, the Rebel Alliance must set out on a desperate mission to
steal the plans for the Death Star. The future of the entire galaxy now rests upon
its success.',
        poster: 'https://images-na.ssl-images-
amazon.com/images/M/MV5BMjEwMzMxODIzOV5BMl5BanBnXkFtZTgwNzg3OTAzMDI@._V1_SX300.jpg',
      },
      {
        title: 'Goodfellas',
        year: '1990',
        duration: '146 min',
        genre: 'Biography, Crime, Drama',
        plot: 'Henry Hill might be a small time gangster, who may have taken part in
a robbery with Jimmy Conway and Tommy De Vito, two other gangsters who might have set
their sights a bit higher. His two partners could kill off everyone else involved in
the robbery, and slowly start to think about climbing up through the hierarchy of the
Mob. Henry, however, might be badly affected by his partners\' success, but will he
consider stooping low enough to bring about the downfall of Jimmy and Tommy?',
        poster: 'https://images-na.ssl-images-
amazon.com/images/M/MV5BNThjMzczMjctZmIwOC00NTQ4LWJhZWItZDdhNTk5ZTdiMWFlXkEyXkFqcGdeQ
XVyNDYyMDk5MTU@._V1_SX300.jpg'
      },
      {
        title: 'Saving Private Ryan',
        year: '1998',
        duration: '169 min',
        genre: 'Drama, War',
        plot: 'Opening with the Allied invasion of Normandy on 6 June 1944, members
of the 2nd Ranger Battalion under Cpt. Miller fight ashore to secure a beachhead.
Amidst the fighting, two brothers are killed in action. Earlier in New Guinea, a
third brother is KIA. Their mother, Mrs. Ryan, is to receive all three of the grave
telegrams on the same day. The United States Army Chief of Staff, George C. Marshall,
is given an opportunity to alleviate some of her grief when he learns of a fourth
brother, Private James Ryan, and decides to send out 8 men (Cpt. Miller and select
members from 2nd Rangers) to find him and bring him back home to his mother...',
        poster: 'https://images-na.ssl-images-
amazon.com/images/M/MV5BZjhkMDM4MWItZTVjOC00ZDRhLThmYTAtM2I5NzBmNmNlMzI1XkEyXkFqcGdeQ
XVyNDYyMDk5MTU@._V1_SX300.jpg'
      },
      {
        title: 'L.A. Confidential',
        year: '1997',
        duration: '138 min',
        genre: 'Crime, Drama, Mystery',
        plot: 'As corruption grows in 1950s LA, three policemen - one strait-laced,
one brutal, and one sleazy - investigate a series of murders with their own brand of
justice.',
```

```
        poster: 'https://images-na.ssl-images-
amazon.com/images/M/MV5BMDBlYzAwZDktNzM2MS00YzBlLWI4ODQtZTlkNmMxZDc3NGRkXkEyXkFqcGdeQ
XVyMTQxNzMzNDI@._V1_SX300.jpg'
      }
  ];
```

- In the template of the component add a movie card tag:

```
<div class="MovieList">
  <app-movie-card></app-movie-card>
</div>
```

- In order to repeat a collection of components we can use the ***ngFor** (more details about the directive here: https://angular.io/guide/template-syntax#ngforof) directive using this syntax:

```
<app-movie-card *ngFor="let movie of movieList"></app-movie-card>
```

- In order to see the new list component, replace the **<app-movie-card></app-movie-card>** with **<app-movie-list></app-movie-list>** in app.component.html
- Check the results in the browser

### 2.5) Passing data into components

We see that the list repeats the same card, because it contains the static information from the **MovieCard** component. We need a way to pass in the custom movie data. In order to pass data to a component we need to use **Inputs**. This is achieved by adding the **@Input** decorator to properties that we want to expose as such. Then we can pass data using the attribute binding ([])

- Add an **@Input()** annotation to each property of the **MovieCardComponent** and remove static string values (ex: **@Input() poster;** ). In order to use it don't forget to import the decorator:

```
import { Component, OnInit, Input } from '@angular/core';
```

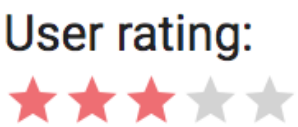
- In the **MovieListingComponent** template add property bindings for the card inputs:

```
<app-movie-card
    *ngFor="let movie of movieList"
    [title]="movie.title"
    [poster]="movie.poster"
    [year]="movie.year"
    [duration]="movie.duration"
    [genre]="movie.genre"
    [plot]="movie.plot"
```

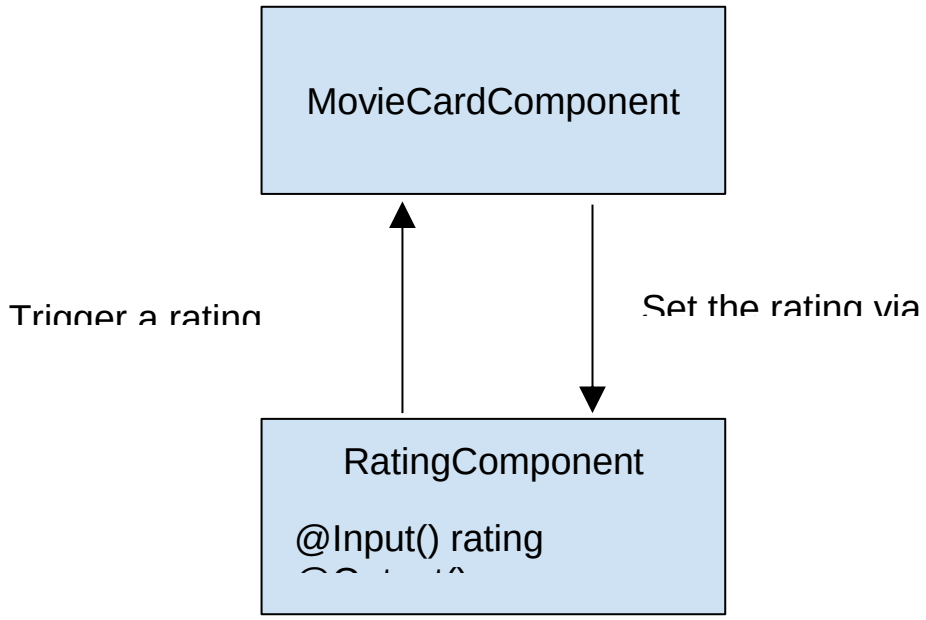
```
></app-movie-card>
```

- Check the result, you should see 4 different movie cards

**2.6) Implement the user rating - unidirectional data flow**




The **RatingComponent** is composed of 5 star icons. Whenever you click on a certain star, all the stars before it will become colored ( eg, if you click on star no.3 all stars from left to right up to and including star number 3 will become colored).
It is important to note that the **RatingComponent** will not hold it's own state (the rating). Rather it will receive it via the **@Input** binding and render it accordingly. This type of component (that does not hold it's own state) is often referred to as a "dumb" component. However, the action to click on the star (change the rating) it and will be handled by the **RatingComponent** but only to delegate the click event to the parent through the **@Output** binding. This way of data flow, where information flows from parent to child and events flow from child to parent is called unidirectional data flow.
Because **RatingComponent** is a child of **MovieCard**, on every click event the **MovieCard** will receive the new rating and update it accordingly, passing it down back to the **RatingComponent**. This behaviour is displayed in the following image:



- Using the terminal, navigate to the components directory
- Create a new component using the angular-cli:

```
$ ng g component Rating
```

- Check out all the generated files and notice that the component is already registered in the **AppModule** (in declarations)
- Add the **<app-rating>** component inside the **MovieCard** template.
- First let's add a star icon. In the **rating.component.html** file replace the text with:

```
<div class="Rating">
  User rating:
  <div>
    <i class="material-icons">star</i>
  </div>
</div>
```

- Now we need to repeat the star icon 5 times. Although it might seem tempting to just copy the icon code 5 times this will become hard to maintain and scale. Instead we will use the ***ngFor** directive. Although we stated earlier that the **RatingComponent** will be a dumb component, it can however hold it's own ui state since it is not state at the application level it's just internal state used for display purposes. It will still receive the number of stars to light up from its parent. We first need to create a collection that holds the state of each star. Add the following property to the **RatingComponent** class:

```
ratings = [ false, false, false, false, false ];
```

This property will hold the colored state of each star.

- Now we can iterate through this collection and generate our star array. In the template add the ***ngFor** directive. This will create 5 instances of the icon each referencing the current rating element in the collection (let rating). Replace the old **<i>** with the new one:

```
<i class="material-icons" *ngFor="let rating of ratings">star</i>
```

- Next we need to add some styles for the colored stars. In the rating.component.css file add the following style:

```
.Rating-Star--colored{
  color: #ee6e73;
}
```

- In order to dynamically add/change classes on elements we can use the **ngClass** directive. More on how **ngClass** works: https://angular.io/guide/template-syntax#ngclass

```
<i
  class="material-icons"
  *ngFor="let rating of ratings;"
  [ngClass]="{'Rating-Star--colored': rating}"
>star</i>
```

In order to see the effect, change some of the values in the **ratings** property from **false** to **true**.

- Now that we can color the stars, we need to implement the rating functionality. We need to react when the user clicks on one of the stars (so that we can color them accordingly). We can do this by using a (click) binding. The click binding takes a handler function as a parameter that will trigger each time a click event occurs.

```
<i
  *ngFor="let rating of ratings"
  class="material-icons Rating-Star"
  [ngClass]="{'Rating-Star--colored': rating}"
  (click)="changeRating()"
>
```

- Although this code allows us to react to user clicks we have one problem. How do we know which star was clicked? We should pass the handler function an index of the clicked star. We can keep track of the index by registering it in the ***ngFor** directive. Update it in the template:

```
*ngFor="let rating of ratings; let i = index"
```

- Now the i variable will hold the current iteration index for each element. Next we need to add this index as a parameter to the handler function:

```
(click)="changeRating(i)"
```

- Let's add an implementation for the function so that we can complete this part of the functionality. In the **RatingComponent** class add the following method:

```
changeRating(index) {
  this.ratings = this.ratings.map((r, i) => ({colored: i <= index}));
}
```

Check if the component is working correctly. On each click on any star, we should all the stars to the left of the selection being colored.

### 2.7) Create data flow between MovieCardComponent and RatingComponent

For the moment the **RatingComponent** is holding its own state. Although this works fine, we want the rating state to be maintained by the parent component (the **MovieCard** component). We need to implement the correct data flow between the components.

- First we need to pass in the rating as an Input. Add a rating input in the **RatingComponent**. Don't forget to import **Input**.

```
@Input() rating;
```

- Next we need to pass the parameter from the parent. In the **MovieCardComponent** template add a **[rating]** binding on the **<app-rating>** element and create a rating property on the component class:

```
 <app-rating [rating]="rating"></app-rating>
```

- Also add a rating property on the **MovieCardComponent:**

```
rating = 0;
```

- We now need a way to initially set the rating from the Input. In the **onInit** lifecycle hook (more about lifecycle hooks here: https://angular.io/guide/lifecycle-hooks ) of the **RatingComponent** add the following code (similar to the changeRating method):

```
ngOnInit() {
    this.ratings = this.ratings.map((r, i) => i <= this.rating);
}
```

- We can further improve the code by extracting the array mapping into into an **updateRating()** method:

```
ngOnInit() {
    this.updateRating();
}
updateRating() {
    this.ratings = this.ratings.map((r, i) => i <= this.rating);
}
```

- We now need to take care of the rating selection. For this we need an **@Output** binding (don't forget to import it):

