Table of Contents

Start

Before we can start building Angular applications we have to prepare the development environment. This means installing the dependencies with NPM and start the application with the CLI to have a running application.

Tasks

1. Import the project into your IDE.
2. Run the following command from the command line, on the same level as `package.json`. Or use the IDE.

   ```
   npm install
   ```

   This will download the Angular framework and all of the dependencies.
3. Start the application server and TypeScript-transpiler in a terminal or from your IDE:

   ```
   npm start
   ```

4. Run the command below in a new command window to open the assignments and have click-able urls.

   ```
   npm run assignments:run
   ```

FAQ

```
npm install
```

If the install has not run properly due to network issues or environment setting, it is possible to copy them from someone else. If even that fails, you can ask the Trainer for a USB stick with a virtual machine which is pre-setup.

Optional

Webstorm & Angular

In the current version of Webstorm / Intellij it is possible to use `@angular/language-service`, for enhanced integration of Angular in the IDE> To use this feature, go to:

- `Preferences` -> Languages & Services -> `TypeScript`
- Tick `Use TypeScript Service` and click `Configure`
- Tick `Angular`.
- Do not tick `Compiler`, the CLI handles this for us.

Augury

Augury is a powerful debugging tool for Angular. If you use Chrome, this extension can give you an in depth view of the application.

CSS

You can find the documentation for the CSS library that is used in the project on https://mildrenben.github.io/surface/

RxDevtools

https://github.com/kwintenp/rx-devtools

Intro

In this short intro you will create a very small Angular app with some basic features. This way, you will get a feeling what Angular is all about. All features will be explained in more detail in later parts of the training.

Generating and running an app

1. In a new folder, type `ng new myFirstAngularApp`. This command generates a skeleton for your new angular app. This skeleton gives you a very good starting point for your app.
2. Change your directory to `cd myFirstAngularApp` where the skeleton was created and run `ng serve`. This command runs a simple http server that serves your newly created app.
3. Browse to `localhost:4200`. Congratulations. You've just created your first Angular app and now you see it running.

Implementing your own component

1. Open the folder `myFirstAngularApp` in your favourite IDE or text editor. Go to `src\app` and open `app.component.html`. This file contains the html that is currently displayed on the screen. Replace the contents of the file with this:

```
<button>Click me</button>

You clicked me 0 times.
```

2. Return to the browser. You will see that changes are immediately picked up, without you having to restart the application or refresh the browser. If you click the button, nothing happens yet. Let's add some interactivity!
3. Open `app.component.ts`. This TypeScript class defines the behaviour for the html you just saw. Add a property `clicked` and a function `onClick`:

```
export class AppComponent {

  clicked = 0;

  onClick() {
    this.clicked++;
    console.log(`You clicked ${this.clicked} times`);
  }
}
```

4. The next step is to tell Angular that `onClick` should be called when the button is clicked. We use a so called event handler for this:

```
<button (click)='onClick()'>
```

5. Open the browser console and click the button. The message `You clicked ... times` will be displayed.
6. To show the number in the template, we will use a so called interpolation. In the html, replace the number 0 with `{{ clicked }}`. This will show the value of the clicked property in the TypeScript class on the screen. Check that it works.

Modules and Components

An Angular-application consists of Modules with each their own tree of components. Remember the difference between certain 'types' of modules and the different style of components.

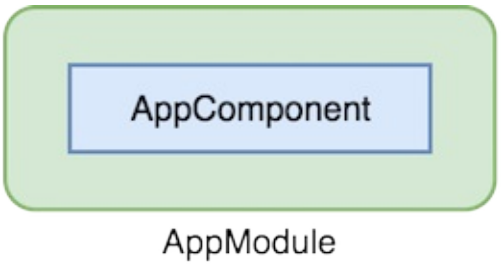
In this assignment we'll setup the Angular application from the start, and convert the current static page to an Angular application. We'll do this in 3 steps:

- Create a Root component & Root module
- Create a specialized component for the books
- Refactor the code to use a Feature Module

Techniques

- @Component
- @NgModule

Tree diagram



Root Component (AppComponent)

The component where the application starts is called the Root Component. Here, we will create and bootstrap this component.

1. Create a new file called `app.component.ts` in the folder `src/app/` and export the TypeScript class `AppComponent`.

   > **Best practice: File- and class names**
   >
   > It's Best Practice to give the class the same name as the file. By doing this, it's easier to find them when needed.

2. In `app.component.ts` : Import `@Component()` from `@angular/core` and put the decorator above the class `AppComponent` . As a parameter, this decorator gets a configuration object.

3. Put the `selector` property in the configuration object and give this the value `ibs-book-shop`

   > **Best practice: Application specific prefix**
   >
   > An application specific prefix ensures that there can be no collisions when combining the app with multiple generic libraries. Angular requires the component selector to be unique, and this prefix helps with keeping your component unique. In our case, we choose 'ibs', which stands very creatively for Ilionx Book Shop

4. Add the property `template` , this is what will be shown in the DOM.

5. Cut everything from `<!-- Start App Component -->` up to `<!-- End App Component -->` from `index.html` and as a string into the `template` property. Use `` tickbacks template strings from ES6 to have the template on multiple lines

6. Put the selector of `AppComponent` in `index.html` , between `<!-- Start App Component -->` and `<!-- End App Component -->`

Root Module (AppModule)

The module that is responsible for bootstrapping an Angular application is called the Root Module In `app.module.ts` we've set an `NgModule` ready to be used as Root Module.

1. Add the properties `declarations:[]` and `bootstrap: []` , and put `AppComponent` in both arrays.

   > **BrowserModule**
   >
   > `BrowserModule` add logic like error handling to an Angular application. Only the Root Module imports the `BrowserModule` .

2. Open main.ts and check out the documentation about what this does.

   > **main.ts**
   >
   > This is the file where an Angular application is started. It is configured in `.angular-cli.json` , and you can change this to any name you'd like.

Refresh

1. Check the browser, and make sure there are no errors in the console. If you have Augury installed, you can use it now. It's hidden in the tabs of the Developer Tools of Chrome

Business Components (BooksComponent)

In this assignment we will make add a Business Component to the application that will manage the books screen.

1. Create in the folder `books` the following files:
   - `books.component.ts`
   - `books.component.html`
2. Create a TypeScript class `BooksComponent` and decorate this with `@Component`, Set the `selector` property to `ibs-books` and `templateUrl` to `./books.component.html`

   > templateUrl's in `@Component` are relative and always have to be prefixed with './'.

3. We've created an interface in `app/books/models/book.interface` which contains the properties for a `Book` Import this interface into `BooksComponent`, and add a property called `books` that will hold a `Book[]`
4. Initialize the property, you can use the data on the bottom of the page. These are the books that we will show initially in the application. Later in the day we will be getting the data from a server.
5. Cut everything from `<!-- Start Books Component -->` to `<!-- End Books Component -->` from the template from `AppComponent`, and paste this in the template of `BooksComponent`.
6. Back in the template of `AppComponent`, add the selector of `BookComponent` in the place where you cut the template from.
7. To actually use the component in `AppComponent`, we need to declare `BooksComponent` in the `AppModule`.
8. Check in the browser if there are no error message. You will also see a difference in Augury, with the newly made components.

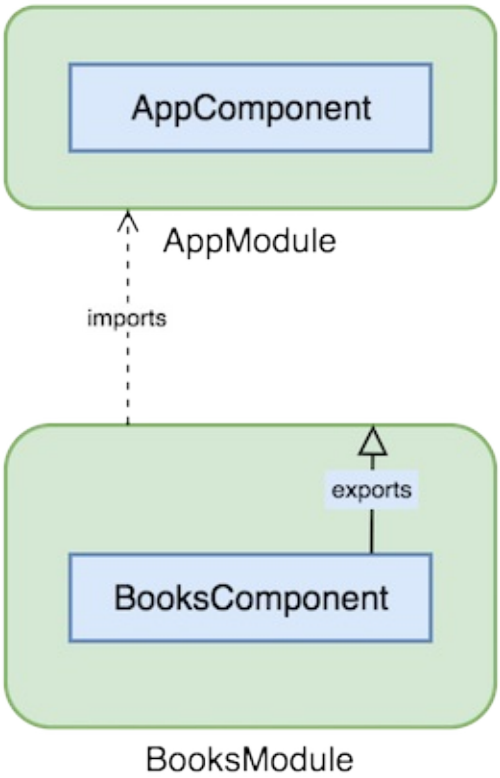
Data

```
[
  {
    "id": 100001,
    "title": "Application Design",
    "author": "O'Rly?",
    "genre": "programming",
    "img": "assets/app-head.jpg",
    "price": 14.95,
    "reserved": false
  },
  {
    "id": 100002,
    "title": "Remote Programming",
    "author": "O'Rly?",
    "genre": "misc",
    "img": "assets/coding.jpg",
    "price": 25.95,
    "reserved": true
  },
  {
    "id": 100003,
    "title": "Deadline programming",
    "author": "O'Rly?",
    "genre": "motivation",
    "img": "assets/coffee-code.jpg",
    "price": 14.95,
    "reserved": false
  }
```

```
    ]
```

Feature Module (BooksModule)

It's a good idea to separate the code into modules. This also adds the benefit to easily lazy load the module if that need comes. Here we'll make a new `BooksModule` which will contain everything related to books.



1. Create the file `books.module.ts` in the folder `books`
2. Export a new class called `BooksModule`, or any name following the style guide.
3. Decorate the class with `@NgModule`, which also gets a configuration object as a parameter.
4. Set the properties `declarations:[]` and `exports: []` in the configuration object, and add `BooksComponent` into both arrays.

   > **exports: []**
   >
   > This array holds the components which we want to be able to use outside the `BooksModule` Otherwise their selectors can only be used in components within `BooksModule`

5. Add `imports: []` to the configuration object, and put `CommonModule` into the array.

   > **CommonModule**
   >
   > `CommonModule` holds the common directives (ngFor, ngIf, ...) and pipse (json, currency, async, ..) of Angular. You will probably import the `CommonModule` into every Feature Module. This will not result in a bigger payload for each module. The Angular CLI (& Webpack) will put all commonly imported modules into its own file when creating a production build.

