Angular Training

What is Angular?

Angular is an application design framework and development platform for creating efficient and sophisticated single-page apps.

Angular Concepts

The architecture of an Angular application relies on certain fundamental concepts.
The basic building blocks of the Angular framework are Angular *components* that are organized into Angular *Modules* (*NgModules).*
An Angular app is defined by a set of *NgModules.*

Modules (@NgModule)

Every Angular app has a root module, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.
NgModules can import functionality from other NgModules, and allow their own functionality to be exported and used by other NgModules. For example, to use the router service in your app, you import the Router NgModule.

Components (@Component())

Every Angular application has at least one component, the root component that connects a component hierarchy with the page document object model (DOM). Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed.

Services (@Injectable())

For data or logic that isn't associated with a specific view, and that you want to share across components, you create a service class. A service class definition is immediately preceded by the @Injectable() decorator. The decorator provides the metadata that allows other providers to be injected as dependencies into your class.

Routing

In a single-page app, you change what the user sees by showing or hiding portions of the display that correspond to particular components. As users perform application tasks, they need to move between the different views that you have defined.

To handle the navigation from one view to the another, you use the *Angular Router*.

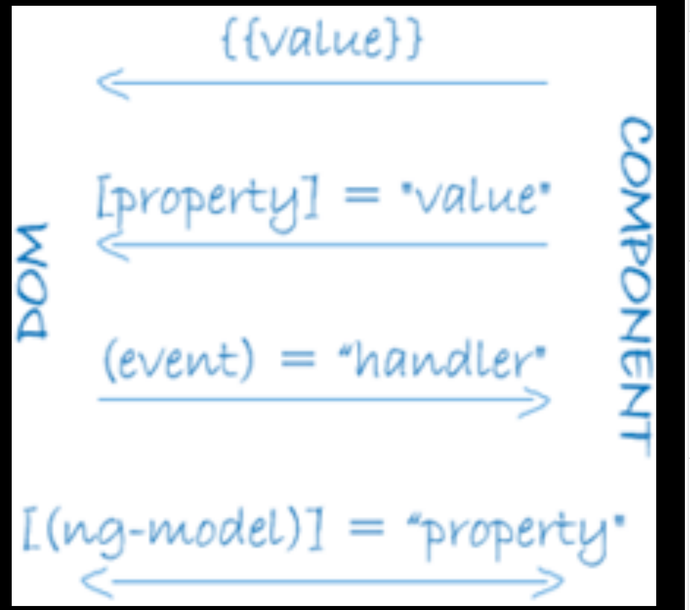
The *Router* enables navigation by interpreting a browser URL as an instruction to change the view.

Data Binding

Data-binding is a mechanism for coordinating what users see, specifically with application data values.

Angular provides many kinds of data-binding. Binding types can be grouped into three categories distinguished by the direction of data flow:

- From the *source-to-view*
- From *view-to-source*
- Two-way sequence: *view-to-source-to-view*



Type	Syntax	Category

Interpolation
Property
Attribute
Class
Style
	`{{expression}}`
	`[target]="expression"`
	`bind-target="expression"`
		One-way
		from data source
		to view target

Event	`(target)="statement"` `on-target="statement"`	One-way from view target to data source
Two-way	`[(target)]="expression"` `bindon-target="expression"`	Two-way

Binding types and targets

The **target of a data-binding** is something in the DOM.

Depending on the binding type, the target can be a property (element, component, or directive), an event (element, component, or directive), or sometimes an attribute name.

The following table summarizes the targets for the different binding types.

