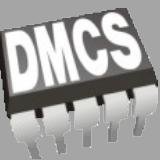




Interactive Web Applications



1. Routing
2. Angular \leftrightarrow SpringBoot (CORS)
3. Dependency Injection
4. Http

1

Routing

Routing

In web development, **routing** means **splitting** the application into **different areas** usually based on rules that are derived from the current **URL** in the browser.

For instance, if we visit the `/` path of a website, we may be visiting the **home** route of that website. Or if we visit `/about` we want to render the “**about page**”, and so on.

Why Do We Need Routing?

Defining **routes** in our application is useful because we can:

- **separate different areas** of the app,
- **maintain the state** in the app,
- **protect areas** of the app based on certain rules.



Routing

Because Angular app is **client-side**, it is **not** technically **required** that we change the **URL** when we change “pages”. But it is worth thinking about for a minute: what would be the consequences of using the same URL for all pages?

- You would not be able to **refresh the page** and keep your location within the app.
- You would not be able to **bookmark a page** and come back to it later.
- You would not be able to **share the URL** of that page with others.



Routing

The browser is a familiar model of **application navigation**:

- **Enter a URL** in the address bar and the browser navigates to a corresponding page.
- **Click links** on the page and the browser navigates to a new page.
- **Click the browser's back and forward buttons** and the browser navigates backward and forward through the history of pages you have seen.



Routing

The **Angular Router** ("the router") borrows from this model. It can **interpret** a browser **URL** as an instruction to navigate to a **client-generated view**. It can **pass optional parameters** along to the supporting view component that helps it decide what specific content to present. You can **bind the router to links**. You can **navigate imperatively** when the user **clicks a button**, **selects from a drop box**, or in response to some other stimulus from any source. And the **router logs activity** in the **browser's history journal** so the **back and forward buttons** work as well.



Routing – <base href>

Most routing applications should add a **<base>** element to the **index.html** as the **first child in the <head> tag** to tell the router how to compose navigation URLs. If the app folder is the application root, set the **href** value exactly as shown here.

```
<> index.html x
1  <!doctype html>
2  <html lang="en">
3  <head>
4    <meta charset="utf-8">
5    <title>Angular21Iwa2026</title>
6    <base href="/">
7    <meta name="viewport" content="width=device-width, initial-scale=1">
8    <link rel="icon" type="image/x-icon" href="favicon.ico">
9  </head>
10 <body>
11   <app-root></app-root>
12 </body>
13 </html>
```

This line declares the **base HTML tag**. This tag is traditionally used to tell the browser where to look for images and other resources declared using relative paths. **Angular Router** also relies on this tag to determine how to construct its routing information.



Routing

The **Angular Router** is an optional service that presents a particular component view for a given URL. It is not part of the **Angular core**. It is in its own library package, **@angular/router**. Import what you need from it as you would from any other Angular package.

```
TS app.ts x
1 import { Component, signal } from '@angular/core';
2 import { RouterLink, RouterOutlet } from '@angular/router';
3 import { Calculator } from './calculator/calculator';
4 import { Helloworld } from './helloworld/helloworld';
5
6 @Component({ new *
7   selector: 'app-root',
8   imports: [RouterOutlet, Calculator, Helloworld, RouterLink],
9   templateUrl: './app.html',
10  styleUrls: ['./app.css']
11 })
12 export class App {
13   protected readonly title : WritableSignal<string> = signal( initialValue: 'angular21-iwa2026');
14 }
```



Routing

```
TS app.routes.ts x
1 import { Routes } from '@angular/router';
2 import {Helloworld} from './helloworld/helloworld';
3 import {Calculator} from './calculator/calculator';
4
5 export const routes: Routes = [ new *
6   {path: 'hello', component: Helloworld},
7   {path: 'calculator', component: Calculator}
8 ];
```

```
TS app.config.ts x
1 import { ApplicationConfig, provideBrowserGlobalErrorListeners } from '@angular/core';
2 import { provideRouter } from '@angular/router';
3
4 import { routes } from './app.routes';
5
6 export const appConfig: ApplicationConfig = { new *
7   providers: [
8     provideBrowserGlobalErrorListeners(),
9     provideRouter(routes)
10  ]
11 };
```



Routing

The **routes array** of routes describes how to **navigate**. Each **Route** maps a **URL path** to a **component**. There are no leading slashes in the path. The **router** parses and builds the final **URL** for you, allowing you to use both relative and absolute paths when navigating between application views.

The **empty path** represents the **default path** for the application, as it typically is at the start. This **default route** **can redirect** to the route for the **/home** URL and, therefore, can display the **HomeComponent**.



Routing

The **order** of the **routes** in the **configuration matters** and this is by design. The router uses a **first-match wins strategy** when matching routes, so more specific routes should be placed above less specific routes. The **wildcard** route `{ path: '**', component: PageNotFoundComponent }` should come last because it matches every URL and should be selected only if no other routes are matched first.