6. Check the browser to see an error in the console, which on of the suggestions are we following? Fix the error accordingly.
7. Remove `BooksComponent` from `AppModule`. Also delete the TypeScript import.

8. Add `BooksModule` to the `imports: []` of `AppModule`.

Optional

More modules

You can use the Angular CLI to easily create more modules and components. We can practice this by creating a few modules and some components. Try to setup the modules with Business and Presentation components, and when you need to share functionality over modules put the components in the correct module.

The goal of this assignment is practicing with the Angular CLI, so no templates are made available. However, when you are done with the other assignments you could implement one of these features for real.

Below are a few ideas that you can use to generate the necessary modules & components.

1. ShoppingCart The shopping cart will hold the items the user has selected, and needs an option to fill in user & buying details.

Here are a few examples of components you could add.

- a page where the content of the shopping cart is shown
- an order page for filling in user details & buying information
- a confirmation page

- CoffeeShop What is a book store without a coffee? This bookstore also has a coffee order page, with coffees, cups, special flavours of sugar and so on. Just like the Books, the user needs to be able to buy these and put them in the Shopping Cart. This means that the ShoppingCart should be able to show multiple types of items.
- BackOffice The managers of the Book store might want an overview of all the items that are currently available in the store, in a nice overview. It would also be nice to get a report of all the sales, and of course how much income has been generated.
- Request an item Not all books or coffees are available all the time. It would be nice if the user has some functionality where they can request books or coffees and get a notification when the item has arrived.

Template Syntax

The Template Syntax is used to render the data from the component in the template, react to events from the user and to modify the DOM structure.

In this assignment we will show the book titles in a list of buttons which will get the data from the Business Component `BooksComponent`. When a user clicks one of the buttons, this selected book will be shown next to the list of book titles.

We will make use of `Presentation components` to show the data and handle the user input. This way of thinking allows us a lot of freedom within the component itself like layout and styling, calculations or passing the data to new components. For the parent (usually a business) component, it does not matter what happens to the data as long as the API stays the same: An input for the list of books, and an output for the chosen book.

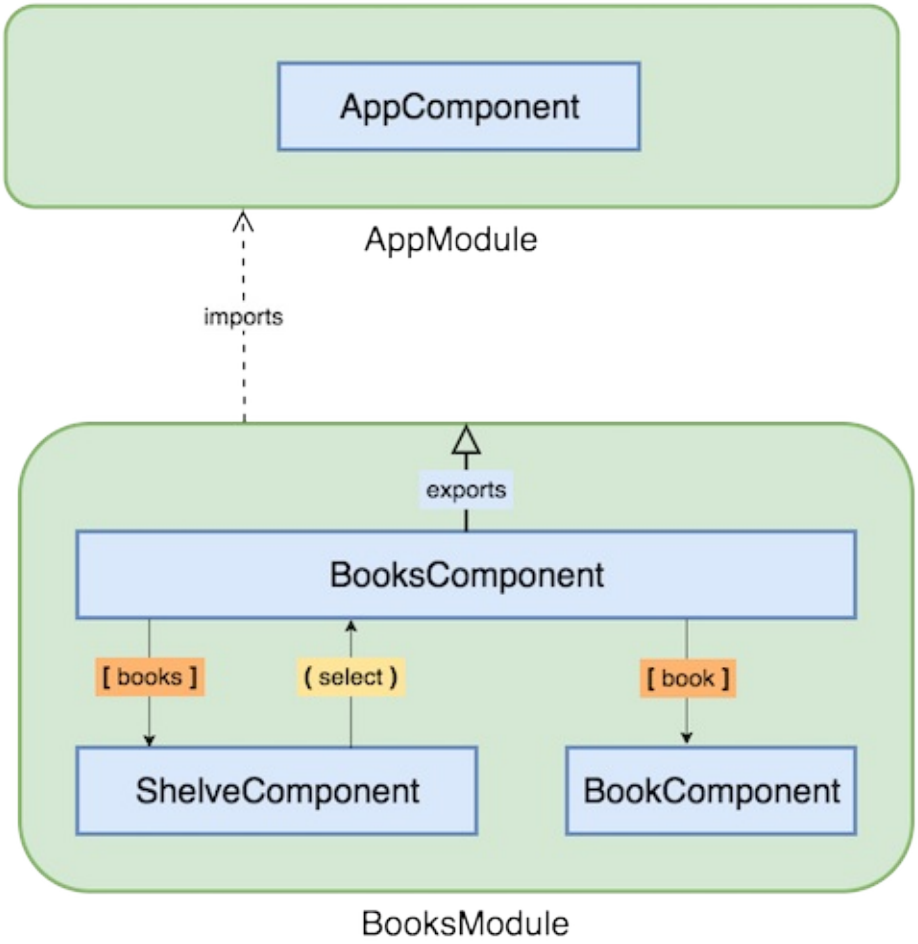
This process is done in 2 steps:

- Create a Presentation component for the list of books
- Create a Presentation component for the selected book

Techniques

- property binding
- event binding
- structural directives
- component communication

Tree diagram


AppComponent
AppModule
imports
exports
BooksComponent
[books]
(select)
[book]
ShelveComponent
BookComponent
BooksModule

shelve.component

The shelve will hold the books as a list of buttons, that the user can click to show the details.

1. Create a new folder named `shelve` in the folder `books`, with two files
   - `shelve.component.ts`
   - `shelve.component.html`
2. Create a new class called `ShelveComponent` and decorate this with @Component. Set the properties of `selector` and `templateUrl` to `ibs-shelve` and `./shelve.component.html` respectively.
3. Add `ShelveComponent` to the `declarations: []` of `BooksModule`

Binding: @Input()

1. Give `ShelveComponent` a property called `books` and decorate this with `@Input()`.
2. Cut the HTML between `<!-- Start ShelveComponent -->` and `<!-- End ShelveComponent -->` paste it into the template of `ShelveComponent`.
3. Back in the template of `BooksComponent`, set the selector of `ShelveComponent` between `<!-- Start ShelveComponent -->` and `<!-- End ShelveComponent -->`.
4. Show the titles of `books` in a `<button>` element with *ngFor and interpolation
5. To get the books from `BooksComponent` into `ShelveComponent`, add `[books]="books"` in the template of `BooksComponent` in the tag `<ibs-shelve>`. This is Property binding
6. Check the browser for any errors. If you have Augury installed, you can do some interesting things with the `@Input` decorator on `ShelveComponent`.

Events: @Output()

When a button is clicked, we want to show the selected book in `BooksComponent`. To get this done, we need to listen for an event from `ShelveComponent` in `BooksComponent`.

1. In `ShelveComponent`, create a new `@Output` property called `select` and instantiate it with an `EventEmitter<Book>`. The `EventEmitter` must be imported from `@angular/core`.
2. Create a method `onSelect` with a parameter `book:Book`, and call from this method the `@Output` property `select.emit()`.
3. To call `onSelect` from the template, we will listen for the `(click)` event of `<button>`. Since we have access to an instance of `book` in each element thanks to `*ngFor="let book of books"`, we can pass this from the template to `onSelect`.
4. In `BooksComponent`, add a property called `selectedBook`, and type it as `Book`.
5. Create a method on `BooksComponent` called `selectBook`. This will get a parameter called `book`. set the property `selectedBook` to value of the incoming parameter.
6. To connect the event from `ShelveComponent` to the method on `BooksComponent`, we need to listen for `(select)=""` from the element `<ibs-shelve>`. Set `selectBook($event)` between the quotes. `$event` is a pseudo-variable, which will tell Angular to get the arguments from the event and pass them trough.

BookComponent

The book component shows all the properties of a book.

1. Create a new folder named `book` in the folder `books`, with two files
   - `book.component.ts`
   - `book.component.html`
2. Create a new class in `book.component.ts` and decorate it with the @Component decorator. Set the `selector` and `templateUrl` properties.
3. Declare the component with the `BooksModule`.
4. Give the component an `@Input` property called `book`.
5. Show the the properties of `book` using `` ` and [] in the template of `` BooksComponent`.
6. Use the `selector` of `BookComponent` in `books.component.html`, and give it the `selectedBook` for the `@Input` book.
7. The first time `BooksComponent` is initialized the property `selectedBook` is `undefined`, and it won't be possible to get properties shown in the template. Check the console in the browser to see the error, and fix accordingly.
8. Set ngIf in the template of `BooksComponent` where `selectedBook` is read. This will not show the element until there is a value for `selectedBook`.
9. Show the price with the Currency Pipe and the Euro symbol.

Optional

ngIf;else

1. Show a message for the user when there is not yet a book selected.

Deselect

Add the feature to deselect a selected book. Where do you handle the user interaction and where the deselection of the current book? Keep the responsibilities of `Business Components` and `Presentation components` in mind while adding this feature.

You can use the following template to show a cross.

```
<p class="right deselect">X</p>
```

@Directive

When you need to add functionality or styling to a Component or an HTML-element, but do not need to add anything to the DOM, a Directive is what you need. Directives are TypeScript classes annotated with `@Directive` instead of `@Component`, and do not have a template.

In this assignment were are going to make a directive that adds a border around books that have been reserverd. This directive is going to be used in the `Shelve` and in the `Book` component. The border is going around the books in the shelve, and around the title in the book detail.

1. Generate a directive called `BookReserved` using the Angular CLI.

```
// if you have the CLI installed globally:
ng g d books/book-reserved

// otherwise, you can run it locally:
npx ng g d books/book-reserved
```

or follow the steps below

1. Create a new file called `book-reserved.directive.ts` in the `books` folder.
2. Export a class called `BookReservedDirective` and decorate it with the `@Directive` annotation.
3. The `@Directive` decorator requires the property `selector`, which is how we reference it in the template. Give it a value that complies with the StyleGuide
4. Declare the directive with the `BooksModule` so we can use the directive in the template of the components declared in this module.

The `BookReservedDirective` needs an input to know what book is reserved, and based on the value of that input it should add a border to the element.