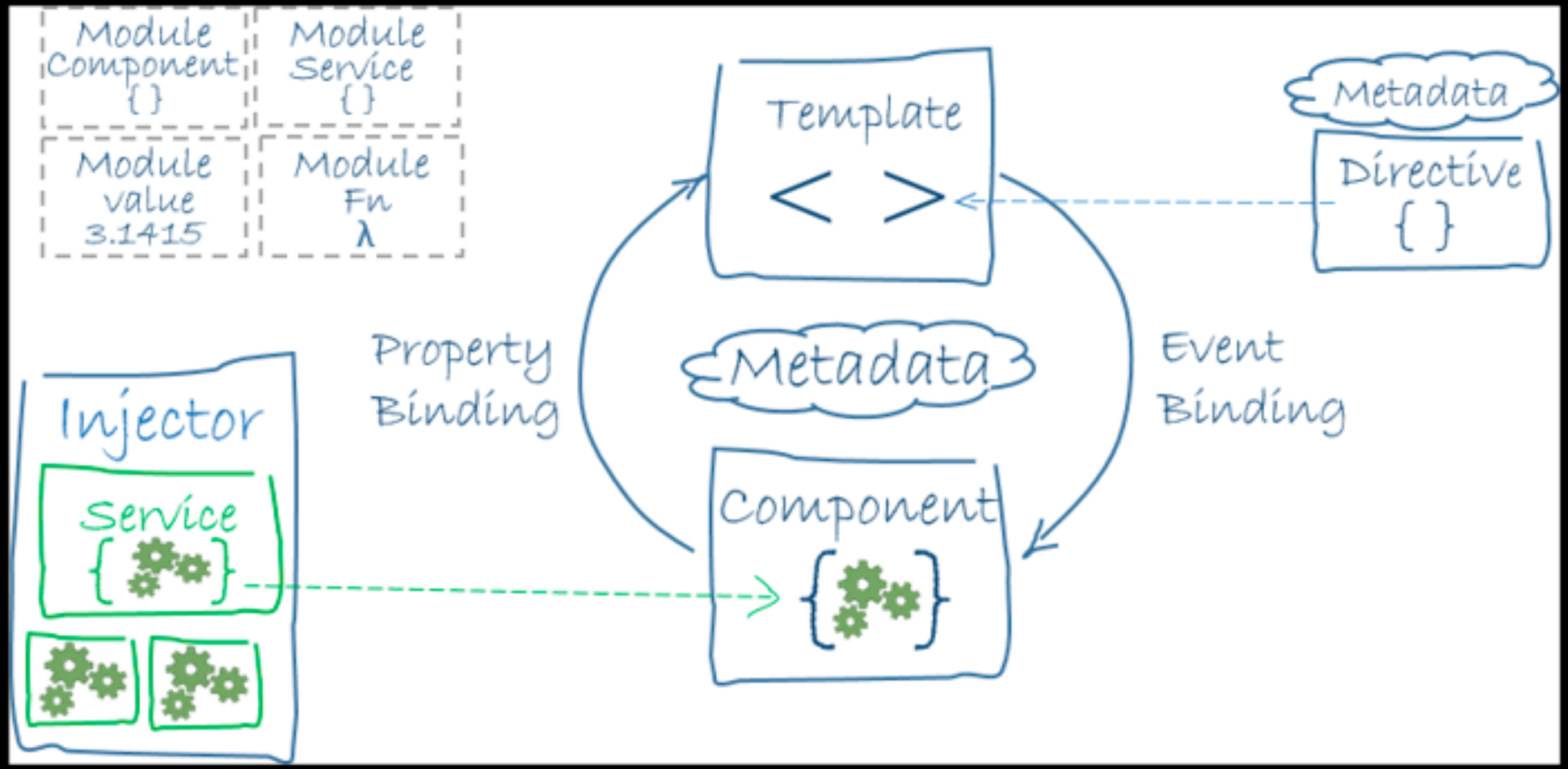
Type	Target	Examples

Property	Element property
	Component property
	Directive property
		`src`, `hero`, and `ngClass` in the following:
		`<img [src]="heroImageUrl">`
		`<app-hero-detail [hero]="currentHero"></app-hero-detail>`
		`<div [ngClass]="{'special': isSpecial}"></div>`

Event	Element event
	Component event
	Directive event
		`click`, `deleteRequest`, and `myClick` in the following:
		`<button (click)="onSave()">Save</button>`
		`<app-hero-detail (deleteRequest)="deleteHero()"></app-hero-detail>`
		`<div (myClick)="clicked=$event" clickable>click me</div>`

Two-way	Event and property	`<input [(ngModel)]="name">`
Attribute	Attribute (the exception)	`<button [attr.aria-label]="help">help</button>`
Class	`class` property	`<div [class.special]="isSpecial">Special</div>`
Style	`style` property	`<button [style.color]="isSpecial ? 'red' : 'green'">`

Summary


Module
Component
{}
Module
Service
{}
Module
value
3.1415
Module
Fn
λ
Template
< >
Metadata
Directive
{}
Property
Binding
Metadata
Event
Binding
Injector
Service
Component

Let’s Code!!!