Routing – <router-outlet>

When we change **routes**, we may want to keep our **outer “layout”** template and only substitute the “inner section” of the page with the route’s component.

In order to describe to Angular **where in our page** we want to render the contents for each route, we use the **RouterOutlet directive**.

The **router-outlet** element indicates where the contents of **each route component** will be rendered.



Routing – <router-link>

We might try linking to the routes **directly** using pure HTML:

```
<a href="/#/home">Home</a>
```

But if we do this, we will notice that **clicking the link triggers a page reload** and that is definitely not what we want when programming **single page applications**.

To solve this problem, **Angular** provides a solution that can be used to link to routes with **no page reload**: the **RouterLink directive**.

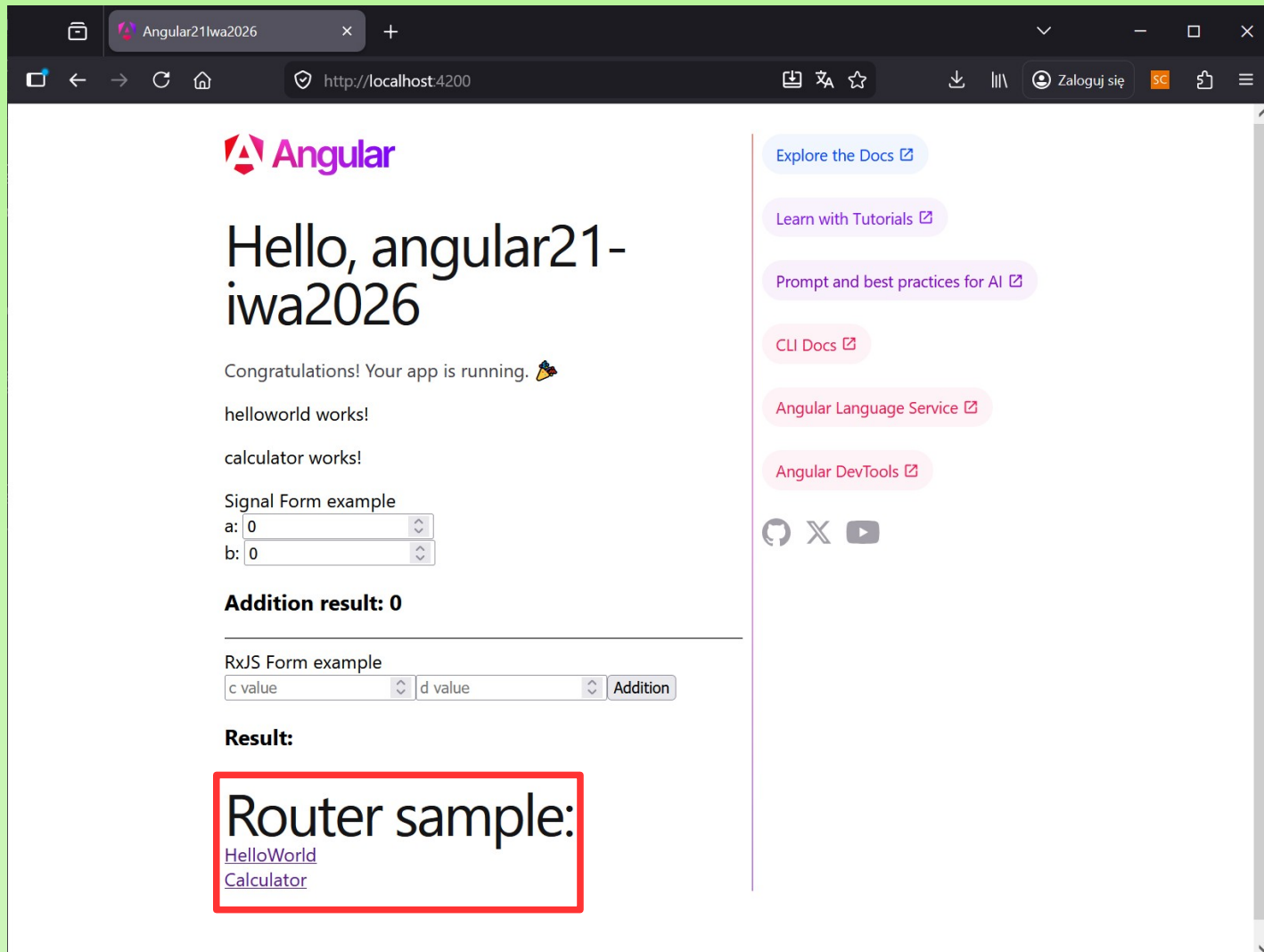


Routing – <router-outlet> and <router-link>

```
<> app.html x
187     <main class="main">
188       <div class="content">
189         <div class="left-side">
235           <h1>Hello, {{ title() }}</h1>
236           <p>Congratulations! Your app is running. 🎉</p>
237           <app-helloworld></app-helloworld>
238           <app-calculator></app-calculator>
239           <h1>Router sample:</h1>
240           <a routerLink="hello">HelloWorld</a>
241           <br>
242           <a routerLink="calculator">Calculator</a>
243           <br>
244           <router-outlet />
245         </div>
```



Routing – simple example



Angular

Hello, angular21-iwa2026

Congratulations! Your app is running. 🎉

helloworld works!

calculator works!