1. Add a property called `reserved` and decorate it with the `@Input()` decorator.
2. Implement the `OnInit` interface from `@angular/core` and add the required method.
3. Create a method that will add the border, something like `addBorder` and call it from ngOnInit. Always try and keep the lifecycle methods clean from any logic.
4. Use the property binding `[]` -syntax to set the `reserved` property from the template of `ShelveComponent`.

To interact with the DOM from a Directive, we need a reference to the HTML-element to toggle the class. Angular provides the `template reference variable` mechanism for this, that you use by adding `#variableName` in the template to the element. Remember the `ngIf/else`, where we put a `template reference variable` on the element we want to show in the else-case.

We can use the `template reference variable` to pass in the HtmlElement into our directive using another `@Input` decorated property.

1. Add a new `@Input` decorated property to `BookReservedDirective` that will hold the HTML element. the type is HTMLElement.
2. In the template of `ShelveComponent`, add a `template reference variable` to the `button`-element.
3. Pass the `template reference variable` into `BookReservedDirective` using the `[]`-syntax.

To finish this task, we need to add the border to the element. The `HTMLElement` that we get from the `template reference variable` has an API to do this. We'll use the property `style.border` to set the border, by assigning a CSS value to it. The example below will add a 2 pixel wide, solid blue border to the element:

```
element.style.border = '2px solid #0000FF'
```

1. Add a border to the element based on the value of the `reserved` property.

Reusing the `BookReserved` directive.

We can reuse the `BookReserved` directive on the book detail to add an extra visual cue to show that the book is reserved.

1. Add the `BookReserved` directive to the template of `BooksComponent`. Make sure you put the directive on an `HTMLElement`, and not on a `Component`. This means we might need to make some changes to the HTML structure.

What happens when we click a reserved book first and then a different book? Why isn't the UI updated with the new value? This is because we only set the border during the `OnInit` lifecycle phase, which is called only once. To have the directive respond new values being passed in, we need to implement the `OnChanges` lifecycle.

1. Implement the `OnChanges` interface, and add the required method.
2. Move method call that adds the border from `OnInit` to `OnChanges`. In the lifecycle hook `ngOnChanges`, all the properties decorated with `@Input()` are updated with the new value.

Refactoring the `BookReserved` directive.

We have created an `@Input` called reserved to pass in the property from the book to the directive. It's possible to combine one `@Input` with the directive itself:

```
<!-- before -->
<button
 ibsBookReserved [reserved]="book.reserved" [element]="el" #el>
</button>

<!-- after -->
<button
 [ibsBookReserved]="book.reserved" [element]="el" #el>
</button>
```

1. Rename the `@Input() reserved` to the name of the directive.
2. Remove the extra code from the templates.

@ViewChild/@ViewChildren

Sometimes you want to get notified when HTML elements or child components become present in the DOM, and perform some action. Since actions are defined in the TypeScript class, we need to get a reference from the template into the component class. Angular provides the decorators `@ViewChild()` / `@ViewChildren()` to do this.

We are going to make a counter in the `shelve` component of how many books there are by counting the buttons in the template. This assignment is to introduce these decorators and the `AfterViewInit` lifecycle hook.

1. Create a template reference variable on the `<button>` element in the template of `ShelveComponent`.
2. Add a property to `shelve.component` to hold the button elements, and decorate it with `@ViewChildren()`

```
@ViewChildren('templateReferenceVariable')
elementQueryList: QueryList<ElementRef>
```

The properties decorated by `ViewChild()` / `@ViewChildren()` are filled by Angular in the `viewInit` lifecycle phase. When this phase has completed and the properties are filled with a reference, Angular calls the lifecycle hook `ngAfterViewInit` on the component.

1. Implement the interface `AfterViewInit` and add the required method.

In this method you can safely call properties decorated with `ViewChild()` / `@ViewChildren()` and be assured they are not undefined.

1. In this method, call `forEach()` on the property, which requires an arrow function as a parameter.

```
.forEach((element) => {
});
```

2. print the property `nativeElement` to the console. What else could you do with this property?

Dependency Injection

At the moment `BooksComponent` shows the books in the template from hardcoded data. In reality this data is likely to be stored on some server, which will be fetched by the application when necessary.

To get or send data, transform it or store is the responsibility of a **Service**. To be even more precise: if the operation has nothing to do with the template, it should be in a service.

Angular gives Dependency Injection out of the box te manage the creation of instances.

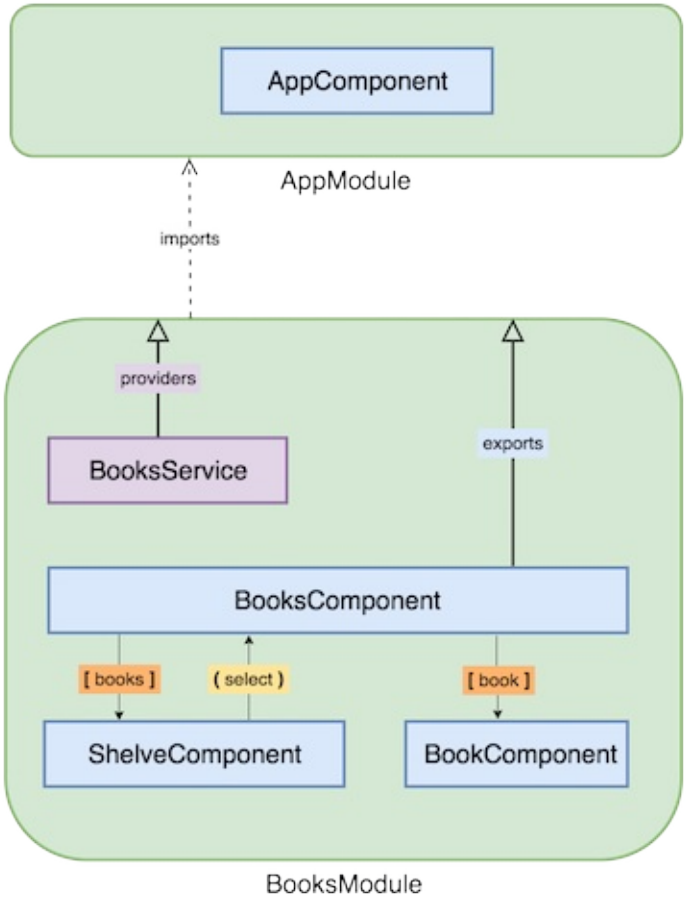
To add this functionality, we are going to need the following 2 steps:

- Create and provide a Service
- Make use of Angular's HttpClient to get the Books from the server

Techniques

- Provide services
- constructor injection
- `private` keyword
- Observables
- Observable operators

Tree diagram


AppComponent
AppModule
imports
providers
BooksService
exports
BooksComponent
[books]
(select)
[book]
ShelveComponent
BookComponent
BooksModule

books.service

The books service will be the communication layer between the component and the `http` service. This decoupling helps with error handling and share data operations for other components.

1. Create a new file called `books.service.ts` in the folder `app/books` and create a new class `BooksService`.
2. Add a method called `getBooks` with a return type `Book[]`, which will return the books. Cut and paste the data from `BooksComponent` into this method, we will replace this with a Http call later in the assignment.
3. Decorate the service with the `@Injectable({providedIn: 'root'})` decorator. Adding this decorator makes the class available for dependency injection in other services or components in the entire application.

Dependency Injection

1. In `BooksComponent`, inject `BooksService` by adding `booksService:BooksService` to the constructor of `BooksComponent`. Don't forget the `private` keyword, so the service is immediately bound to `this` of the `BookService`.
2. Add `implements OnInit` to the class definition if you haven't already.

   > **Best practice: OnInit & ngOnInit()**
   >
   > It's a best practice to handle data in component in any of the life-cycle events, and not the constructor. See life cycle for a detailed explanation.

3. Create the method `ngOnInit` in the component and call get the books from the `booksService`. For now, you can directly set the property `books` in the component to the result of the method.

HTTP

On the URL `http://localhost:3004/` there is a small NodeJS server (defined in `/src/server/server.js/`) with some endpoint to communicate with. We will replace the hardcoded list of books with an `http.get()` call to `http://localhost:3004/overview`, which will get the books.

1. In `BooksModule`, import the `HttpClientModule` from `'@angular/common/http'` put it in `imports: []`. Now the Angular `HttpClient` is available for this module and it's components.
2. Create a constructor in `BooksService` and inject `http: HttpClient`. Import `HttpClient` from `@angular/common/http` if the IDE does not do this for you. Use the `private` keyword to connect `http` to `this` of `BooksService`.
3. In the method `getBooks`, return `http.get<Book[]>()` with the parameter `http://localhost:3004/overview` instead of the books array.
4. Change the return type of the method to `Observable<Book[]>`. Import `Observable` from `'rxjs'` if the IDE does not do this for you.

subscribe

1. In `BooksComponent`, call `subscribe()` on `getBooks()`.
2. In the success handler of subscribe, set the result of `getBooks()` to `books`.

async pipe

Angular has an `AsyncPipe`, which will remove some boilerplate, and automatically handles subscriptions. This pipe will also `unsubscribe` if necessary when the component is destroyed.

In `BooksComponent` we need to work with an `Observable<Book[]>` instead of `Book[]`.

1. Change the type of the property `books: Book[]` to `books: Observable<Book[]>` in `BooksComponent`
2. Remove the `subscribe`, and directly set `getBooks()` to `books`. Note that at this point the Observable is not executed when this code runs. An Observable is executed when `subscribe` is called, which will be done by the `AsyncPipe` For now, it is a simple variable to be passed around as seen fit.
3. In the template of `BooksComponent`, put `| async` behind `books` in the `[property binding]` which passes the books to `ShelveComponent` This way, `ShelveComponent` still just gets a `Book[]`, and does not know that instead of hardcoded data, the data is now asynchronously fetched by an HTTP call. This shows how good encapsulation makes changes to applications very easy.

Optional

Dependency Injection

It's possible to set the title of the window in the browser from an Angular application with the Title-service.

1. Change the title of the window to the selected book. Which component should be responsible for this?

BookReservedDirective 2.0

If you have not yet finished the assignment @Directive, do that first. It is the basis for this assignment. This directive is not according to the StyleGuide, because it interacts with a DOM element directly. Angular provides a service that adds the possibility to interact with DOM elements, even when you are not in the browser. Angular made it a *Best Practice* to always use this service, so advanced features like migrating to server side rendering can be done without issues.