Let’s create an Angular application.
First we need Nodejs installed (https://nodejs.org/en/).
With node and npm installed we need to run some commands.

Install the Angular CLI

You use the Angular CLI to create projects, generate application and library code, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.

To install the Angular CLI, open a terminal window and run the following command:

npm install -g @angular/cli

To create a new workspace and initial starter app:

Run the CLI command ng new and provide the name training, as shown here:

ng new training

You will be asked these two questions:
Would you like to add Angular routing? Y (Yes)
Which stylesheet format would you like to use? SCSS

After the project has been successfully created, you can check if everything is running correctly.

cd training && ng serve —open

The default port that angular uses is 4200. The default server is available at *http:///localhost:4200/*

Project structure

Your most important files for this training will be inside ***/src*** folder.

Source files for the root-level application project.

APP SUPPORT FILES	PURPOSE
`app/`	Contains the component files in which your application logic and data are defined. See details below.
`assets/`	Contains image and other asset files to be copied as-is when you build your application.
`environments/`	Contains build configuration options for particular target environments. By default there is an unnamed standard development environment and a production ("prod") environment. You can define additional target environment configurations.
`favicon.ico`	An icon to use for this application in the bookmark bar.
`index.html`	The main HTML page that is served when someone visits your site. The CLI automatically adds all JavaScript and CSS files when building your app, so you typically don't need to add any `<script>` or`<link>` tags here manually.
`main.ts`	The main entry point for your application. Compiles the application with the JIT compiler and bootstraps the application's root module (AppModule) to run in the browser. You can also use the AOT compiler without changing any code by appending the `--aot` flag to the CLI `build` and `serve` commands.
`polyfills.ts`	Provides polyfill scripts for browser support.
`styles.sass`	Lists CSS files that supply styles for a project. The extension reflects the style preprocessor you have configured for the project.
`test.ts`	The main entry point for your unit tests, with some Angular-specific configuration. You don't typically need to edit this file.

Configuration

Since we don’t have a backend service, let’s emulate one. This package is meant for use only for development!!

Let’s install it:

npm install angular-in-memory-web-api

This package will intercept all our HTTP calls, so we don’t need any backend service to run our app.

This code is all available at:
https://github.com/danilanna/angular-training/tree/step-1

Let’s create our first component

Type this on the terminal:

ng generate component home

A folder should be generated inside /src/app/home

When we run this command, angular CLI creates a new component to us.

We cannot see the HomeComponent yet. This is because we did not add it yet to our application. Let’s do that.

Go to app.component.html file and add "<app-home></app-home>" to the last line of the file.

Now you can see the home.component.html, which should be “home works!”.

This happened because the HomeComponent is bootstrapped by the tag “<app-home></app-home>”, without this tag, Angular is not capable to show the component. You can check this tag matches exactly what is inside the file home.component.ts at: selector: ‘app-home’. This is the selector that Angular will use to bootstrap the component.

Now, let’s refactor it and move the code from app.component.html to home.component.html. Let’s leave only <router-outlet></router-outlet> in the app.component.html file.

You should see a blank page now! Why? Because there is no more “<app-home></app-home>” tag.

Instead, let’s create the router for it.

Let’s create our route

Open the file ***app-routing.module.ts***.

You can see this file has an array of type **Routes**.

The **Routes** type, accepts many parameters, for now we will use only *path* and *component*.

In the ***routes*** array, let’s create a JS object {}, and inside this object let’s put the *path* and *component* attributes.

```
{path: '', component: HomeComponent}
```

Note that you should import the HomeComponent also. If your IDE doesn’t do this automatically (import { HomeComponent } from ‘./home/home.component';)

What is happening here?

The *path* attribute is telling to Router what is path that should match and when is matches, should open the provided *component*.