```
@Output() onChangeRating = new EventEmitter();
```

The **EventEmmiter** allows us to communicate between component using it's **.emit()** method. More on **EventEmmiter** here: https://angular.io/guide/template-syntax#custom-events-with-eventemitter; Dont forget to import **Output** and **EventEmmiter** from @angular/core:

```
import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';
```

- Update the **changeRating()** method to emit a new change event whenever a star is clicked:

```
changeRating(index) {
    this.onChangeRating.emit(index);
}
```

- We've now set up a proper binding mechanism. We need to implement functionality on the **MovieCard** component now. On the template update the **<app-rating>** element:

```
<app-rating
    [rating]="rating"
    (onChangeRating)="handleRatingChange($event)"
></app-rating>
```

- Now add a **handleRatingChange** method in the component class:

```
handleRatingChange(index) {
    this.rating = index;
}
```

Check the code functionality in the browser. Although the mechanism is working correctly (events are triggered and bindings are updated - just add some console.log's to check) the component still doesn't work as intended.

- The last step is to update the star rating (the rating array in the **RatingsComponent**) whenever the rating Input changes. We can accomplish this inside the **ngChanges()** lifecycle hook which triggers each time an input has changed. Add the following code in the **RatingsComponent** class.

```
ngOnChanges(changes) {
  this.updateRating();
```

```
}
```

As a good practice, each time we use a lifecycle hook we need to add the corresponding interface to the class implementation list (and don't forget to import it from @angular/core):

```
export class RatingComponent implements OnInit, OnChanges
```

### BONUS TASKS:

- Style the movie card
- Add a nice header

# 3. Services

As of now we've been holding the movie data inside the component. Although convenient it's not the best way to go, especially if other parts of your application need to access the data. For this we can use **SERVICES**. They are classes on which we apply the **@Injectable()** annotation. The Injectable annotation make the service available for available for Dependency Injection. More on DI here: https://angular.io/guide/dependency-injection

First, we make a model for our Movie entity.

#### 3.1) Create the Movie model

- Create a folder called **services** in /src/app
- Create a file called **Movie.ts** - this will hold the movie class:

```
export class Movie {
  constructor(
    public title: string,
    public year: number,
    public duration: string,
    public genre: string,
    public plot: string,
```

```
    public poster?: string,
  ) {}
}
```

The public parameters in the constructor make those properties available on the class (we don't need to explicitly declare them. Non public/private constructor parameters are just regular function arguments and are not available outside the constructor.
The "?" before the poster marks that argument as being optional. Thus when we create a movie object we may or may not include a poster. The other arguments are mandatory.

### 3.2) Create the MovieService service

- Uning the terminal, navigate to the services directory and create a service using the angular CLI:

```
$ ng g service Movies
```

- Check out all the generated files especially the **`@Injectable()`** decorator on the **`MoviesService`** class.
- Besides marking the service as injectable, in order for the service to be available for DI, we need to declare it in the module providers list.
- Go to **`app.module.ts`** and add the service to the providers list, after you import it.

```
import { MoviesService } from './services/movies.service';

@NgModule({
  ...
  providers: [MoviesService],
  ...
})
```

Next we need to move the movie list JSON from the **`MovieListComponent`** to the service

- Create a private **movies** property in the **`MoviesService`** class; it will hold an array of **`Movie`** instances:

```
private movies: Movie[];
```

- Move the movie list from the **`movieList`** property in **`MovieListComponent`** to the **`movies`** property in **`MoviesService`** (but keep an empty **`movieList`** property on **`MovieListComponent`**)
- We also need to convert the objects in the list into **`Movie`** objects. After you import **`Movie`** in the **`MoviesService`** class replace each element of the array with a new **`Movie`** class instance like this:

```
{
```

```
    title: 'Rogue One',
    year: '2016',
    duration: '133 min',
    genre: 'Action, Adventure, Sci-Fi',
    plot: 'All looks lost for the Rebellion against the Empire as they learn of the existence of a new super weapon, the Death Star. Once a possible weakness in its construction is uncovered, the Rebel Alliance must set out on a desperate mission to steal the plans for the Death Star. The future of the entire galaxy now rests upon its success.',
    poster: 'https://images-na.ssl-images-amazon.com/images/M/MV5BMjEwMzMxODIzOV5BMl5BanBnXkFtZTgwNzg3OTAzMDI@._V1_SX300.jpg',
},
```

replace with

```
new Movie(
    'Rogue One',
    2016,
    '133 min',
    'Action, Adventure, Sci-Fi',
    'All looks lost for the Rebellion against the Empire as they learn of the existence of a new super weapon, the Death Star. Once a possible weakness in its construction is uncovered, the Rebel Alliance must set out on a desperate mission to steal the plans for the Death Star. The future of the entire galaxy now rests upon its success.',
'https://images-na.ssl-images-amazon.com/images/M/MV5BMjEwMzMxODIzOV5BMl5BanBnXkFtZTgwNzg3OTAzMDI@._V1_SX300.jpg' ),
```

- Do this for the rest of the films
- Create a method in the service called **`getMovies()`** that return the private **`movieList`** property

```
getMovies() {
   return this.moviesList;
}
```

### 3.2) Using the MovieService service in MovieListComponent

- In the **`MovieListComponent`** Class -> After you import the service, you need to declare it as a public/private property in the component constructor. The compiler will now to inject the right instance of the service.

```
constructor(private moviesService: MoviesService) { }
```

- In order to restore previous functionality in the Movie list component we need to get the list of films from the service and pass it to the **movieList** property. We can do this in the **ngOnInit()** lifecycle hook:

```
ngOnInit() {
  this.movieList = this.moviesService.getMovies();
}
```

### 3.3) Passing a movie to the MovieCardComponent

Now that we have a list of Movie objects we can pass these down to the **MovieCard** component, as opposed to each individual property.

- In the **MovieCardComponent** class, remove all the property **@Input** bindings and replace them with one **@Input() movie** binding.
- In the **MovieCardComponent** template add "**movie.**" to all the previous property bindings (eg. **movie.title** instead of **title**)
- In **MovieListComponent** template replace the previous bindings with only one movie binding:

```
<app-movie-card
    *ngFor="let movie of movieList"
    [movie]="movie"
></app-movie-card>
```

### 3.4) [TASK] Adding a delete movie button

STEPS:

**A. MovieCardComponent**

1. Create in **@Input index** binding so that we know what movie index does the card represent
2. Add a delete button in the **MovieCard** component template
3. Create a click handler for the button
4. Create an **@Output() onDelete** binding
5. Emit the index from the click handler

**B. MovieService**

1. Create a **removeMovie(index)** method that deletes the movie at the respective index (you can just use splice)

C. **MovieListComponent**

1. In the **MovieList** template add the **[index]** binding to the **<app-movie-card>** component; you can obtain the index by saving it from the ***ngFor** directive (see **RatingComponent**)
2. Add an **(onDelete)** binding with a **handleDeleteMovie** method; dont forget to pass **$event** in the handler
3. In the **MovieList** class, implement **handleDeleteMovie(index)**
    a. Call the **removeMovie** method on the service passing it the index
    b. Get the new movie list by calling the **getMovies()** and assigning the result to the **movieList** property (and since we're using this in the **OnInit** method as well, consider extracting it to a separate method)

# 4. Router

For now we have only one page for our application. We would of course like to expand the functionality of the application and implement a master/detail architecture.
Routing can be set up pretty easily. Angular uses component routing which means that routes are actually container components. We'll create 2 "pages", **home** and **details**
You can find a more in depth overview on routing here: https://angular.io/guide/router

### 4.1) Create the home and details component

- Create a folder called **pages** in `/src/app`
- In the **pages** directory create 2 components using the Angular CLI:

```
$ ng g component Home
$ ng g component Details
```

- Move the **<app-movie-list></app-movie-list>** component from `app.component.html` to `home.component.html`

### 4.2) Defining routes

In order to use Angular router we need to include it's dependencies globally during bootstrap in `src/app/app.module.ts`. As this will be our root component for routing we used **RouterModule.forRoot** passing the **Routes**.

- We need to define routes for **Home** and **Details**. We can use ***the empty path*** to redirect empty routes to the Home component and the special path ***'**'*** to match any remaining routes to a component to render a not found component (Eg: 404). Add the following in the app.module in imports:

```
imports: [
...
  RouterModule.forRoot([
      {
        path: '',
        component: HomeComponent,
      },
      {
        path: 'movie,
        component: DetailsComponent,
      },
      {
        path: 'movie/:id',
        component: DetailsComponent,
      },
      {
        path: '**',
        redirectTo: '/'
      }
    ]),
```

- We also need to import the RouterModule:

```
import { RouterModule } from '@angular/router';
```

Notice that we created 3 routes:

- **The default route** - this will leave us in the movie list screen
- **The create route** - will match /movie url's (without an id)
- **The edit route** - will match /movie/:id routes, where id is a placeholder for actual values
- In case we input a rout that does not match any of the above, we're automatically redirected to **/**

### 4.3) Add a router outlet

- We need to add a placeholder to render the content for the different routes on our Application. We can use **<router-outlet>** so child components can render its content depending on the current route. Add it to the **AppComponent** template, in place of the movie list that was previously deleted:

```
<router-outlet></router-outlet>
```

Now try to navigate to various url's (from the browser address bar) and see the results.

### 4.4) Add navigation

To navigate between pages we use **routerLink** directive taking the route path.

- In the **HomeComponent** add the following link:

```
<a routerLink="/movie">Add new</a>
```

- In the **DetailsComponent** add the following link:

```
<a routerLink="/">Cancel</a>
```

Try to navigate between the pages and see the results and see the results.

### 4.4) Create an edit button

- In the **MovieCardComponent** class create a method that returns the edit route:

```
getEditRoute() {
  return '/movie/' + this.index;
}
```

- In the **MovieCardComponent** template add the following link:

```
<a [routerLink]="getEditRoute()">edit</a>
```

Try to navigate between the pages and see the resulting url.

### 4.5) Lazy loading the details route

The angular router makes it pretty easy to lazy load content and access it only when the user hits a certain route. The content needs to be stored in a module

- Navigate to the src/app/pages/details folder, create a new module called **Details** using the Angular CLI:

```
$ ng g module Details
```

- Import the **DetailsComponent** and add it to the **declarations** array:

```
declarations: [ DetailsComponent ],
```

- Import the **RouterModule** and add it to the **exports** array (this is needed for ):

```
exports: [ RouterModule ]
```

- Declare the 2 sub routes (create and edit movie) and add them to the **imports** array:

```
RouterModule.forChild([{
     path: '',
     component: DetailsComponent,
   },
   {
     path: ':id',
     component: DetailsComponent,
   }])
```

The **forChild** method is used to register additional routes. Only call **RouterModule.forRoot** in the root **AppRoutingModule.**

- In the **AppModule**, remove the two routes that were declared in the **DetailsModule** and replace them with:

```
{
   path: 'movie',
   loadChildren: 'app/pages/details/details.module#DetailsModule',
},
```

Now reload the page and open the Web Inspector, on the network page. Clear all the loaded resources so you can better observe the new loaded resources. Navigate to the details page and you should see the resources needed for the **Details** section loading lazily.

# 5. Forms

Angular is one of the best frameworks when it comes to working with forms. If you have an application that's composed mainly of forms, then Angular might be your best choice. There are two different approaches to building forms: Template driven and Model driven (reactive). In the template approach most of the work regarding defining the form elements is done in the template (making heavy use of ngModel). On the other hand Model driven forms are created in code and are more advanced in terms of functionality. For example form changes can be treated as an observable.
More about forms here: https://angular.io/guide/user-input

For our example we'll use template driven forms and work on the create/edit movie functionality in the Details page.

### 5.1) Create the create/edit movie form

- In the **DetailsComponent** template add a **<form>** tag before the Cancel button added in the previous chapter
- Now we need to add the following form elements inside the <form>, corresponding to a movie properties:

```
<input type="text" placeholder="Insert title">
<input type="number" placeholder="Insert year">
<input type="text" placeholder="Insert duration">
<input type="text" placeholder="Insert genre">
<textarea placeholder="Insert plot" class="materialize-textarea"></textarea>
```

- Next we need to activate the form functionalities. Add **movieForm="ngForm"** to the <**form**> element. The **ngForm** directive will hold any child controls using **ngModel** and track their validity. Inside the class **movieForm** will be a property that we're going to add next.

```
<form movieForm="ngForm">
```

### 5.2) Binding the form to the model

- We now need to create the model behind the form. This will be a movie object (since that's what we're going to create/edit). Add a **movieForm** property to the **DetailsComponent** class and initialize it with empty values (don't forget to import Movie):

```
export class DetailsComponent implements OnInit {
  movieForm: Movie = new Movie('', null, '', '', '', '');
....
```

- Now we need to bind the form elements to the properties of the model. For this we're going to use the  For each of the **<input>** and the **<textarea>** add **[(ngModel)]** to set the two-way binding (eg. for the tile add **[(ngModel)]="movie.title"** )

If you check the web application now, you will see that you get an error message about ngModel. This is because ngModel is a directive declared on the **FormsModule**. In order to use the directive, you need to import the module. Another thing to note is that if we import this module in the DetailsModule (and not at the top most level) it will be lazy loaded, so we don't need to load it in other parts of the application.

- Import the **FormsModule** in the **DetailsModule**:

```
import { FormsModule } from '@angular/forms'
...
@NgModule({
  imports: [
    ...
    FormsModule
  ],
...
```

- In order to verify that the form works, add the following right before the end of the form:

```
{{movieForm | json}}
```

Now if you add text in the inputs you should see them reflected in the display of the movieForm property. The "|" character denotes a pipe. In this case we're using the json pipe which transforms an input text into its json format. The pipe is a special type of Angular component that is used in the template for data processing and display. You have a variety of pipes build in the framework but you can also define your own.

### 5.3) Submitting the form

- Add a submit button at the end of the form, but inside the <form> element:

```
<button class="btn" type="submit">
   Save
</button>
```

- Register the submit action in the form element:

```
<form movieForm="ngForm" (ngSubmit)="submitForm()">
```

- Create a **submitForm()** method in the **DetailsComponent** class and log the **movieForm** property.

### 5.4) Creating a new movie

- Before we use data from the form, we need to create a way to add movies in the movie collection. Add an **addMovie()** method in the **MoviesService**. The method receives a movie as a parameter and adds it to the movies collection:

```
addMovie(movie: Movie) {
    this.movies = [...this.movies, movie];
}
```

- Import and inject the service to the **DetailsComponent**

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

- Modify the **submitForm** method so that it calls the add movie method from the service:

```
submitForm() {
    this.movieService.addMovie(this.movieForm);
}
```

### 5.5) Navigate back to home after creation

- Add a reference to the Router service in **DetailsComponent**.  And don't forget to import it:

```
constructor(
    private movieService: MoviesService,
    private router: Router) { }
```

- In the submit method, add a navigation instruction immediately after the call to the movie service:

```
this.router.navigate(['/']);
```

### 5.6) [TASK] Adding an edit movie button

**STEPS:**

**A. MovieService**

1. Create an **getMovie(index)** method that return the movie at the specified index from the movies collection
2. Create an **updateMovie(index, newMovie)** method that modifies the movie at the certain index with the one received as an argument

**B. DetailsComponent**

0. For the next part, make sure that you navigate from an edit movie button (the url should start with http://localhost:4200/movie/ and have the selected movie id)
1. Create a **movieId** property that we'll use to store the movie id and decide whether we're in edit or create mode.
2. In order to get the id parameter form the route we need a reference to the **ActivatedRoute** service. Add this in your constructor as a private attribute (**private activatedRoute: ActivatedRoute**)