Signal Form example

a:

b:

Addition result: 0

RxJS Form example

c value d value Addition

Result:

Router sample:
[HelloWorld](#)
[Calculator](#)

Explore the Docs [↗](#)

Learn with Tutorials [↗](#)

Prompt and best practices for AI [↗](#)

CLI Docs [↗](#)

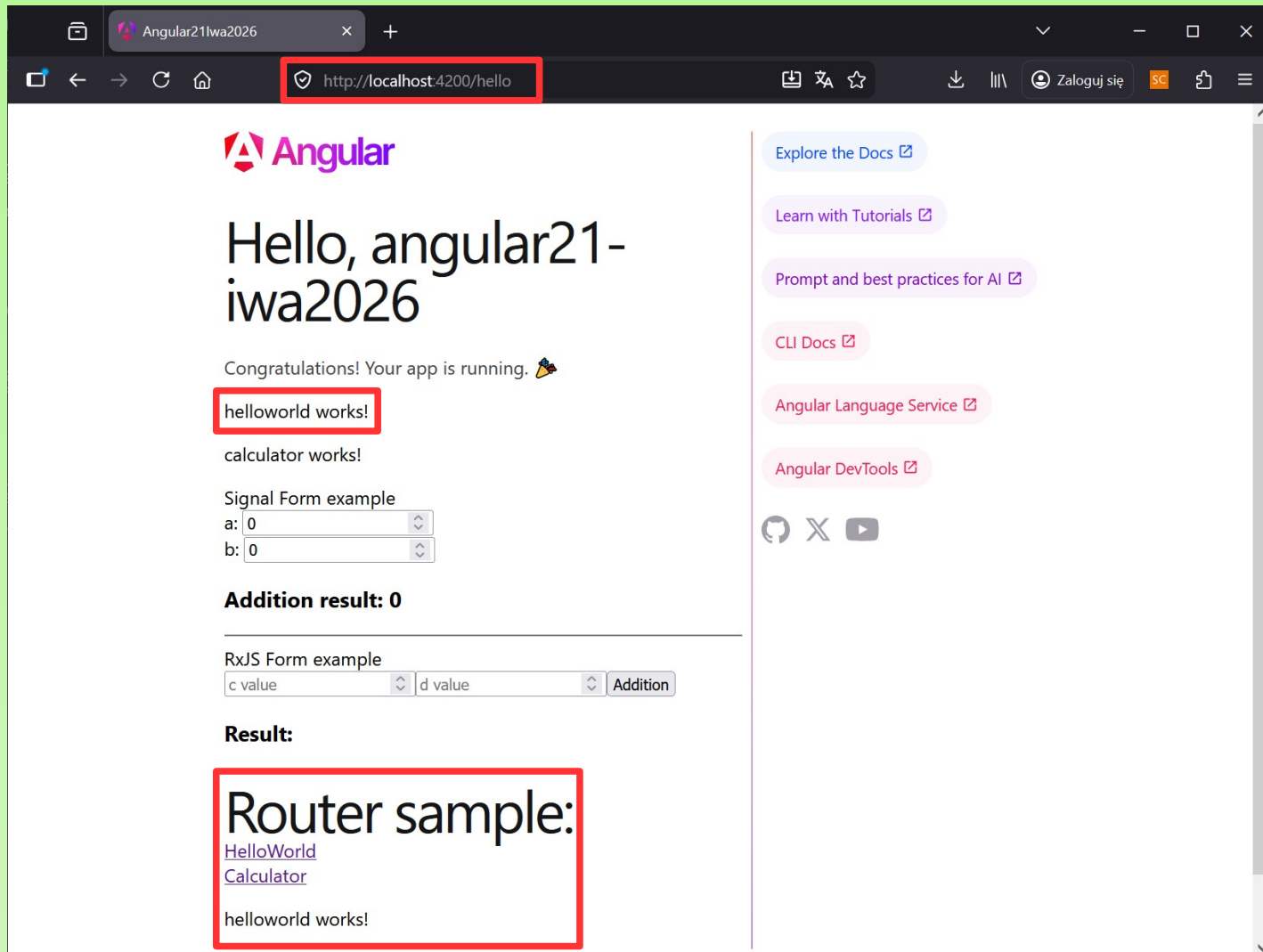
Angular Language Service [↗](#)

Angular DevTools [↗](#)

🔄 ✕ 📺




Routing – simple example



Angular21Iwa2026

http://localhost:4200/hello

 **Angular**

Hello, angular21-iwa2026

Congratulations! Your app is running. 🎉

helloworld works!

calculator works!