The empty path we passed, means the root folder for our application.

Now you should see again the page working, and we didn’t have to put the tag “***<app-home></app-home>***“ in the application, we left this responsibility to the *router*.

Let’s move also the *title = ‘training'; from app.component.ts to home.component.ts* to leave the *app.component.ts* body completely empty.

This code is all available at:
https://github.com/danilanna/angular-training/tree/step-2

Let’s create another component

Type this on the terminal:

ng generate component germany-states

A folder should be generated inside /src/app/germany-states

Let’s create again, another the router for it.

Open the file ***app-routing.module.ts.***

Let’s now point create the path with will route to the GermanyStatesComponent.

{path: ‘germany-states‘, component: GermanyStatesComponent}.

Note: Don’t forget to import the *GermanyStatesComponent* if your IDE doesn’t do it automatically for you.

```
import { GermanyStatesComponent } from ‘./germany-states/germany-states.component';
```

Now if we type in the browser http://localhost:4200/germany-states you will see that the router is working. The page should show: ”germany-states works!”

Let’s create a mock

Let’s create a new file called “***germany-states.ts***” inside the folder /src/app/germany-states

In this file we will represent the “model" of our germany-states.

To do so, we will create an ***interface*** to represent this object. But WHY an ***interface*** and not a ***class***?

An interface is a virtual structure that only exists within the context of TypeScript. The TypeScript compiler uses interfaces solely for *type-checking* purposes. Once your code is transpiled to its target language, it will be stripped from its interfaces - JavaScript isn’t typed, there’s no use for them there. An interface is simply a structural contract that defines what the properties of an object should have as a name and as a type.

The germany-states will contain an ***id*** and a ***name.*** So let’s create it:

```
export interface GermanyStates {
  id: number,
  name: string,
}
```

Now that we have defined the structure, let’s create the service for it.

In the terminal, type:

ng generate service germany-states/germany-states

This will generate a new service for us at /src/app/germany-states folder.

Let’s now we will create a method to show the Germany states:

```
getStates(): GermanyStates[] { return this.states; }
```

Now we will create an object to be returned when we call the method *getStates()*
states: GermanyStates[] = [{ id: 1, name: "Baden-Württemberg" }];
The full list of Germany states can be found at https://github.com/danilanna/angular-training/blob/step-3/src/app/germany-states/germany-states.service.ts

Let’s call the service

Back to our germany-states component, let’s modify it. Open the **germany-states.component.ts** file.

Let’s create a variable to represent the GermanyStates:

```
states: GermanyStates[] = [];
```

Let’s update our constructor and add a parameter with type GermanyStatesService.

```
constructor(private service: GermanyStatesService) { }
```

We will use the DI of Angular, and we will make the Angular container to inject the GermanyStatesService dependency for us, this is how DI works in Angular.

After that, let’s call the method *getStates()* inside the GermanyStatesService.

Let’s do that inside the ngOnInit method. The ngOnInit is a life cycle hook called by Angular to indicate that Angular is done creating the component. It is a good practice to initialize all the angular scope things under the ngOnInit() not under constructor. So let’s put the following code into the ngOnInit() method.

```
this.states = this.service.getStates();
```

Now, let’s modify the HTML. Let’s create a list to show all the states:

```
<ul> <li *ngFor="let state of states">{{state.name}}</li> </ul>
```

The *ngFor is Angular's *repeater* directive. It repeats the host element for each element in a list.
The syntax in this example is as follows:

- is the host element.
- states holds the mock states list from the GermanyStatesComponent class, the mock states list.
- state holds the current state object for each iteration through the list.
- The ***let*** word will make sure that the variable *state* is only scoped inside the loop

Now you should be able to see all the germany-states available in the GermanyStatesService.

http://localhost:4200/germany-states

This code is all available at:
https://github.com/danilanna/angular-training/tree/step-3

Let’s play a bit

Let’s create a Master/Detail behaviour.

Let’s first import the *FormsModule* used by angular. Go to *app.module.ts* file and add the import:
import { FormsModule } from ‘@angular/forms';

In the *imports* array, add the ***FormsModule:***
imports: [BrowserModule, FormsModule, AppRoutingModule]

Restart your server!!