```
constructor(
   private movieService: MoviesService,
   private router: Router,
   private activatedRoute: ActivatedRoute) { }
```

3. In the **ngOnInit()** lifecycle hook store the **movieId** parameter suning the params observable on the **ActivatedRoute** service; also, if we have a valid id (eg. we're in edit mode) we should load the movie data and prefill the form

```
ngOnInit() {
  this.activatedRoute.params.subscribe(params => this.movieId = params['id']);
  if (this.movieId !== undefined) {
    this.movieForm = this.movieService.getMovie(this.movieId);
  }
}
```

4. In the **submitForm()** method make the same check to see if a movie if was defined. If we have a movieId, then call the **updateMovie** method on the **MovieService**; if not call the **addMovie** method; after any of them, keep the back navigation code.

# 6. Http

For our final chapter, we're going to integrate the built-in HTTP module to get IMDB data when we want to add a new movie. More on the HttpClient here: https://angular.io/guide/http

### 6.1) Integrating the http service

- First we need to import the module in the **AppModule**:

```
import { HttpClientModule } from '@angular/common/http';

...

@NgModule({
 ...
 imports: [
   ...
   HttpClientModule,
     ...
```

- We need to inject in the **MoviesService** a reference to the http service:

```
import { HttpClient } from '@angular/common/http';
....
constructor(private http: HttpClient) { }
```

- Next we will add a method to search a movie by it's IMDB id and call the .get() method on the http service.

```
searchImdb(id) {
  return this.http
    .get(`http://www.theimdbapi.org/api/movie?movie_id=${id}`)
}
```

We use the TheIMDbApi api to get movie data. The endpoints receives the imdb id for the movie and returns the movie details as stored in IMDb.

### 6.2) Parsing data received from the endpoint

- Data from the endpoint won't always come in the exact format that we need for our application. Thus we need to create parsing methods. In **Movie.ts** we need to create a static method to parse data and return a **Movie** object:

```
static parseMovie(apiObject) {
  return new Movie(
    apiObject.title,
    +apiObject.year,
    `${apiObject.length} min`,
    apiObject.genre.join(', '),
    apiObject.description,
    apiObject.poster.large
  );
}
```

We can now use this method in the service to modify the response we receive. We're done with the data logic, now we need to add the UI

**6.3) [TASK] Getting data from IMDB**

**A. DetailsComponent Class**

1. Add an **imdbId** property
2. Add a **imdbLookup()** method that calls the **searchImdb()** method on the **MoviesService**, providing it the **imdbId** parameter you created earlier.
3. In the callback for the subscribe parse the response using the static **parseMovie** method on the **Movie** and give the **movieForm** the value of the response

```
imdbLookup() {
  this.movieService.searchImdb(this.imdbId)
    .subscribe(movieResponse => {
      this.movieForm = Movie.parseMovie(movieResponse);
    });
}
```

**B. DetailsComponent Template**

1. After the movie form, add an input and bind it to the **imdbId** property using **ngModel**
2. Add a button to call the **searchImdb()** method
3. Check to see if everything works - lookup a movie on IMDb and get the movie code (eg. http://www.imdb.com/title/**tt3783958**/ -> the code is **tt3783958**), insert it in the input field and see if the info fills in

# Annex 1

**Styling the application.**

The application uses a small Material ui helper css library called Materialize CSS and a material icons library (for the rating stars). In order to use it in your project, just add a link in the head:

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
<link
href="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.100.2/css/materialize.min.c
ss" rel="stylesheet">
```

Please refer to the documentation pages in order to see how to use these:

- http://materializecss.com/
- https://google.github.io/material-design-icons/