1. Inject the service `Renderer2` into the `BookReservedDirective`.
2. The `renderer` service has a method to set the style, called setStyle(). Refactor the code using this method.

Now that we know how to use injection, we don't need the separate `@Input` to get a reference to the element the directive is on. Angular can inject this element for us.

1. Inject the element of type `ElementRef` into the `BookReservedDirective`.
2. The `ElementRef` has a property `nativeElement`, which is the same HTML Element as we got from the template. The `nativeElement` is what the `renderer` needs to set the style.
3. Remove the `template reference variable` and the `[]`-syntax from the places where `BookReservedDirective` is used. This makes the `BookReservedDirective` easier to use, because the API surface has become smaller.

Observables

Error handling

You should always handle errors when working with Observable or Promises, at least with a message to the user to inform them what went wrong. Fot http calls, this is even more important, because the user does not see any errors if they are not handled.

1. Show an error message to the user when a http call fails. You can simulate this, by changing the URL in the `books.service`.

multiple `subscribe`s

It can happen that there are multiple components listening for the same http-call, or that a new component is rendered in the DOM which makes the entire Observable run again. We'll have a look at how this is done.

1. Copy the `ibs-shelve` tag in the template of `BooksComponent`, so that there are 2 instances listening for events from 1 Observable. If it doesn't work, make sure the `async`-pipe appears twice in the template.
2. In the Developer tools in the browser, open the tab `Network` and filter on `XHR` to only see AJAX calls. When you refresh, how many calls are made to the backend for the exact same data?
3. Try to get only 1 http call performed, when there are more components listening. Have a look at the share()-operator.

Transforming Observables.

In the multiple `subscribes` assignment we've made 2 components that show the exact same data. Make one of the components only show the **reserved** books. Do you make a custom component that filters the data itself, or do you let `BooksComponent` handle it?

You can make use of the `map()`-operator to transform the content of the data that goes trough the sequence, in this case `Book[]`. The `filter()` function on Array can be used to only get the books we want.

```
> ** Best practice: Why not filter()? **
>
> with filter() you can exclude certain events in the sequence from propagating. Since our even is the `Book[]`
, and we always want this value for the sequence.
```

Testing

Testing ensures your application will continue working as designed when altering or refactoring code. They can also ensure the correct functioning of Angular components. Test files are near the code in `.spec.ts` files.

Techniques

- Jasmine
- TestBed
- `configureTestingModule`
- `compileComponents`
- `TestBed.get('')`

> **Red bar**
>
> Make the test fail by expecting a length other than 2. Also check what will happen when the test has NOT been wrapped with `async()`

books.service test

The services are the most important to be tested since these contain the majority of business logic. Services often have dependencies on other services, like the HttpClient. These interactions will have to be mocked to create independent tests.

1. Create a new file `books.service.spec.ts` in the same folder as `books.service.ts`
2. Imports the `BooksService` from the project and `TestBed` from `@angular/core/testing`.
3. Within the first `describe`, start with `beforeEach(() => TestBed.configureTestingModule({}))`.

   > *configureTestingModule*
   >
   > This method configures a temporary testing module for running the test. This testing module, called `TestBed`, is configured identically to any NgModule. The {}-parameter contains the object definition typically used to describe a module. If the component or service being tested has specific module dependencies, they should be added to the `imports: []` of the object definition

4. Configure the `TestBed` with a `BooksService` (in providers) and its dependencies (in imports).

Mocking HttpClient

Since `BooksService` uses the `HttpClient`-service, we'll have to mock this. Mocking allows independence and control. We will use Angular's `HttpClientTestingModule` and `HttpTestingController` classes to simulate http calls.

- Testing Http Requests
- Add `HttpClientTestingModule` to the `TestBed`'s `imports: []`.
- Within the `describe()` add a new test using `it()`.
- Fetch a `BooksService` instance using TestBed.get(BooksService).
- Call the `getBooks()` method and `subscribe` to the result. The `subscribe`'s success handler will be containing the `expect`s that actually test the class. There's a call expected to `/overview` which we can check using the `HttpTestingController`.
- Obtain an `HttpTestingController` instance by calling `TestBed.get(HttpTestingController)` and assigning the result to a variable.
- Call `expectOne('localhost:3004/overview')` on `httpTestingController` and assign the `TestRequest` result to a variable. If this URL is not called, the test will fail.

  > *TestRequest* The TestRequest is an Angular class allowing you to check the sent request for header accuracy, for example. The TestRequest also provides functionality to return valid responses, simulate error messages and network errors.

- Now we'll have to return a response using `TestRequest`, so the `subscribe`'s success handling will be executed. We will use the `flush` method for this. The first parameter of this method is the response body. Return a valid `Book[]`, this matches the `getBooks()` return type.
- Add an `expect` checking if the received books match the ones we defined in the `Response` to the `subscribe` of `getBooks()`. For `Object`s use the `toEqual` matcher. This matcher will check if the properties and values are equal.
- Finally we'd like to see that all Http-calls have been processed and none are still open. Use the `verify()` method on the `HttpTestingController` instance for this. This will make the test fail if there are still any open calls. This could occur if for example the `retry()` operator has been used.

- Write a test for the error handler using the same `flush`-method, but this time provide a second parameter passing the `status` and `statusText`.

  > **Best Practice - error scenario**
  >
  > Always add an test for the error handling. At a minimum test the presence of an error message. This makes your test more robust and forces error handling to be used for every `Observable`.

- Run the tests by executing `ng test` on the command line.

shelve.component test

The shelve component contains `@Input` s and `@Output` s we will have to control from the parent.

1. Create a `shelve.component.spec.ts` file in the same folder as `shelve.component.ts`
2. Import the `ShelveComponent` .
3. Import `TestBed` from `'@angular/core/testing'` ;
4. Start with `beforeEach(() => TestBed.configureTestingModule({}))` and put `declarations: []` in the `ShelveComponent` .
5. In the same `beforeEach` method call `compileComponents()` of `TestBed` .
6. Create a new test in the `describe()` using `it()` . Try to fit the following sentence into the `describe` and `it` s: 'When the shelve component is given a list of books, the book titles are rendered in the template'

ComponentFixture

> **ComponentFixture properties**
>
> - `componentInstance` : allows interaction with the *class*.
> - `debugElement` : allows interaction with the *template*.

1. Call `TestBed.createComponent(ShelveComponent)` in the test. This returns a `ComponentFixture` .
2. Assign this return value to a variable `fixture: ComponentFixture<ShelveComponent>` .
3. Fill the `books` property of `fixture.componentInstance` with a new Observable, by importing `of` from `'rxjs/operators'` and returning two `Books` objects.

```
of([ { title: 'a', author: 'b' }, { title: 'c', author: 'd' } ]);
```

4. Call `fixture.detectChanges();` to start the change detection process from Angular which will call the lifecycle methods and render the value in the template.
5. Add an expect to the number of elements in the list. We've mocked the input with a list of 2 books.

```
expect(fixture.debugElement.queryAll(By.css('button')).length).toBe(2);
```

6. Run the tests by executing `ng test` on the command line.

Optional

Reactive Forms

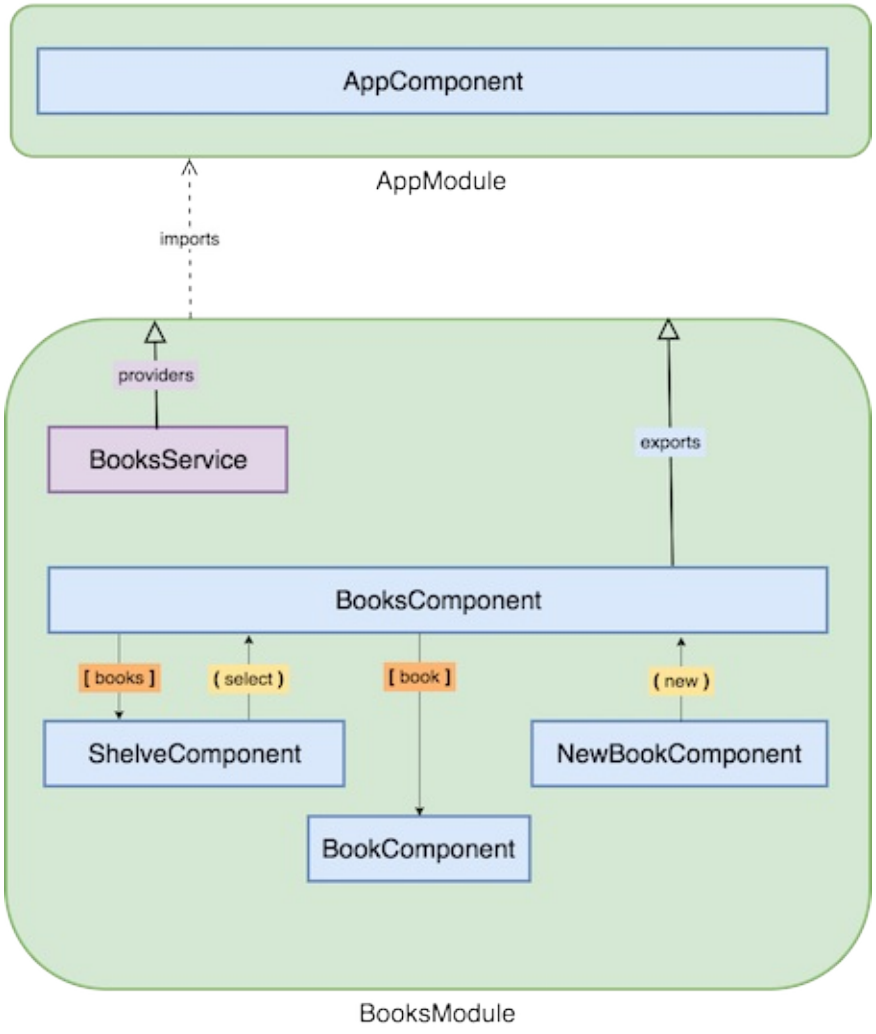
Almost every application has a form in which users can enter data. Such a form usually has validation for the inputs, and tells the user when a field has a wrong value. The form is then sent to the backend for further validation of business rules, and eventually saved.