Signal Form example

a:

b:

Addition result: 0

RxJS Form example

c value d value Addition

Result:

Router sample:

[HelloWorld](#)

[Calculator](#)

helloworld works!

[Explore the Docs](#)

[Learn with Tutorials](#)

[Prompt and best practices for AI](#)

[CLI Docs](#)

[Angular Language Service](#)

[Angular DevTools](#)



Routing – simple example

Angular21iwa2026

http://localhost:4200/calculator

Angular

Hello, angular21-iwa2026

Congratulations! Your app is running. 🎉

helloworld works!

calculator works!

Signal Form example

a: 0

b: 0

Addition result: 0

RxJS Form example

c value d value Addition

Result:

Router sample:

[HelloWorld](#)

[Calculator](#)

calculator works!

Signal Form example

a: 0

b: 0

Addition result: 0

RxJS Form example

c value d value Addition

Result:

Explore the Docs

Learn with Tutorials

Prompt and best practices for AI

CLI Docs

Angular Language Service

Angular DevTools



2

Angular ↔ SpringBoot

What is CORS?

The web pages you visit make frequent requests to load assets like images, fonts, and more, from many **different places across the Internet**. If these requests for assets go **unchecked**, the **security** of your browser may be at **risk**. For example, your browser may be subject to **hijacking**, or your browser **might blindly download malicious code**. As a result, many **modern browsers** follow **security policies** to **mitigate** such risks.



What is CORS?

The **same-origin policy** is **very restrictive**. Under this policy, a document (i.e., like a web page) hosted on **server A** can only interact with other documents that are also on **server A**. In short, the **same-origin policy** enforces that documents that interact with each other have **the same origin**.

An **origin** is made up of the following **three parts**: the **protocol**, **host**, and **port number**.



What is CORS?

Not having a security policy can be **risky**, but a security policy like **same-origin** is a bit too **restrictive**. However there are security policies that strike a **mix of both**, like **cross-origin**, which has evolved into the **cross-origin resource sharing** standard, often abbreviated as **CORS**.

CORS is a mechanism that uses **additional HTTP headers** to let a user agent **gain permission** to access selected resources from a server on a **different origin (domain)** than the site currently in use.



What is CORS?

For **security reasons**, browsers **restrict cross-origin HTTP requests** initiated from within scripts. For example, **XMLHttpRequest** and the **Fetch API** follow the **same-origin policy**. This means that a web application using those APIs can only request HTTP resources from the same domain the application was loaded from, **unless CORS** headers are used. It means that servers must implement ways to **handle** requests from **origins outside** of their own. **CORS** allows servers to **specify who** (i.e., which origins) can access the assets on the server, among many other things.



Enable CORS on the Server

To enable **CORS** on the server, add a **@CrossOrigin** annotation to the **Controllers**.



```
© StudentRestController.java ×  
13 @RestController  
14 @CrossOrigin(origins = "http://localhost:4200")  
15 @RequestMapping("/students")  
16 public class StudentRestController {
```

It could also be configured at **method** level or in general at **global** level in application configuration.



Enable CORS on the Server

If you are using **Spring Security**, make sure to [enable CORS at Spring Security level](#) as well to allow it to leverage the configuration defined at Spring MVC level.

```
WebSecurityConfig.java ×  
25 public class WebSecurityConfig {  
51     @Bean  
52     public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {  
53         http  
54         .cors(Customizer.withDefaults())
```



3

Dependency Injection

Dependency injection is an important application **design pattern** used to organize and share code across an application by allowing you to "inject" features into different parts.

As an application grows, developers often need to reuse and share features across different parts of the codebase.

Dependency injection allows developers to address common challenges such as:

Improved code maintainability: It allows cleaner separation of concerns which enables easier refactoring and reducing code duplication.

Scalability: Modular functionality can be reused across multiple contexts and allows for easier scaling.

One way to think about “**the injector**” is as a **replacement** for the **new operator**. That is, instead of using the language-provided **new** operator, **Dependency Injection** lets us configure how objects are created.



A **dependency** is any **object, value, function or service** that a class needs to work but does not create itself. In other words, it creates a relationship between different parts of your application since it would not work without the dependency.

There are two ways that code interacts with any **dependency injection system**:

- Code can **provide**, or make available, values.
- Code can **inject**, or ask for, those values as dependencies.