Let’s now back to our germany-states component. Open the germany-states.component.html file.

Let’s add a click event to our state list: (click)=“onSelect(state)"
<li *ngFor="let state of states" (click)=“onSelect(state)">{{state.name}}

We are using the event binding here, bind by the click event. Every time the user click in an element, it will fire an event and call the “*onSelect*” method, passing the clicked hero.

Let’s add the detail now:

```
<div *ngIf="selectedState">

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

  </div>
```

The ****ngIf*** is used when we wants to hide or show something in the HTML, based on some property or value. In this case we are telling to angular to only show the detail section when the variable ***selectedState*** is not null.

The “|” is a pipe, and angular provides many default pipes already, which are very useful when formatting dates, currency for example…

The ***[(ngModel)]*** is how two-way data-binding works in angular. In this case the data flows from HTML to component, and from the component to HTML.

Let’s play a bit

Now, let’s modify the germany-states.component.ts file.

We need to create a variable, in order to show what have been selected by the user.

```
selectedState: GermanyStates;
```

And we also need to create the method, which will be called when the user clicks on any element in the list.

```
onSelect(state: GermanyStates): void {
    this.selectedState = state;
  }
```

Done! The Master/Detail is completed!

Let’s go back to the germany-states.component.html and modify the style of the list.

Let’s add a functionality, which will change the colour of the element, if that element is selected to show in the details.

```
[style.color]="selectedState === state ? 'blue' : ‘black'"
```

We are changing the colour based on a condition. This is possible thanks to the property binding in angular.

```
<li *ngFor="let state of states" (click)="onSelect(state)" [style.color]="selectedState === state ? 'blue' : ‘black'">{{state.name}}</li>
```

This code is all available at:
https://github.com/danilanna/angular-training/tree/step-4

Let’s refactor a bit

Let’s move the detail part from the germany-states.component to a new one called germany-states-detail.component.

First, let’s create the new component:
ng generate component germany-states-detail

Let’s move the detail HTML part we just created, and move it to the new created component.

In the file germany-states-detail.component.ts let’s create again the *selectedState* variable, which will be used to show the state information. It’s almost identical the one we have previously created, but now, the variable will be controlled externally, in order to achieve this, we have to use the *@Input* decorator.

```
@Input() selectedState: GermanyStates;
```

Let’s add now the app-germany-state-details selector to the germany-states.component.html, exactly where the detail part was before:

```
<app-germany-states-detail [selectedState]=“selectedState"></app-germany-states-detail>
```

Now, instead of just adding the selector “app-germany-states-detail“, we also need to add the property “*selectedState*”, in order to be able to make the biding works.

Angular will bind the property “*selectedState*” using the “@Input” decorator in the detail component.

This code is all available at:
https://github.com/danilanna/angular-training/tree/step-5

Let’s add some style

Let’s add Boostrap to our project. There are many CSS libraries that we could use, for example Angular Material, but we also needed to learn it, so let’s add just a simple CSS library to make our project a little bit more beauty.

Open index.html and add the CSS inside the <head> tag.

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.5.3/dist/css/bootstrap.min.css" integrity="sha384-TX8t27EcRE3e/ihU7zmQxVncDAy5uIKz4rEkgIXeMed4M0jlfIDPvg6uqKI2xXr2" crossorigin="anonymous">
```

Open app.component.html and let’s add a nav-bar before the <router-outlet> tag:

```
<nav class="navbar navbar-expand-lg sticky-top bg-dark">
  <div>
    <ul class="navbar-nav">
      <li class="nav-item">
        <a class="nav-link text-white" routerLink="/">Home</a>
      </li>
      <li class="nav-item">
        <a class="nav-link text-white" routerLink="/germany-states">Germany States</a>
      </li>
    </ul>
  </div>
</nav>
```

Now we can navigate in our application.

Let’s add some style

Let’s now add the table to our germany-states.component.html:

```
<table class="table table-striped">
    <thead>
        <tr>
            <th scope="col">Id</th>
            <th scope="col">Name</th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="let state of states" (click)="onSelect(state)" [class.table-dark]="selectedState === state">
            <td>{{state.id}}</td>
            <td>{{state.name}}</td>
        </tr>
    </tbody>
</table>
```

Note that we changed the style property from [style.color] to [class.table-dark]. The difference now we instead of adding the color property to the tag, we are now adding a class to the tag.

Now let’s modify the details section. Open the file germany-states-detail.component.html and add:

```
  <div class="card bg-light mb-3" *ngIf="selectedState">
    <div class="card-header">Details</div>
    <div class="card-body">
      <h5 class="card-title">Id: {{selectedState.id}}</h5>
      <p class="card-text">Name: {{selectedState.name | uppercase}}</p>
    </div>
  </div>
```

This code is all available at:
https://github.com/danilanna/angular-training/tree/step-6

Let’s connect with backend

First let’s configure our call. Let’s generate the mock-service:

ng generate service in-memory

Let’s modify the app.module.ts and add our services.

```
import { HttpClientModule } from '@angular/common/http';
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';
import { InMemoryService } from './in-memory.service';
```

In normal situation, in order to make HTTP calls, it’s require to import the “*HttpClientModule*”. We are importing also the in-memory module because our project doesn’t have any backend to connect, so we are mocking. In normal projects we didn’t have to make this configuration and import this module.

In the same file, we need to add these modules to our *imports* array:

```
HttpClientModule,
HttpClientInMemoryWebApiModule.forRoot(InMemoryService),
```

Again, in normal situation, you would not need to import the InMemory module, this is only for our mocking purposes.

Let’s connect with backend

Now, let’s modify the recently created in-memory.service.ts. Let’s add a dependency:

```
import { InMemoryDbService } from 'angular-in-memory-web-api';
```

Now, let’s add this interface to our class:

```
export class InMemoryService implements InMemoryDbService
```

Let’s implement the method createDb(), and we need to return and object, which contains the states. This states are the same states that are in the file *germany-states.service.ts*, for example:

```
createDb() {
    const states = [
      {
        id: 1,
        name: "Baden-Württemberg",
      },

    ];
    return {states};
  }
```

Add all the states from the file *germany-states.service.ts* to this array.

Let’s connect with backend

Let’s refactor the *germany-states.service.ts*. Add the imports:

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

Since we moved the states array to the in-memory service, we don’t need the *states* variable anymore.

Add the api url to the file:

```
private statesUrl = ‘api/states';
```

Add the headers:

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

In order to make the HTTP calls, add a constructor, and in this constructor we need to inject the HttpClient:

```
constructor(private http: HttpClient) { }
```

Now we need to modify the method *getStates()*, it will now return an Observable of GermanyStates, and make a HTTP GET call:

```
getStates(): Observable<GermanyStates[]> {
    return this.http.get<GermanyStates[]>(this.statesUrl);
  }
```

Let’s refactor the *germany-states.component.ts. We need to modify the ngOnInit()* method.

Now instead of just assigning the return of the function to *this.states*, we need to *subscribe* to the *observable*, and then assign the result to the states variable:

```
this.service.getStates()
    .subscribe(states => this.states = states);
```

This code is all available at:
https://github.com/danilanna/angular-training/tree/step-7

Let’s add a functionality

Let’s clear the detail on the page Germany states. To do so let’s create an event, which will interact from the child to the parent.

Let’s modify the germany-states.component.html file. Let’s add the event to the detail tag:

```
(resetEvent)=“reset()” / <app-germany-states-detail [selectedState]="selectedState" (resetEvent)="reset()"></app-germany-states-detail>
```

Let’s now modify the germany-states.component.ts file and create the *reset()* function.

```
  reset() {
    this.selectedState = null;
  }
```

Now let’s modify the child component. On germany-states-detail.component.ts file, add a button to call the clear operation.

