Aplikacja SPA: Angular + .NET Core Web API

Celem ćwiczeń jest zbudowanie aplikacji typu SPA (Single Page Application), której back-end będzie zaimplementowany na platformie .NET Core z wykorzystaniem Web API, a front-end przy użyciu frameworka Angular. Front-end i back-end będą uruchomione na odrębnych serwerach, w związku z czym do ich współpracy zostanie wykorzystany mechanizm CORS. Dane aplikacji po stronie back-endu będą przechowywane w pamięciowej bazie danych, do której dostęp będzie realizowany poprzez Entity Framework.

Do realizacji ćwiczeń potrzebne jest następujące oprogramowanie:

- .NET Core 2.0

- Visual Studio Code + C# extension

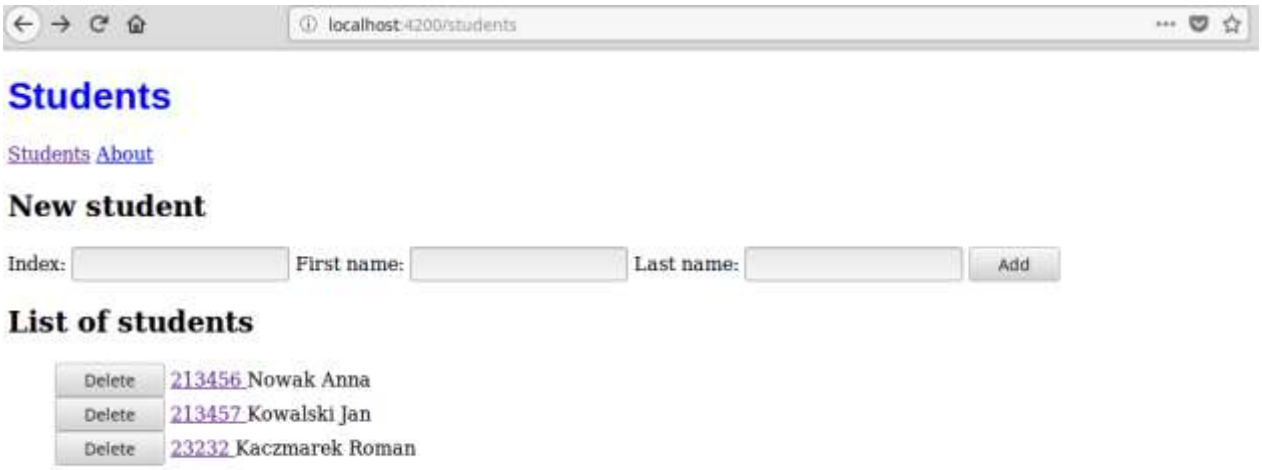
- node.js + npm

- Angular CLI

- Postman

Uwaga: Powyższe oprogramowanie jest już zainstalowane na maszynie wirtualnej dostępnej w laboratorium.

Docelowy wygląd aplikacji:



Ćwiczenie 1 (Back-end)

1. Otwórz okno terminala. Utwórz jako podkatalog katalogu domowego katalog dla projektu backendowego. Uczyń go katalogiem bieżącym:

mkdir StudentsApi

cd StudentsApi

2. Utwórz korzystając z interfejsu linii komend .NET Core projekt typu Web API:

dotnet new webapi

3. Uruchom środowisko Visual Studio Code i otwórz w nim folder utworzonego projektu.

4. Otwórz w VS Code plik Sturtup.cs do edycji.

Jeśli zostanie wyświetlony komunikat, że "Required assets to build and debug are missing...", to zleć ich dodanie.

5. Odszukaj plik zawierający klasę przykładowego kontrolera Web API utworzony przez kreator projektu. Obejrzyj kod tej klasy.