"Values," in this context, can be any JavaScript value, including objects and functions. Common types of injected dependencies include:

- Configuration values: Environment-specific constants, API URLs, feature flags, etc.
- Factories: Functions that create objects or values based on runtime conditions
- Services: Classes that provide common functionality, business logic, or state



Dependency Injection

29

dr inż. Rafał Kotas, rkotas@dmcs.pl

Services are reusable pieces of code that can be shared across your Angular application. They typically handle data fetching, business logic, or other functionality that multiple components need to access.

```
TS studentService.ts x
1  import {inject, Injectable} from '@angular/core';
2  import {HttpClient, HttpHeaders} from '@angular/common/http';
3  import {catchError, Observable, of, tap} from 'rxjs';
4  import {Student} from '../models/student';
5
6  const httpOptions = {
7    headers: new HttpHeaders({ 'Content-Type': 'application/json' })
8  };
9
10  @Injectable({
11    providedIn: 'root',
12  })
13  export class StudentService {
14
15    private http : HttpClient = inject(HttpClient);
```



Register a service provider – root, @NgModule or @Component?

src/app/heroes/heroes.component.ts

```
import { Component } from '@angular/core';
import { HeroService } from '../hero.service';

@Component({
  selector: 'app-heroes',
  providers: [ HeroService ],
  template: `
    <h2>Heroes</h2>
    <app-hero-list></app-hero-list>
  `
})
export class HeroesComponent { }
```

```
    @Injectable({
      providedIn: 'root'
    })
    export class StudentService {
```

src/app/app.module.ts (providers)

```
providers: [
  UserService,
  { provide: APP_CONFIG, useValue: HERO_DI_CONFIG }
],
```



Register a service provider – root, @NgModule or @Component?

The three choices lead to differences in **service scope** and **service lifetime**.

- **Preferred:** At the application **root** level using **providedIn**. Once created, a service instance lives for the life of the app and Angular injects this one service instance in every class that needs it.
- Angular **module providers** (**@NgModule.providers**) are registered with the application's **root injector**. Angular can **inject** the corresponding services in **any class** it creates. Once created, a service instance lives for the life of the module and Angular injects this one service instance in every class that needs it in this module.
- A **component's providers** (**@Component.providers**) are registered with **each component instance's own injector**.

Angular can only **inject** the corresponding services in **that component** instance or one of its descendant component instances. Angular cannot inject the same service instance anywhere else.

Note that a component-provided service may have a limited lifetime. Each new instance of the component gets its own instance of the service and, when the component instance is destroyed, so is that service instance.



Summary

There are three steps we need to take in order to perform an injection:

1. **Create the dependency** (e.g. the service class)
2. **Configure the injection** (i.e. register the injection)
3. **Declare the dependencies** on the receiving component

The first thing we do is creating the service class, that is, the class that exposes some behavior we want to use. This will be called the **injectable** because it is the thing that our components will receive via the injection.

A **provider** provides (creates, instantiates, etc.) the **injectable**. In Angular when you want to access the **injectable**, you **inject** a dependency into a component/service/module and Angular's dependency injection framework will locate it and provide it to you.



4

Http

HttpClient

Most **front-end applications** communicate with **backend services** over the **HTTP protocol**. Modern browsers support two different **APIs** for making HTTP requests: the **XMLHttpRequest** interface and the **fetch() API**.

The **HttpClient** in **@angular/common/http** offers a simplified client **HTTP API** for **Angular** applications that rests on the **XMLHttpRequest** interface exposed by browsers. Additional benefits of **HttpClient** include **testability features**, **typed request and response objects**, **request and response interception**, **Observable apis**, and **streamlined error handling**.



HttpClient (in „standalone component” approach)

By default, **HttpClient** uses the **XMLHttpRequest API** to make requests. The **withFetch** feature switches the client to use the **fetch API** instead.

HttpClient is provided using the **provideHttpClient** helper function, which most apps include in the application providers in **app.config.ts**.

```
TS app.config.ts x
1  import { ApplicationConfig, provideBrowserGlobalErrorListeners } from '@angular/core';
2  import { provideRouter } from '@angular/router';
3  import { routes } from './app.routes';
4  import {provideHttpClient, withFetch} from '@angular/common/http';
5
6  export const appConfig: ApplicationConfig = { new *
7    providers: [
8      provideBrowserGlobalErrorListeners(), provideRouter(routes), provideHttpClient(withFetch())
9    ]
10 };
```



HttpClient (in „modular” approach)

Before you can use the **HttpClient**, you need to **import** the Angular **HttpClientModule**. Most apps do so in the **root AppModule**.

```
app.module.ts
1 import { NgModule } from '@angular/core';
2 import { BrowserModule } from '@angular/platform-browser';
3 import { AppComponent } from './app.component';
4 import { HomeComponent } from './home/home.component';
5 import { StudentsComponent } from './students/students.component';
6 import { RouterModule, Routes } from '@angular/router';
7 import { HttpClientModule } from '@angular/common/http';
8 import { FormsModule } from '@angular/forms';
```