```
<button class="btn btn-primary" (click)=“clear()">Clear</button>
```

Now let’s modify the Typescript component. We need first to import the necessary dependencies.

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

Let's create now the event, that will emit an event to the parent component. So let’s use *@Output* decorator to achieve this. The *@Output* allow data to flow from the child *out* to the parent. *@Output()* marks a property in a child component as a doorway through which data can travel from the child to the parent. The child component then has to raise an event so the parent knows something has changed. To raise an event, *@Output()* works hand in hand with EventEmitter, which is a class in *@angular/core* that you use to emit custom events.

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

  clear() {
    this.resetEvent.emit();
  }
```

This code is all available at:
https://github.com/danilanna/angular-training/tree/step-7

Let’s create more functionalities

Let’s generate a new component:

ng generate component germany-states-edit

Let’s add it to the route, add the import:

```
import { GermanyStatesEditComponent } from ‘./germany-states-edit/germany-states-edit.component';
```

And add the a new entry to the route:

```
{path: 'germany-states/:id', component: GermanyStatesEditComponent},
```

Let’s add more columns to the germany-states.component.html, edit and remove to the table head:

```
<th scope="col">Edit</th>
<th scope=“col">Remove</th>
```

Now add the table body:

```
<td><a routerLink="/germany-states/{{state.id}}" class="btn btn-info">Edit</a></td>
<td><button class="btn btn-danger" (click)="delete(state)">Remove</button></td>
```

In the code above, we are creating a router which links to the router we have just created. And also a button which will handle the delete action.

Let’s extract the method that fetch the states from *ngOnInit* to a new method *getStates:*

```
  getStates(): void {
    this.service.getStates().subscribe(states => this.states = states);
  }

  ngOnInit(): void {
    this.getStates();
  }
```

In the germany-states.component.ts, let’s now create the method which will handle the delete action.

```
  delete(state: GermanyStates) {
    this.service.deleteState(state.id).subscribe(() => {
      this.getStates();
      this.reset();
    });
  }
```

This code will call the delete method on the service level, that will be created.

Let’s create more functionalities

Let’s modify the GermanyStatesService and add the new HTTP methods:

First, add the imports:

```
import { Observable, of } from 'rxjs';
import { catchError } from ‘rxjs/operators';
```

Let’s add a method that will handle the error calls*:*

```
 private handleError<T>(operation: string, result?: T) {
    return (error: any): Observable<T> => {

      console.error(error);

      console.log(`${operation} failed: ${error.message}`);

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

This method handles the HTTP error call. Currently it’s only logging the error in the console.

Let’s modify the method *getStates(), and add the error handling:*

```
  getStates(): Observable<GermanyStates[]> {
    return this.http.get<GermanyStates[]>(this.statesUrl)
    .pipe(catchError(this.handleError<GermanyStates[]>('getStates', [])));
  }
```

Let’s create more functionalities

Let’s add now the remaining methods.

```
  getState(id: number): Observable<GermanyStates> {
    const url = `${this.statesUrl}/${id}`;
    return this.http.get<GermanyStates>(url)
    .pipe(catchError(this.handleError<GermanyStates>('getState', {} as GermanyStates)));
  }

  updateState(state: GermanyStates): Observable<GermanyStates> {
    return this.http.put<GermanyStates>(this.statesUrl, state, this.httpOptions)
    .pipe(catchError(this.handleError<GermanyStates>('updateState', {} as GermanyStates)));
  }

  deleteState(id: number): Observable<GermanyStates> {
    const url = `${this.statesUrl}/${id}`;
    return this.http.delete<GermanyStates>(url, this.httpOptions)
    .pipe(catchError(this.handleError<GermanyStates>('deleteState')));
  }
```

Let’s create more functionalities

Let’s modify the edition page. Open the file germany-states-edit.component.html, and add the form:

```
<div *ngIf="state">
<form>
    <h2>Edit: {{state.id}}</h2>
    <div class="form-group">
      <label for="stateName">Name</label>
      <input type="text" class="form-control" [(ngModel)]="state.name" id="stateName" name="stateName">
    </div>
    <button class="btn btn-primary mr-2" (click)="goBack()">Back</button>
    <button type="submit" class="btn btn-primary" (click)="save()">Save</button>
  </form>
</div>
```

Let’s create more functionalities

Let’s modify now the file germany-state-edit.component.ts, and let’s add the imports:

```
import { ActivatedRoute } from '@angular/router';
import { Location } from '@angular/common';
import { GermanyStatesService } from '../germany-states/germany-states.service';
import { GermanyStates } from '../germany-states/germany-states';
```

Let’s add a variable, that will hold the *state* value:

```
state: GermanyStates;
```

In the constructor, let’s add the required dependencies and use the DI of Angular

```
constructor(private route: ActivatedRoute, private service: GermanyStatesService, private location: Location) { }
```

In the *ngOnInit* we will retrieve the *id* from the url param, and then call the service to fetch the state by id:

```
  ngOnInit(): void {
    const id =+ this.route.snapshot.paramMap.get('id');
    this.service.getState(id).subscribe(state => this.state = state);
  }
```

Now let’s add the save method, and call the service to update the state

```
save(): void {
    this.service.updateState(this.state).subscribe(() => this.goBack());
  }
```

Now let’s add the goBack method, which simply go to the back page

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

This code is all available at:
https://github.com/danilanna/angular-training/tree/step-8

Let’s refactor!

Now that we have some components, let’s think how they are related. We have only germany-states page, which means we could isolate all its components to one module. Let’s do this!

Let’s create a new module:

ng generate module germany-states/germany-states --routing=true --flat=true

The above command will create a new module, the --routing=true option will also create a routing module, and the option --flat=true means it will be created at the /germany-states folder.

Now let’s move all our germany-states components (germany-states-detail, germany-states-detail) into the folder germany-states. (UPDATE ALL THE IMPORTS!!!)

Go to app.module.ts file, remove all the germany-states components that are imported. Remove the components from the declarations array. Make sure that the module GermanyStatesModule is imported, and it is on *imports* array.

Open now the file app-routing.module.ts. Remove again the germany-states components, and also remove the routing.

Let’s configure now the germany-states.module.ts. Add the imports for our components:

```
import { GermanyStatesComponent } from './germany-states.component';
import { GermanyStatesDetailComponent } from './germany-states-detail/germany-states-detail.component';
import { GermanyStatesEditComponent } from './germany-states-edit/germany-states-edit.component';
```

Add them to the declarations:

```
declarations: [
    GermanyStatesComponent,
    GermanyStatesDetailComponent,
    GermanyStatesEditComponent,
  ],
```

The imports should contain the germany-states routing, commons and forms modules:

```
imports: [
    CommonModule,
    FormsModule,
    GermanyStatesRoutingModule
  ]
```

Do not forget to import them also:

```
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { GermanyStatesRoutingModule } from './germany-states-routing.module';
```

Let’s refactor!

Now let’s configure the routes. Open germany-states-routing.module.ts. Let’s import our components:

```
import { GermanyStatesComponent } from './germany-states.component';
import { GermanyStatesEditComponent } from './germany-states-edit/germany-states-edit.component';
```

Let’s add the routing:

```
const routes: Routes = [
  {path: 'germany-states', component: GermanyStatesComponent},
  {path: 'germany-states/:id', component: GermanyStatesEditComponent},
];
```

We don’t need the HomeComponent anymore, right? Let’s remove it! Delete the folder home!
Remove also the HomeComponent from the app.module.ts file. Remove the import, and also from the declarations array.
Open the file app-routing.module.ts and also remove the HomeComponent import, and also from the route.
Add a new value in the route:

```
const routes: Routes = [
  {path: '', redirectTo: 'germany-states', pathMatch: 'full'},
];
```

Now every time the user open the root path, it will be redirected to *germany-states* routing, which will show the *germany-states* component!

This code is all available at:
https://github.com/danilanna/angular-training/tree/step-9

Tests

Let’s test our project now! To do so we need to run:

ng test

This kind of tests are more suitable for integration and unit tests.

There are many tests failing in our project. All the fixed tests can be found at https://github.com/danilanna/angular-training/tree/step-10

When we run:

ng e2e

This as the name suggests, this will run the angular e2e tests.

The file *app.po.ts* is used for holding the Page Object of our application. The Page Object Class has all of the logic to find the elements on our page and how to navigate to the URL.

To check the test coverage, you can run:

ng test --no-watch --code-coverage

It will create a folder at the root project called *coverage*, inside it you can open the file coverage/*index.html* and check the coverage of the project.

This code is all available at:
https://github.com/danilanna/angular-training/tree/step-10