In this assignment, we will make a form that will store a new book on the server.

Techniques

- FormGroup, FormControl
- `(ngSubmit)`
- [formGroup], formControlName
- validation

Tree diagram



ReactiveFormsModule

1. Add `ReactiveFormsModule` to the `imports` of `BooksModule`.

Component

1. Create a new folder named `new-book` in `books` with the following files:
   - `new-book.component.ts`
   - `new-book.component.html` Use the template below for `new-book.component.html`.
2. Create a new `@Component` class inside `new-book.component.ts` and give it the following properties:
   - `selector: 'ibs-new-book'`
   - `templateUrl: './new-book.component.html'`
3. Register `NewBookComponent` with `BooksModule` by putting it in the `declarations: []` property.

   > **angular-cli**
   >
   > Or, you can use the CLI to generate the folder and files and link it automatically with the module:
   >
   > - `ng generate component books/new-book -m books`
   > - `ng g c books/new-book -m books`
4. Put the `selector` of `NewBookComponent` in the template of `BooksComponent` between `<!-- Start NewBookComponent -->` and `<!-- End NewBookComponent -->`.

FormGroup, FormControl

1. Create a new property on `NewBookComponent` called `bookForm: FormGroup;`. Import `FormGroup` from `@angular/forms` if the IDE does not do this for you.
2. In the life-cycle hook `ngOnInit`, instantiate `bookForm` with a new `FormGroup`. As a parameter, a `FormGroup` gets a configuration object with a `"key":FormControl` structure.
3. Add the `Book` properties as keys in the `FormGroup`, and as value create a new `FormControl`.

   ```
   { title: new FormControl() }
   ```

   - title
   - author
   - genre
   - reserved
4. To fill the genres in the fill, you can use the data below. This is a new property on `NewBookComponent`.

   ```
   bookGenres: string[] =  [
      'drama',
      'thriller',
      'crime',
      'fantasy'
   ];
   ```

Submit

1. Create a new method on the `NewBookComponent` which will handle the sending of the form. Give this method a fitting name.

2. For now, just print the `.value` of the form to the console. Later on we will send it to the backend.
3. To catch the submit-event of the form, put `(ngSubmit)` on `<form>` and connect it to the method you've created.
4. Try it out in the browser, also look what the value of the form is when you have not touched it or when you've cleared the fields.

Templates

form

```
<div class="card">
  <h3> New book </h3>

  <form novalidate>
    <div class="g--12">
      <div class="container">
        <label for="title" class="g--4 no-margin"> Title </label>
        <input type="text" id="title" class="g--8 no-margin">
      </div>
      <!-- error message -->
    </div>

    <div class="g--12">
      <div class="container">

        <label for="author" class="g--4 no-margin"> Author </label>
        <input type="text" id="author" class="g--8 no-margin">
      </div>
      <!-- error message -->
    </div>

    <div class="g--12 container no-margin">
      <label for="genre" class="g--4"> Genre </label>
      <select id="genre" class="g--8">
        <option>drama</option>
        <option>thriller</option>
        <option>crime</option>
        <option>fantasy</option>
      </select>
    </div>

    <div class="g--12 container no-margin">
      <label for="reserved" class="g--4"> Reserved </label>
      <input type="checkbox" id="reserved"/>
    </div>

    <button type="submit" class="btn--raised g--6 m--6 font-rem"> Add </button>
  </form>
</div>
```

Template

[formGroup], formControlName

1. Bind the property `bookForm` to `<form>` by using `[formGroup]`.
2. Connect the `FormGroup`-properties `title`, `author`, `genre` en `reserved` of `bookForm` to `<input>`-fields with `formControlName`.
3. To fill the `select` with `<option>`s for the `genres`, use the `*ngFor` directive. Start with a no value `<option>` with the text `Pick a genre`.

> **[value] & [ngValue]**
>
> For setting the value property on select you might want to use an Object instead of a string for the value of the option. But, when using []-binding on HTML Elements, you always bind a `string` value to the property. This means objects will be set as `[Object]object`. Solution: [value] only works for string values. Is useful for simple data selections, like days, months or years. [ngValue] works with Objects and Arrays, and is necessary if you want to get a more complex result from the select.

html

`html`

Validations

The fields `author` and `title` need to be required, and luckily most books have those properties.

1. To make the fields required, the `FormControl` objects need a `Validator`. You can pass in an array of `Validator` s as a second parameter of `FormControl`. the first parameter is the initial value for the form, which can be `''` for an empty field.
2. Show a message for every field which is not valid. Put the messages just below `<!-- error message -->`
3. Make the error messages in the template visible when a field is invalid and touched. Use a `*structuralDirective` to optionally show the element. To get the state of any field, you can use the following syntax in the template.

```
    bookForm.get('title').hasError('required')
    bookForm.get('title').invalid
    bookForm.get('title').dirty
    bookForm.get('title').touched
    bookForm.get('title').pristine
```

4. As you might have noticed, there is a lot of duplication by getting the field from the form every time. This can be made more DRY by defining a `getter` on `NewBookComponent`.

   new-book.component.ts

```
get title() {
  return this.bookForm.get('title');
}
```

   new-book.component.html

```
<span *ngIf="title.invalid"></span>
```

error message

```
<span class="g--8 m--4 color--alizarin"></span>
```

Optional

- Form POST
- Template trigger
- Bi-directional service
- price
- ngClass
- delete
- FormBuilder
- Nested FormGroup
- FormArray
- listeners
- Edit existing books
- custom validator

Form POST

To save the book on the server, we have to make an HTTP POST call to the backend with the new book. The book is validated on the server side for the presence of an author and a title, the other fields are optional. If the book is stored successfully, you will get the stored books with an ID as a response. Since communicating with the backend is the job for a service, we are going to add the POST to the `BookService`, and use that from `NewBookComponent`.

1. Create a `store` method on the `BooksService`. Implement this method yourself, or use the template below. Give it the correct return type.
2. Inject the `BooksService` in the constructor of `NewBookComponent`.
3. In your method which handles the submit, check if the form is `valid` and emit the book to the parent if this check passes.
4. `subscribe` to the `Observable` from the `BookService`, to execute the sequence and do the POST call.
5. In the success-handler, reset the `bookForm` so all values are cleared and the state is reset if the response is successful.

Error handling

The book server checks if the book has already been stored, by checking the title & author. If the book is already in the shop, it will return an error with code `422` and a message.

1. Show the message from the server in the UI when a book is added with the same title and author. Try to keep the `HttpErrorResponse` class scoped in the service, this means that the component won't have knowledge of this class. To rethrow the error from the service so it will end up in the error handler of the component, you need the `catchError` & `throwError` operators. ```javascript import {catchError} from 'rxjs/operators'; import {throwError} from 'rxjs';

http.post() .pipe(catchError(error: HttpErrorResponse) => { // do some checking on the type of error. const customError = ... return throwError(customError); })

```
##### Updating the shelve
There are several ways to update the list of books when a book is added or deleted in the backend.

##### Trigger the AsyncPipe
If you have used the AsyncPipe, you can reassign the variable used in the template with a new call to `getBooks
`.
```

```
The AsyncPipe will pick up the change, and execute the observable.

##### Bi directional service
With a [Bi-directional service](https://angular.io/guide/component-interaction#parent-and-children-communicate-via-a-service) you manage the observable yourself, instead of directly listening for the http calls.
This is a very powerful mechanism, but requires you to unsubscribe from the observable yourself. This requires some rebuilding and rethinking of the application.

#### price

1. Add an input field for `price`.

```javascript
<div class="g--12">
  <div class="container">

    <label for="price" class="g--4 no-margin"> Price </label>
    <input type="text" class="g--8 no-margin" id="price">
  </div>
  <!-- error message -->
</div>
```

1. The price has to conform to the format ###(.##), and no book is more expensive then 999.99. Validate the input for these cases and show an error message for the user. You can use the following regular expression if necessary. `^\d{1,3}([\.]\d{2})?$`

ngClass

Give the input fields a visual indication when they are invalid. The class `field-invalid` can be put on `<input>` to achieve this.

```
[ngClass]="{'className': boolVal}"
or
[class.CLASSNAME]="boolVal"
```

delete

Now that we can add books, it would also be nice to be able to remove a book. There is an endpoint available on `/book/:id` which handles http `DELETE` calls to delete a book from the server. The placeholder `:id` is meant for the ID of the book. This endpoint gives you an empty response with status code 204 when the deletion was successful.

When the book, with id 100004 for example, is not found, you will get the a 404 response with the following body

```
{ detail: `Book: 100004 could not be found`, deleted: false }
```

1. Create a method in `BooksService` that will execute the delete.
2. Add a button to the application which will eventually call the delete method in `BooksService`. Remember the responsibilities of the different components, and put the code in the right places.

   ```
   <button class="g--12 btn--red color--white">Delete</button>
   ```

3. For now, you have to refresh to see the deleted book be removed from the screen. In a following optional assignment we will try to update the view automatically.
4. Show a message on screen if something went wrong with deleting the book. Try to delete the same book twice, to get the error message from the backend.
   ```

FormBuilder

Angular provides a FormBuilder service to remove some of the boilerplate code required with `FormGroup` / `FormControl` classes.

1. Rebuild the form in `NewBookComponent` to use the `FormBuilder` service.

Nested FormGroup

It's also possible to nest FormGroups inside other `FormGroup` s. In our book example, you might want to store some meta information about a book, like the date it was added to the store or the ISBN identifier. With a nested `FormGroup` , you can combine the date and ISBN under a group. Currently, when you get the `.value` of the `bookForm` , you'll get a response like:

```
{
  "title": "Application Design",
  "author": "O'rly",
  ...
}
```

With nested `FormGroup` s, the `.value` of `bookForm` could look like this:

```
{
  "title": "Application Design",
  "author": "O'rly",
  "meta": {
    "date": "1970-01-01",
    "ISBN": "1234"
  }
}
```

1. Add two new fields to the form in a nested `FormGroup` .

FormArray

The FormArray class makes is possible to create dynamic forms, where you can add or remove fields or `FormGroups` from the form. This is useful when you want to have a domain object with a property that can have multiple values. In our book-example, this could be the `genre` -field. There are many books that do not fit into one genre, but need te be categorized multiple.