HttpClient (in „modular” approach)

Include **provideHttpClient** in the providers of your app's NgModule.

```
@NgModule({  
  > declarations: [ 7 elements... ],  
  > imports: [ 4 elements... ],  
  providers: [provideHttpClient()],  
  bootstrap: [AppComponent]  
})  
export class AppModule { }
```



HttpClient

Having imported **HttpClient**, you can **inject** the **HttpClient** into an application class as shown in the following **StudentService** example.

```
studentService.ts
1  import {inject, Injectable} from '@angular/core';
2  import {HttpClient, HttpHeaders} from '@angular/common/http';
3  import {catchError, Observable, of, tap} from 'rxjs';
4  import {Student} from '../models/student';
5
6  const httpOptions = {
7    headers: new HttpHeaders({ 'Content-Type': 'application/json' })
8  };
9
10 @Injectable({
11   providedIn: 'root',
12 })
13 export class StudentService {
14
15   private http : HttpClient = inject(HttpClient);
16
17   private studentsUrl : string = 'http://localhost:8080/students';
```



HttpClient

Http methods return **one value**. All **HttpClient** methods return an **RxJS Observable** of something.

HTTP is a **request/response** protocol. You make a request, it returns a single response.

In general, an **observable** can return **multiple values** over time. An **observable** from **HttpClient** always **emits a single value** and then completes, never to emit again.



HttpClient – handle errors

To **catch errors**, you "**pipe**" the **observable** result through an **RxJS catchError()** operator. Import the **catchError** symbol from **rxjs/operators**.

Extend the **observable** result with the **.pipe()** method and give it a **catchError()** operator. This operator **intercepts** an **Observable** that failed. It **passes the error** to an **error handler** that can do what it wants with the error.

```
studentService.ts ×
13   export class StudentService {
36     /** POST: add a new student to the server */
37     addStudent(student: Student): Observable<Student> {
38       return this.http.post<Student>(this.studentsUrl, student, httpOptions).pipe(
39         tap((studentAdded: Student) :void => this.log( message: `added student id=${studentAdded.id}`)),
40         catchError(this.handleError<Student>( operation: 'addStudent'))
41       );
42     }
}
```



HttpClient – handle errors

The following **handleError()** method **reports the error** and then returns an innocuous result so that the application keeps working.

The example **errorHandler()** will be shared by many **StudentService** methods so it is generalized to meet their different needs.

Instead of handling the error directly, it returns an **error handler function to catchError** that it has configured with both the name of the operation that failed and a safe return value.



HttpClient – handle errors

```
TS studentService.ts x
13 export class StudentService {
71     /**
72      * Handle Http operation that failed.
73      * Let the app continue.
74      * @param operation - name of the operation that failed
75      * @param result - optional value to return as the observable result
76      */
77     private handleError<T>(operation = 'operation', result?: T) : (error: any) => Observable<T> {
78         return (error: any): Observable<T> => {
79
80             // TODO: send the error to remote logging infrastructure
81             console.error(error); // log to console instead
82
83             // TODO: better job of transforming error for user consumption
84             this.log( message: `${operation} failed: ${error.message}`);
85
86             // Let the app keep running by returning an empty result.
87             return of(result as T);
88         };
89     }
90
91     /** Log a StudentService message with the MessageService */
92     private log(message: string) : void {
93         console.log('StudentService: ' + message);
94     }
}
```



HttpClient – Tap into the Observable

The **StudentService** methods will **tap** into the flow of **observable** values and send a message (via `log()`) to the console.

They will do that with the **RxJS tap operator**, which looks at the **observable values**, **does something** with those values, and **passes** them along. The **tap** call back **does not touch** the values themselves.

```
TS studentService.ts ×
13   export class StudentService {
36     /** POST: add a new student to the server */
37     addStudent(student: Student): Observable<Student> {
38       return this.http.post<Student>(this.studentsUrl, student, httpOptions).pipe(
39         tap((studentAdded: Student) :void => this.log( message: `added student id=${studentAdded.id}`)),
40         catchError(this.handleError<Student>( operation: 'addStudent'))
41       );
42     }
}
```



HttpClient – GET method

TS studentService.ts ×

```
10 @Injectable({
11   providedIn: 'root',
12 })
13 export class StudentService {
14
15   private http : HttpClient = inject(HttpClient);
16
17   private studentsUrl : string = 'http://localhost:8080/students';
18
19   /** GET students from the server */
20   getStudents(): Observable<Student[]> {
21     return this.http.get<Student[]>(this.studentsUrl).pipe(
22       tap((studentsList: Student[]) : void => this.log( message: `size of the list = ${studentsList.length}`)),
23       catchError(this.handleError<Student[]>( operation: 'getStudent all'))
24     );
25   }
```



