Services

Services

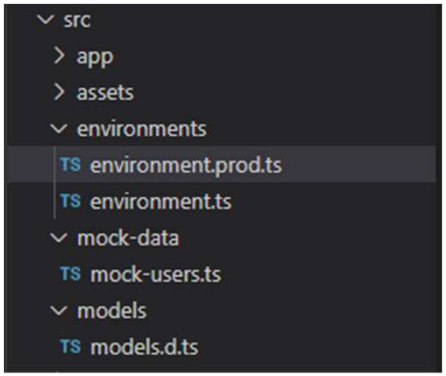
- Services are to serve our template with bussiness logic.
- Ideally components should only have template bindings.

- If parent should be only one to be infromed about event using @output is fine, but sometimes we need to inform more than one of our components about some event happens.

- Later during this course we learn how to achieve that with angular implementation of redux pattern.
- But for our training purposes we will use subjects

HTTP MAGIC - SETUP

- First of all we need some backend API.
- Luckily we had one here:
- Where to hold this URL?
  If we want to use different URL for two different envi
- it is good to put it in each environments.




```
export const environment = {
  production: true,
  apiUrl: 'https://workshop-wro.azurewebsites.net/api',
};
```

HTTP MAGIC - SETUP

- Next we need to comunicate somehow.
- ng g s services/books
- httpClient would help our service
- Let’s inject it then
- `constructor(private httpClient: HttpClient) {}`
- Add auto import ctrl + .
- httpClient provide to us very useful methods like get and post.
- We need our api url
- `apiUrl = environment.apiUrl;`

HTTP MAGIC - GET

- And GET method.

```
getBooks(): Observable<Book[]> {
    return this.httpClient.get<Book[]>(
      `${this.apiUrl}/books`
      );
  }
```

- It looks easy or hard – it depends, when you used to have to do that by plain methods it looks really easy. If this is your „first time” it could be a little bit overwhelming.

  We are strongly typing returning object to be type observable `Observable<Book[]>`
  It help us to ensure that we correctly write the body.

HTTP MAGIC - GET

- And GET method.

```
getBooks(): Observable<Book[]> {
    return this.httpClient.get<Book[]>(
      `${this.apiUrl}/books`
      );
  }
```

- Next we use our service get method `this.httpClient.get` and then again forcing type `<Book[]>` and passing our url into method

HTTP MAGIC - POST

```
getBooks(): Observable<Book[]> {
    return this.httpClient.get<Book[]>(
      `${this.apiUrl}/books`
      );
  }
```

```
addBook(book: Book): Observable<Book> {
    return this.httpClient.post<Book>(
      `${this.apiUrl}/books`,
      book
    );
  }
```

- The thing that was changed is that now we are passing object to the backend.
- Our method need to accept `book: Book` add we need to pass it then into httpClient post `book`

Inject our books service and use it

- `this.booksService.getBooks()`
- Remember to inject it first
- `private booksService: BooksService,`
- That is all we need to use.

- `this.books$ = this.booksService.getBooks();`

- `this.booksService.addBook()`

- `addDummyBook() {`
- `    this.booksService.addBook(this.dummyBook);`
- `  }`

Some filtering in flight

- Since we no longer store the data in component itself and only subscribing to observable in html, we dont have our filtering applied.

- Of course rxjs care for us providing methods to modify data in flight.

  We learn more about rxjs operators later, but we can use basic one to applied our filtering.

```
this.books$ = this.booksService.getBooks().pipe(
  map(books => {
    return books.filter(b => b.id > 2);
      })
    );
```

Srsly? Dummy? What about form?

- Yes, indeed, dummy book.
- We need to practice http.
- We are trying to get into one concept at a time.
- But of course there will be time for forms as well.

Exercise

- Delete a book by Id
- Get a book by Id and display some details component

Forms

- There are two ways of creating forms using angular.

- Template driven form (for easier cases, no need to react to some data changes, we have huge html where some logic goes)

- Reactive forms (for more complex scenarios, easy to add block with couple of controls on wish)

We will focus on reactive forms because it is applicable to both easy and complex cases and allow us to have more control over the flow

Reactive forms

- Reactive forms module

```
imports: [
    ...
    ReactiveFormsModule,
  ],
```

- Generate component with form (book-form)

Simple control

- `name = new FormControl('');`
- Ofc we need import – ctrl + .

- `name = new FormControl('');`
- Now we have it but need some floow

Add new book on demand

- `<button (click)="addbook()">add book</button>`
- `<button (click)="cancelAdding()">cancel</button>`
- `<app-book-form *ngIf="isInEditMode"></app-book-form>`
- We want to display form on demand (later in popup).

- `isInEditMode: boolean;`

Add new book on demand

```
addbook() {
    this.isInEditMode = true;
}

cancelAdding() {
    this.isInEditMode = false;
}
```

Change value from code

```
<button (click)="setDefault()">set default</button>
setDefault() {
    this.name.setValue('Dummy book');
  }
```

- As simple as that – of course we can use set value when reacting on other events or based on some @Input from the parent. If you think about editing right now you are correct.

- True FORM of FORM

True form

```
bookForm = new FormGroup({
    title: new FormControl(''),
    releaseDate: new FormControl(''),
    director: new FormControl(''),
    genres: new FormControl(''),
  });
```

...and Html

```
<form [formGroup]="movieForm">
  <input type="text" formControlName="title">
  <input type="text" formControlName="releaseDate">
  <input type="text" formControlName="director">
  <input type="text" formControlName="genres">
</form>
```

- Of course you can add labels.

Update to 9

- ng update is all you need (or should be all you need)

Some usefull resources and sources

- https://angular.io/ there is literally everything (sometimes condensed)
- https://material.angular.io/ great UI library working with hammer.js usefull when you dont feel comfortable with styles.
- https://angular.io/api?type=pipe list of pipes included in angular

- https://www.youtube.com/watch?v=jnp_ny4SOQE&feature=youtu.be&t=1320 ivy compiler which will be default in v9