1. Use FormArray to dynamically add multiple genres to a new book.

listeners

With `.valueChanges` and `.statusChanges` , which are properties of `FormGroup` and `FormControl` , you can listen for any new value or when the validation status of the field changes. By reacting to input changes it's easier te make something like an autocomplete field.

1. Listen for changes on the `title` -field using `.valueChanges` , and make a GET request on `http://localhost:3004/book/exists/:title` to see if the title already exists. The endpoint responds with an empty list when there are no matches, or a filled list with all book where the start of the title matches exactly as entered.

Edit existing books

The `NewBookComponent` could be reused to also support editing of existing books. What changes would you need to make to get `NewBookComponent` reusable for existing data?

1. Add support to edit existing books to `NewBookComponent`

custom validator

The `Validator` s that are added to the `FormControl` s are just functions that return `null` when valid, or an error object when the field is invalid.

1. Write a custom validator for `author`, that checks if each word in the input is capitalized. Show an error to the user, telling them which word needs to be capitalized.

save template BooksService**

```
save(book:Book): Observable<boolean> {
    return this.http
      .post('http://localhost:3004/store', book);
  }
```

Routing

At the moment the book form and the book list/detail are on the same page, which makes it more complex than necessary. In practice, these components would have their own pages, their own urls on which they can be reached. Angular has a powerful Routing mechanism which makes this possible.

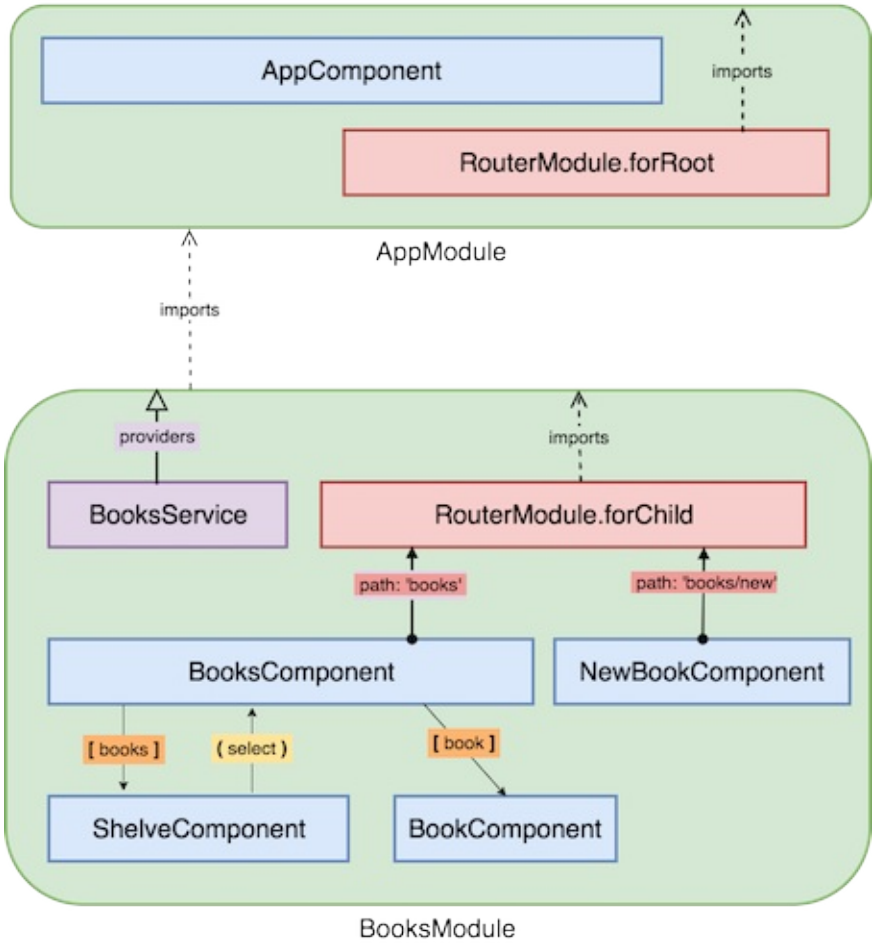
In this assignment we will add Routing to the application, and give `NewBookComponent` and `BooksComponent` their own urls. We will do this in 4 steps:

- Define the Routes for the BooksModules
- Define the Root Routes for the application
- Add links to get to the new URL
- Add a fallback route to catch unknown URL's

Techniques

- RouterModule
- `.forRoot`
- `.forChild`
- Route
- routerLink
- https://angular.io/tutorial/toh-pt5

Tree diagram


AppComponent
imports
RouterModule.forRoot
AppModule
imports
providers
imports
BooksService
RouterModule.forChild
path: 'books'
path: 'books/new'
BooksComponent
NewBookComponent
[books]
(select)
[book]
ShelveComponent
BookComponent
BooksModule

BooksRoutes

We will add Routing to the feature module `BooksModule` first. There are advantages to separate the routes per module, like adding lazy loading. Currently, `BooksComponent` shows everything; the list of books, the selected book and adding a new book to the application. This component will be replaced in `AppComponent` by a `<router-outlet>`, which will put the right component in place based on the URL.

books-routing.module.ts

It's a best practice to define routes in their own module. When additional routing features are added like guards or the routing gets big, it keeps the code cleaner because it's separated.

1. Create a new file that will hold the routing configuration, called `books-routing.module.ts`. naming guidelines: routing
2. Create a `const` called `BOOKS_ROUTES` and instantiate this with a new `Route[]`.

```
const BOOKS_ROUTES: Route[] = [];
```

3. Create a new `Route` in the array with the properties `path: 'books'` and `component: BooksComponent`.
4. Create a `Route` for the `NewBookComponent`. The path for will be `books/new`.
5. Export a variable `BooksRoutes: ModuleWithProviders` and instantiate this with `RouterModule.forChild(BOOKS_ROUTES)`

> **RouterModule.forChild()**
>
> The forChild() method adds routes to an existing Router instance, which is created by forRoot(). forChild() is meant for Feature Modules, and add the benefit to be easily lazy loaded.

```
export const BooksRoutes: ModuleWithProviders = RouterModule.forChild(BOOKS_ROUTES)
```

6. In `BooksModule`, import `BooksRoutes` and add it to `imports:[]`. This registers the routes with the `BooksModule`.
7. Since the form is now shown in it's own page, it can be removed from the template of `BooksComponent`.
8. To make it al look a bit nicer, replace the class `card g--10 m--1` with `card g--12` in the template of `NewBookComponent`.

app-routing.module.ts

1. Create a new file for the Root routing called `app-routing.module.ts`.
2. Create a variable called `APP_ROUTES` and instantiate this with a new `Route[]`.
3. Create a new `Route` definition with an empty `path` `''` which does a redirect to `books` and add this to `APP_ROUTES`. Remember to put `pathMatch` to `full` to this route is only executed when the path is *exactly* `'/'`.

```
<Route>{ path: '', pathMatch: 'full', redirectTo: '' }
```

> **redirectTo**
>
> Because we want to immediately redirect to books when we hit the landing page, we make use of the redirect property. A different scenario would be a landing page, like `HomeComponent`, where the user sees relevant information.

4. Export a variable `AppRoutes: ModuleWithProviders` and instantiate this with `RouterModule.forRoot(APP_ROUTES)`.

> **RouterModule.forRoot()**
>
> With forRoot() an instance of the Router service is created. It is *important* that forRoot is used once in the entire application. The Root Module should be the only place where forRoot() is used.

5. Import `AppRoutes` in `AppModule` and add this to the `imports` property of `NgModule`.
6. Replace in the template of `AppComponent` the tag `<ibs-books></ibs-books>` with `<router-outlet></router-outlet>`.

routerLink

To make the components reachable for the user form the interface, there need to be links in the application to the `Route`s. Angular provides the `routerLink` directive for this use.

1. Use the template below for the navigation bar in the application. You can put it below the placeholder `<!-- start NavigationBar -->` in the template of `AppComponent`.

```
<div class="g--12 tile bg--teal no-margin-vertical">
<div class="g--10 m--1 container no-margin-vertical">
 <div class="g--3 no-margin color--white">
   <a> Books </a>
 </div>
 <div class="g--3 no-margin color--white">
   <a> New </a>
 </div>
</div>
</div>
```

2. Add the directive `routerLink=""` to the `a`-tags.
3. Connect the `routerLink`s to the `paths` defined in `books-routing.module.ts`. Remember that the path's in `routerLink` directives are always relative.

> **paths starting with `/`**
>
> When a path in a `routerLink` starts with a `/`, then this path is interpreted as absolute. This means that angular starts with the Root Module to find the route.

4. Check in the browser if the right components are loaded when using the links in the navigation bar.

'404'

It's possible that a user enters a url that is not registered in the application. In this case, the user should get an error message and maybe an option to go to an existing page. To make this happen, we'll create a `NotFoundComponent`, declare it with the `RootModule` and add a route configuration to handle this scenario.

NotFoundComponent

1. Use the Angular CLI to generate a component, or create it manually if you like the practice.

```
npx ng g c not-found -m app
```

2. Show an error message to the user
3. add an link to navigate to a valid page. In this case we'll navigate to the url '/', aka the landing url.

Route configuration

To `catch` all the routes that are not registered in the Angular application, a special `path` is available: `**`. This path *needs to be the last entry* of the entire route configuration for the application.

1. Add the following route configuration to the `root routing` module:

```
{ path: '**', component: NotFoundComponent }
```

2. Make sure that the import of `AppRoutes` in `app.module` is *after* the import of the `BooksModule`.

> *try it out* Put the `AppRoutes` before the `BooksModule` in the imports. what happens if you try to go to `/books`? Have a look at the Augury plugin to see if you can figure out why.

Optional

Children

There are now 2 paths defined in the book routes, that both start with `/books`. This can be reduced so it reduces the noise and makes it easier to maintain.