HttpClient – GET method with parameter

Most web APIs support a **get by id** request in the form `api/student/:id` (such as `api/students/11`). Add a **`StudentService.getStudent()`** method to make that request.

```
studentService.ts x
13 export class StudentService {
27   /** GET student by id. Will 404 if id not found */
28   getStudent(id: number): Observable<Student> {
29     const url = `${this.studentsUrl}/${id}`;
30     return this.http.get<Student>(url).pipe(
31       tap(_ : Student => this.log( message: `fetched student id=${id}`)),
32       catchError(this.handleError<Student>( operation: `getStudent id=${id}`))
33     );
34   }
}
```

There are three significant differences from **`getStudents()`**:

- it constructs a request URL with the **desired student's id**,
- the server should respond with a **single student**,
- returns an **`Observable<Student>`**.



HttpClient – PUT method

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

- the URL,
- the data to update (the modified student in this case),
- options.

The **URL** is unchanged. The **students web API** knows which **student** to **update** by looking at the **student's id**.

The students web API expects a **special header** in HTTP save requests. That **header** is in the **httpOptions constant** defined in the **StudentService**.



HttpClient – PUT method

TS studentService.ts ×

```
13 export class StudentService {  
62   /** PUT: update the student on the server */  
63   updateStudent(student: Student, id:number): Observable<Student> {  
64     return this.http.put<Student>( url: `${this.studentsUrl}/${id}`, student, httpOptions).pipe(  
65       // tap(_ => this.log(`updated student id=${student.id}`)), // same as the line below  
66       tap((studentUpdated: Student) :void => this.log( message: `updated student id=${studentUpdated.id}`)),  
67       catchError(this.handleError<any>( operation: 'updateStudent'))  
68     );  
69   }
```

TS studentService.ts ×

```
6   const httpOptions = {  
7     headers: new HttpHeaders({ 'Content-Type': 'application/json' })  
8   };
```



HttpClient – POST method

```
ts studentService.ts x
13 export class StudentService {
36   /** POST: add a new student to the server */
37   addStudent(student: Student): Observable<Student> {
38     return this.http.post<Student>(this.studentsUrl, student, httpOptions).pipe(
39       tap((studentAdded: Student) :void => this.log( message: `added student id=${studentAdded.id}`)),
40       catchError(this.handleError<Student>( operation: 'addStudent'))
41     );
42   }
}
```

StudentService.addStudent() differs from **updateStudent()** in two ways:

- it calls **HttpClient.post()** instead of **put()**,
- it expects the **server to generate** an **id** for the new student, which it returns in the **Observable<Student>** to the caller.



HttpClient – DELETE method (one student)

TS studentService.ts ×

```
13 export class StudentService {  
44   /** DELETE: delete the student from the server */  
45   deleteStudent(student: Student | number): Observable<Student> {  
46     const id: number | undefined = typeof student === 'number' ? student : student.id;  
47     const url = `${this.studentsUrl}/${id}`;  
48     return this.http.delete<Student>(url, httpOptions).pipe(  
49       tap(_ : Student => this.log( message: `deleted student id=${id}`)),  
50       catchError(this.handleError<Student>( operation: 'deleteStudent'))  
51     );  
52   }
```



HttpClient – DELETE method (all students)

TS studentService.ts ×

```
13      export class StudentService {  
54          /** DELETE: delete all the students from the server */  
55          deleteStudents(): Observable<Student> {  
56              return this.http.delete<Student>(this.studentsUrl, httpOptions).pipe(  
57                  tap(_ : Student => this.log( message: `deleted students`)),  
58                  catchError(this.handleError<Student>( operation: 'deleteStudents'))  
59              );  
60          }
```



HttpClient – Student model class

Ts student.ts ×

```
1  export class Student {  
2  
3      id?: number;  
4      firstname: string;  
5      lastname: string;  
6      email: string;  
7      telephone: string;  
8  
9      constructor(firstname: string, lastname: string, email: string, telephone: string) {  
10         this.firstname = firstname;  
11         this.lastname = lastname;  
12         this.email = email;  
13         this.telephone = telephone;  
14     }  
15  
16 }
```



HttpClient – component

```
TS students.ts x
1 import {Component, inject, OnInit, signal} from '@angular/core';
2 import {Student} from '../models/student';
3 import {StudentService} from '../services/studentService';
4
5 @Component({
6   selector: 'app-students',
7   imports: [],
8   templateUrl: './students.html',
9   styleUrls: ['./students.css'],
10 })
11 export class Students implements OnInit {
12
13   private studentService : StudentService = inject(StudentService);
14
15   studentList : WritableSignal<Student[]> = signal<Student[]>([]);
16
17   ngOnInit() : void {
18     this.getStudents();
19   }
20
21   getStudents(): void {
22     this.studentService.getStudents().subscribe({
23       next: (newStudentList : Student[] ) : void => {
24         this.studentList.set(newStudentList);
25       },
26       error: () : void => {},
27       complete: () : void => {}
28     });
29   }
}
```