Child routes

1. In `books-routing.module.ts`, create a new Route with just the properties `path` and `children`. It's not required to always route to a `component`.
2. Set the `path` property to `books`. This is the path which will be prefixed for all `child`-routes.
3. Move the `Route`s of `BooksComponent` and `NewBookComponent` to the `children`-array of the new `Route`.
4. Remove `books/` from the `path`'s of `BooksComponent` and `NewBookComponent`.
5. Check the browser to see if everything still works with the new configuration.

routerLinkActive

The `routerLinkActive` directive adds classes to elements based on the active route. They need to be defined on the same element as `routerLink`, or a parent element.

1. Add `routerLinkActive` to the divs surrounding the links in the navigation bar. The class `bg--river` can be used to style the link, when the router is active.
2. Check the browser to see what happens when you navigate to both components. Can you explain why both links are 'active' when the `NewBooksComponent` is on screen? You can make use of `routerLinkActiveOptions` to keep this from happening. Have a look at the documentation for a detailed guide.

Parameters

It's also possible to define routes with parameters. For practice we'll make a page that will show the details of 1 book. There is an endpoint available on `localhost:3004/book/:id` where you can get a book by their ID.

1. In `BooksService`, create a method to get a book by id. Use `getBooks` as an example on how to do a GET call.
2. If you have not yet made a separate component for a single book `book.component.ts`, do this now.

In order to reuse the Presentation Component `BookComponent`, we need a new Business Component that will read the URL for the book id, fetches the book from the `BookService` and pass it into `BookComponent`.

1. Create a new Component that will read the URL and fetches the book via the service. Try to think of a good name.
2. Inject `ActivatedRoute` and `BooksService` in the constructor of your new Component.
3. See presentation `5.Routing`, slide 18 to see the possible ways to interact with `ActivatedRoute`. Which one do you choose?
4. Use the id from the URL to call the method in `BooksService`.
5. Use the `async`-pipe to pass the book into `BookComponent`.

books-routing.module.ts

1. Create a new `Route` with a path parameter `/:id` and connect it to the newly made Component.

Navigate with Router

When someone clicks a book in the list, we are going to redirect them to the new detail page.

1. Remove `ibs-book` from the template `BooksComponent`, it will now be shown in it's own page.
2. Inject the `Router` -service in `BooksComponent`.
3. When the user selects a book, `selectBook` is called via the `(event)` binding.

Libraries

There are a lot of useful libraries available on NPM, which will speed up the development of your application. A few of the most popular types of library add UI components like tables or datepickers, or highly optimized functions like lodash.

In this assignment we will add a table to the application from the Angular Material library.

We will make a new module called 'backoffice', that will show the book inventory in a table and is available on its own url.

Techniques

- adding a library to the application
- importing functionality from a library
- using library documentation & examples

Angular Material Component Developer Kit (CDK)

The CDK provides un-opinionated components to be used by the application. The main goal of the CDK is add very general functionality, which can easily be customized by the developers.

- CDK table
- CDK table api

Angular CDK table

To use the CDK table in the application the following steps are needed . install packages & dependencies from npm . create the feature module . import the necessary packages modules into the feature module. . use the table following the documentation

Install packages from npm

manual

Most libraries have an installation guide that tells you what to install. Below are the commands that you need to execute

```
npm install --save @angular/material @angular/cdk
```

angular cli

you can use the following command to add libraries to the project using the CLI, and configure the app automatically.

```
npx ng add @angular/material
```

This will add the necessary dependency `@angular/cdk` automatically to `package.json` , and installs them. Furthermore, `app.module` will be updated with the `BrowserAnimationsModule` added to the imports. This is required for `@angular/material` to work.

Create the feature module

. Generate backoffice module and import it into the root module

```
ng g m backoffice
```

. Generate inventory component

```
ng g c backoffice/inventory
```

. create backoffice routing module with a route to the `InventoryComponent`

```
ng g m backoffice/backoffice-routing -m backoffice
```

. Add a `routerLink` to the navigation bar in `AppComponent` .

. Check if it works

Import library modules

An example for using the CdkTableModule is provided by google. In this assignment you will use the example as a starting point, and tailor it for the inventory component.

. import `CdkTableModule` into backoffice module

. copy the template into the inventory template

. copy the css into the inventory template

Using the library components

check out the component from the example.
Follow the steps below to rebuild it to work with our book service.

Supply template variables.

First we need to provide the variables that are required in the template:

- displayedColumns
- datasource

displayedColumns

create the property `displayedColumns` fill with book properties: id, title, author, price. in the template, update the value of `cdkColumnDef` for each column to match the `displayedColumns` .

datasource

the datasource is a layer between the table and the data provider, which is the bookservice in our case.

. create a new file called `books.datasource.ts` in the `backoffice` -folder. export a new class, and have it extend `DataSource` from `'@angular/cdk/collections'` .

. implement the constructor with a call to super(), and the methods connect/disconnect. connects returns an `Observable<Book[]>`

. add a parameter of type `BookService` to the constructor. return `getBooks()` in the `connect()` method.

> **Why not provide the Datasource as a service?**
>
> the datasource is not a service, since we might want multiple instances of it and it helps with testability of the class. However, this means we do not have injection available and have to provide any constructor parameters by hand.

. Add the property `datasource` to `InventoryComponent` .

. inject the bookService in the constructor of `InventoryComponent`

. in ngOnInit, create a new BooksDatasource instance and assign it to the dataSource property.

Styling

The CDK table has no styling by default, and what we have in the css is what gives it the current look. Angular Material provides a table with more styling and features, but it's built upon the CDK table.

. change the cdk table to the material table.

1. import `BrowserAnimationsModule` into the `app.module` . This is required for the animations to work.
2. import `MatTableModule` from `@angular/material` into the feature module.
3. change all `cdk` in the inventory template to `mat` , according to documentation
4. you can now remove most of the CSS in inventory.component.css and the specific CSS classes in your html, since the material table automatically uses material design CSS.

Material design

The material design table offers a lot of features you can use in the inventory component. And the material design library offers also other components that you can use in your own applications.

BrowserAnimationModule

1. Add the `BrowserAnimationModule` to the `AppModule`. You are going to need it later, because some of @angular/material's components have built in animations.

Material table features

Add extra features to your table. Use the documentation to find out how.

1. Make the table sortable by clicking on a table header.
2. Add pagination to the table.

Material design components

Have a look at all the components that are available in the material design library, and see how you can use them in our application. For example:

1. Use Material Design buttons instead of the buttons we use now.
2. Use Material Design form controls in the form for creating new books.
3. Import a theme to customize the look and feel of the application.
4. Have a look at layout, navigation, popups and progress spinners and use your imagination to make the application even more beautiful.

Optionals

Lazy loading

The size of the application has gotten bigger by adding functionality and imported that into the `RootModule`. By splitting the application by URL and lazily load the modules, you can improve the initial load time by many factors.

. Add lazy loading for the books and backoffice modules. Follow the documentation

> *things to note* Injecting bookservice works at the moment without providing it on the backoffice module. Why is this?

When you've added lazy loading to both books and backoffice modules, what happens now when you check it in the browser?

See limited provider scope for an explanation. more info:

- Prevent reimport of the CoreModule
- Bad practices

. Restructure the application with a `CoreModule`. What other module that is imported into `BooksModule` can be moved to the `CoreModule`?

RxJS

RxJS is a Javascript library that implements the Reactive Extensions pattern. Other languages like Java, C#, Python and others also have libraries that implement this pattern, that are more or less similar to RxJS. A key difference is in the naming of some of the operators, but the underlying principle is the sam.e

When you get a grip with RxJS, the other implementations are easy to pick up.

Install packages from npm

Angular requires RxJS, because it is used to communicate internally and a lot of public API parts use RxJS as the facade to which you can connect. No further installation is needed.

Best practices

RxJS best practices

Operators & .pipe()

One of the strengths of RxJS are the 120+ operators, which take away a lot of boilerplate to handle the common tasks. A few of the most common are:

- map()
- filter()
- tap()

Techniques

- Operators
- Combining operators

Information

Operators

Operators are methods that do 1 thing to the Observable, or the value that is going through the sequence. These operators can be split into 2 types:

- Creation operators
- Pipeable operators

Creation operators

Creation operators create a new Observable from non-observable values.

- of
- ErrorObservable.create
- fromPromise
- fromEvent

Pipeable operators

Pipeable operators are passed into the `pipe()` function, and can change the value or direct the observable sequence.

- map
- filter
- concatmap / mergeMap / forkJoin
- share

concatmap / mergeMap / forkJoin

https://medium.com/@tomastrajan/practical-rxjs-in-the-wild-requests-with-concatmap-vs-mergemap-vs-forkjoin-11e5b2efe293

share

https://medium.com/@_achou/rxswift-share-vs-replay-vs-sharereplay-bea99ac42168

Route Parameter Observable.

It's also possible to define routes with parameters. For practice we'll make a page that will show the details of 1 book. There is an endpoint available on `localhost:3004/book/:id` where you can get a book by their ID.

1. In `BooksService`, create a method to get a book by id. Use `getBooks` as an example on how to do a GET call.
2. If you have not yet made a separate component for a single book `book.component.ts`, do this now.

In order to reuse the Presentation Component `BookComponent`, we need a new Business Component that will read the URL for the book id, fetches the book from the `BookService` and pass it into `BookComponent`.

1. Create a new Component that will read the URL and fetches the book via the service. Try to think of a good name.
2. Inject `ActivatedRoute` and `BooksService` in the constructor of your new Component.
3. See presentation `5.Routing`, slide 18 to see the possible ways to interact with `ActivatedRoute`. Which one do you choose?
4. Use the id from the URL to call the method in `BooksService`.
5. Use the `async` -pipe to pass the book into `BookComponent`.

Drag 'n Drop

A drag and drop is a feature that can be very complex, but is made easier by using a few RxJS features.

We will make a feature where you can reorder the shelve by dragging and dropping the rows.

Note - Advanced assignment

Implementing this feature requires more than only RxJS, because you need interaction with the DOM and HTMLElement properties. For convenience, at the bottom of the page you can find the methods you can copy/paste and use for the non RxJS functionality. However, if you like the challenge you can implement it yourself.

Techniques

- Angular - Template reference variable & @ViewChildren
- RxJS - multiple operators and tie ins with Angular and DOM
- DOM - HTMLElement and style properties.

Steps

HTML & Styling

1. add reorder element to shelve LI <>
2. change button class g--12 to g--11 to make it look better

get reorder elements into typescript class

1. add #draggable to in html
2. Get a reference to the `#draggable` s into the TypeScript class by using `@ViewChildren`

```
@ViewChildren('draggable') bookButtons: QueryList<ElementRef>
```

Properties that are decorated with @ViewChildren, are available in the afterViewInit lifecycle method. But in our case, the buttons are only added to the view when the call to the backend has returned. Luckily, the `bookButtons` property provides an Observable for this: `.changes`. The `next` handler will be called any time an update to the DOM has happened that touches what you're listening for. In our case, when an element with the `#draggable` template variable is added or removed.

1. print each button to the console in the right life cycle hook.

```
this.bookButtons.changes
.subscribe((queryList: QueryList<ElementRef>) => {
 queryList.forEach((draggable: ElementRef) => console.log(draggable))
});
```

add drag and drop to each element

A way to think about Reactive programming is by creating a small story about what needs to happen: when the `mousedown event` is triggered, get the clientX & clientY from the `mousedown event` then ONLY LISTEN FOR `mousemove event` s. every mousemove should: ALSO not do the default, and TRANSFORM the mouse event to an object with the new position of the element. UNTIL the `mouseup event` has been triggered. ALSO the `mouseup event` should drop the element to the new position in the DOM

In this story there are 3 events defined: `mousedown` , `mousemove` and `mouseup` . These are the Observables we need to combine. The capitalized words in the story correspond to an Observable operator, in a sense.

implementation

The element we need to reorder is the `<li>` , but the element we click on is the `<span>` .

1. get a reference to the parent 'li' element using and assign it to a variable.
   `draggableSpan.nativeElement.closest('li');`
2. create 3 observables with 'fromEvent'
3. 'mousedown' on draggable -> the element we want to move (li)
4. 'mousemove' on document -> the position of the cursor on the document
5. 'mouseup' on document -> anywhere the user let's go off the button on the document
6. We are going to start with the 'mouseDown' observable, since this is where the user starts the event. use `.pipe` to add the `mergeMap` operator. This corresponds to the `ONLY LISTON FOR` part of the story.

   > mergeMap operator we use this operator because we want to merge the events from `mousedown` with `mousemove` events. other operators we could have chosen are `concatMap` and `switchMap` . see this link for an explanation.

7. in the `mergeMap` operator, pass in an 'arrow' function that returns the `mousemove` observable. This is all that is needed to connect the 2 observables. However, we need to use data from the `mousedown` and transform the data from `mousemove` to something we can use.
8. Use `({ clientX: startX, clientY: startY })` as a parameter to the arrow function in `mergeMap` . This will get the properties `clientX` and `clientY` from the `mousedown` -event, and put them in the variables `startX` and `startY` .

Now, what happens by default, is that the browser wants to select text when you hold the mousedown and move it. We want to prevent this, but we do *not* want to change the observable data. This is what we can achieve with the `tap` -operator, and `preventDefault()` from the event.

1. Add `pipe()` to the `mousemove` observable, and add the `tap` operator to handle the `ALSO` part in the story. the parameter for the arrow function is the `MouseMove` event, that has the method `preventDefault()`

Next, we need to transform the `MouseMove` event to a simple object that has the new position for the element we are moving.

1. add `map()` to the pipeline of `mousemove` (after `tap`). The parameter for the arrow function is the `MouseMove` event. use the following code to get the new location of the element we are dragging:

```
{
  left: e.clientX - startX,
  top: e.clientY - startY
}
`
```

Lastly, we want to stop listening when the user lets go of the mousebutton, which is represented by the `mouseup` -observable.

1. add `takeUntil` to the pipeline, and pass in the `mouseup` observable. This will stop the observable when the observable is triggered.

All these steps will have created 1 new Observable sequence, which will drag the element with the cursor. We still need to subscribe to this observable to execute it, and update the position of the element we are dragging.

1. Assign the sequence we just built to a new variable, `mousedrag`.

```
const mousedrag = mousedown.pipe(...);
```

2. subscribe to mousedrag, and in the `next` handler update the position of the element.

```
// draggable is the element we are moving. It should be the same variable we created in step 1.
draggable.style[ 'z-index' ] = '9999';        // set the z-index very high. This paints the element above a
ny other and will give it the floating effect.
draggable.style.position = 'relative';        // set the position property to relative, so we actually see
the element move with the cursor.
draggable.style.top = position.top + 'px';    // update the top position
draggable.style.left = position.left + 'px';  // update the left position.
```

To make the element change position in the list, we need to add a pipeline to the `mouseup` observable. This will have a side effect that the position of the element in the DOM is updated to where the mouse is.

1. Add `pipe()` to `mouseup`, and add `tap` to handle the side effect. you can use the code below, or try and write your own.

Helper methods

```
  /**
  * drop the element in the dom on the current cursor position.
  **/
  private drop(x: number, y: number, draggable: any) {
      const el = this.getDrop(x, y);
      const parent = el.closest('li');
      if (parent) {
        const position = this.placeBeforeOrAfter(parent, draggable);
        parent.insertAdjacentElement(position, draggable);
      }

      draggable.style = 'none';  // reset position
    }

    /**
    * Determines if the element should be placed before or after the current element.
    * This is based off the y value. If the current element is above the dragged element, it should be placed a
bove the current.
    * else below.
    **/
    private placeBeforeOrAfter(parent, draggable) {
      return  parent.getBoundingClientRect().top > draggable.getBoundingClientRect().top
        ? 'afterend'
        : 'beforebegin';
    }

    /**
    * Gets the 3rd element from any given x, y position.
    * in our application context, this will be the `li` where the user is hovering on.
    **/
    private getDrop(x: number, y: number) {
      return document.elementsFromPoint(x, y)[ 3 ];
    }
```

Optional

import syntax

Import anything that is exported from a TypeScript file.

```
import { Book } from '../path/to/component'
```

Component

A TypeScript class and a template that controls a part of the screen.

@Component

`@Component` is a Decorator, but works like a function, from `@angular/core` and has a configuration object as a parameter `{ }`.

The most commonly used are below, and a complete list can be found here.

```
@Component({
    selector                    // string; The name of the component used in the template
    template | templateUrl      // string; The template itself using tickbacks ``, or the url to the template.
If the moduleId is set, a relative path will suffice
    styles | stylesUrl          // array | string; The location for component specific CSS, or a reference to a
 css file
})
```

Lifecycle hooks

Lifecycle hooks are called by Angular during different sections of the application cycle

Angular calls the following hooks in this order:

```
    * ngOnChanges - called when any input or output binding changes
    * ngOnInit - after the first ngOnChanges
    * ngDoCheck - hook for custom Change Detection.
    * ngAfterContentInit - After the component has been initialised
    * ngAfterContentChecked - After every component content check.
    * ngAfterViewInit - After component view(s) have been initialised.
    * ngAfterViewChecked - After each component view(s) check.
    * ngOnDestroy - Right before a @Directive is being destroyed.
```

interpolation

Showing a property in the template with Angular's template syntax.

```
{{ componentProperty }}
```

Pipe

Change how the interpolated property is outputted in the template.

```
{{ | pipeName }}
```

event

Call a method from the template on the TypeScript class

```
<button (click)="handleClick($event)">Click!</button>
```

Structural directive

These directives changes the DOM structure and can be recognized by the `*`.

ngFor

Loop over an iterable and repeat the element that the directive is put upon.

Angular documentation

```
<div *ngFor="let item of items"></div>
```

ngIf

Conditionally show an element based on the truthy/falsy.

```
<div *ngIf="evalToBoolean"></div>
```

ngIf / else

Show a different element when the resolved property is false.

```
<div *ngIf="evalToBoolean; else #loading"></div>
<ng-template #loading><span>Loading</span></ng-template>
```

Elvis

The Elvis operator is a falsy-check which can be used in templates.

```
<span> {{ undefinedProperty.name }} </span>     // will give an error
```

```
<span> {{ undefinedProperty?.name }} </span>    // Do a falsy check, which will remove the error from the conso
le.
                                                 // The <span> element is rendered in the DOM
```

template reference variable

A reference to an element in the template. Can be given to the component.

```
<!-- #localVariable = HTML element -->
<div #localVariable (click)="clicked(localVariable)"></div>

<!-- #localVariable = NgControl element -->
<input ngControl="" #localVariable="ngForm">
```

$event

Pseudo variable that contains the value from the event.

All methods that are connected like `(click)="doAction($event)"` will automatically get the parameter `$event` filled by Angular This helps with encapsulation, because you don't need to know the name of the parameter from the event.

@ViewChild()

Decorator to get an element from the template into a template class. The decorated property is callable in ngAfterViewInit.

@ViewChildren()

Decorator to get a List of elements from the template into a template class. The decorated property is callable in ngAfterViewInit.

Service

A service in the Angular context is an injectable class which is responsible for doing the work. It has no UserInterface dependencies.

providers

With this you can provide an instance of a Service to the entire application. Make sure you only have the service defined in 1 array, to ensure they are Singleton's.

```
providers: [ MyService ]
```

AsyncPipe

With the `async` -pipe from Angular it's easier to manage simple Observables. This pipe also ensures that components are unregistered from the sequence when the component is destroyed.

operators

Observable operators must be imported from 'rxjs/operators' to use them.

Thanks to the Angular CLI this duplication will be removed when making a production build, so the payload will be as small as possible. Because of the size of number of operators which cover many use cases, it's better to import the ones you need for the application you are building.

Route

Couples an url to a component, or aggregate components under a single path.

```
Route:  {
   path?: string;
   component?: Type | string;
   redirectTo?: string;
   children?: Route[];
   pathMatch?: string;

   outlet?: string;
   canActivate?: any[];
   canActivateChild?: any[];
   canDeactivate?: any[];
   canLoad?: any[];
   data?: Data;
   resolve?: ResolveData;

   loadChildren?: string;
 }
```