HttpClient – component

TS students.ts x

```
11 export class Students implements OnInit {  
31   add(firstname: string, lastname: string, email: string, telephone: string): void {  
32     firstname = firstname.trim();  
33     lastname = lastname.trim();  
34     email = email.trim();  
35     telephone = telephone.trim();  
36     this.studentService.addStudent({ firstname, lastname, email, telephone } as Student)  
37       .subscribe({  
38         next: (newStudent: Student) : void => {  
39           this.studentList.update((currentList: Student[]) : Student[] => [...currentList, newStudent]);  
40         },  
41         error: () : void => {},  
42         complete: () : void => {}  
43       });  
44   }
```



HttpClient – component

```
TS students.ts x
11 export class Students implements OnInit {
12
13     delete(student: Student): void {
14         this.studentService.deleteStudent(student).subscribe(): void => {
15             this.studentList.update(currentList : Student[] => currentList.filter(c : Student => c !== student));
16         }
17     };
18 }
19
20 deleteAll(): void {
21     this.studentService.deleteStudents().subscribe(): void => {
22         this.studentList.update(currentList : Student[] => {
23             console.log( ...data: `Deleting ${currentList.length} students.`);
24             return [];
25         });
26         // or use .set() method
27         // this.studentList.set([]);
28     }
29 };
30 }
```



HttpClient – component

TS students.ts x

```
11 export class Students implements OnInit {
65   update(firstname: string, lastname: string, email: string, telephone: string, chosenToUpdateStudent: Student): void {
66     let id: number | undefined = chosenToUpdateStudent.id;
67     firstname = firstname.trim();
68     lastname = lastname.trim();
69     email = email.trim();
70     telephone = telephone.trim();
71     console.log(id);
72     if (id !== undefined) {
73       this.studentService.updateStudent({firstname, lastname, email, telephone} as Student, id)
74         .subscribe({
75           next: (updatedStudent: Student) : void => {
76             this.studentList.update( currentStudents : Student[] =>
77               currentStudents.map(currentStudent : Student =>
78                 currentStudent.id === updatedStudent.id ? updatedStudent : currentStudent))
79             },
80           error: () : void => {
81             },
82           complete: () : void => {
83             }
84         })
85     }
86 }
```



HttpClient - .html template

```
<> students.html x
1 <h2>Students</h2>
2 <div>
3   <label>Student firstname:
4     <input #studentFirstName : HTMLInputElement />
5   </label>
6   <br>
7   <label>Student lastname:
8     <input #studentLastName : HTMLInputElement />
9   </label>
10  <br>
11  <label>Student email:
12    <input #studentEmail : HTMLInputElement />
13  </label>
14  <br>
15  <label>Student telephone:
16    <input #studentTelephone : HTMLInputElement />
17  </label>
18  <br>
19  <!-- (click) passes input value to add() and then clears the input -->
20  <button (click)="add(studentFirstName.value, studentLastName.value, studentEmail.value, studentTelephone.value);
21    studentFirstName.value=''; studentLastName.value=''; studentEmail.value=''; studentTelephone.value=''">
22    Add new student
23  </button>
24 </div>
```



HttpClient - .html template

<> students.html x

```
25 @for (student of studentList(); track student.id; let isFirst = $first; let isLast = $last){
26   @if (isFirst) { <h2>The list of students:</h2> }
27   <li>
28     {{student.id}} {{ student.firstname }} {{ student.lastname }} {{ student.email }} {{ student.telephone }}
29     <button title="delete" (click)="delete(student)">Delete</button>
30     <button title="update" (click)="update(studentFirstName.value,studentLastName.value, studentEmail.value, studentTelephone.value, student);
31       studentFirstName.value=''; studentLastName.value=''; studentEmail.value=''; studentTelephone.value=''">
32       Update
33     </button>
34   </li>
35   @if (isLast) { <br> <button title="delete all" (click)="deleteAll()">delete all</button> }
36 } @empty {
37   The collection is empty.
38 }
```



HttpClient – example

Router sample:

[HelloWorld](#)
[Calculator](#)
[Students](#)

Students

Student firstname:

Student lastname:

Student email:

Student telephone:

The list of students:

- 156 Rafał Kotas rkotas@dmcs.pl 123123123
- 160 Jan Kowalski jkowalski@email.com 987654321



// TODO (during next two laboratory classes):

- 1) implement routing in Angular application
- 2) implement and test six http methods from lecture example (please notice changes in backend application that simplifies the model – look into example code)
- 3) implement and test two „missing” http methods
(put all, patch)
- 4*) *ADDITIONAL: modify all methods to work with extended model by ORM*





THE END

WIKAMP → IWA_6.zip