6. Uruchom aplikację (np. klawiszem Ctrl+F5 z VS Code lub komendą „dotnet run” z linii poleceń).

7. W przeglądarce wprowadź adres http://localhost:5000/api/values aby sprawdzić czy aplikacja działa.

8. Utwórz w projekcie folder Models (na tym samym poziomie co istniejący folder Controllers).

9. W folderze Models utwórz nowy plik Student.cs i umieść w nim poniższą definicję klasy Student:

```
namespace StudentsApi.Models
{
   public class Student
   {
      public long Id { get; set; }
      public int Index { get; set; }
      public string FirstName { get; set; }
      public string LastName { get; set; }
   }
}
```

10. W folderze Models utwórz klasę kontekstu bazy danych (w pliku StudentContext.cs) o następującej treści:

```
using Microsoft.EntityFrameworkCore;

namespace StudentsApi.Models
{
   public class StudentContext : DbContext
   {
      public StudentContext(DbContextOptions<StudentContext> options)
         : base(options)
      {
      }

      public DbSet<Student> Students { get; set; }

   }
}
```

11. Zarejestruj kontekst bazy danych w kontenerze wstrzykiwania zależności, tak aby można go było wstrzykiwać do kontrolerów. W tym celu przejdź do edycji pliku Startup.cs i dokonaj w nim poniższych zmian:

a) Usuń właściwość Configuration i konstruktor klasy

b) W ciele metody ConfigureServices jako pierwszy wiersz dodaj:

```
services.AddDbContext<StudentContext>(opt=>opt.UseInMemoryDatabase("StudentList"));
```

Ustawiona opcja zleca wykorzystanie pamięciowej bazy danych do przechowywania danych poprzez zdefiniowany kontekst bazodanowy. Wykorzystywany dostawca bazy danych Entity Framework Core InMemory database nie musi być dodatkowo instalowany, gdyż jest automatycznie dostępny dzięki obecności referencji

```
<PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.3" />
```

w pliku projektu *.cs.proj.

c) Metodę Configure zastąp poniższą uproszczoną implementacją:

```
public void Configure(IApplicationBuilder app)
{
   app.UseMvc();
}
```

d) Dodaj poniższe instrukcje using:

```
using Microsoft.EntityFrameworkCore;
using StudentsApi.Models;
```

12. W folderze Controllers utwórz plik z kontrolerem StudentController o następującej treści:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using StudentsApi.Models;

namespace StudentApi.Controllers
{
   [Route("api/[controller]")]
   public class StudentController : Controller
   {
      private readonly StudentContext _context;

      public StudentController(StudentContext context)
      {
         _context = context;
         InitializeMemoryDatabase();
      }

      private void InitializeMemoryDatabase()
      {
         if (_context.Students.Count() == 0)
         {
            _context.Students.Add(new Student { Index = 213456,
FirstName = "Anna", LastName = "Nowak" });
            _context.Students.Add(new Student { Index = 213457,
FirstName = "Jan", LastName = "Kowalski" });
            _context.SaveChanges();
         }
      }
   }
}
```

Zwróć uwagę na atrybut określający ścieżkę która będzie wywoływać utworzony kontroler oraz na wstrzyknięcie obiektu kontekstu bazy danych.

Aby aplikacja już przy pierwszym uruchomieniu prezentowała jakieś dane, w kontrolerze została dodana pomocnicza metoda (wywołana w konstruktorze), która dodaje dwa obiekty do pamięciowej bazy danych.

13. Dodaj w kontrolerze dwie poniższe metody obsługujące żądania GET, odpowiednio pobierające wszystkich studentów i studenta o podanym identyfikatorze:

```
[HttpGet]
public IEnumerable<Student> GetAll()
{
   return _context.Students.ToList();
}

[HttpGet("{id}", Name = "GetStudent")]
public IActionResult GetById(long id)
{
   var student = _context.Students.FirstOrDefault(s => s.Id == id);
   if (student == null)
   {
      return NotFound();
   }
   return new ObjectResult(student);
}
```

Zwróć uwagę na:

- Dostęp w metodzie kontrolera do przekazywanego identyfikatora
- Nazwę ścieżki wywołującej drugą z metod. Taka nazwana ścieżka ułatwi nam później generowanie linku do zasobu.

14. Zapisz wszystkie zmiany. Zatrzymaj aplikację i uruchom ją ponownie (np. klawiszem Ctrl+F5).

15. Przetestuj z poziomu przeglądarki działanie adresów:

http://localhost:5000/api/student

http://localhost:5000/api/student/1

16. Utwórz w kontrolerze StudentController poniższą metodę do tworzenia nowej instancji zasobu:

```
[HttpPost]
public IActionResult Create([FromBody] Student student)
{
   if (student == null)
   {
      return BadRequest();
   }

   _context.Students.Add(student);
   _context.SaveChanges();

   return CreatedAtRoute("GetStudent", new { id = student.Id }, student);
}
```

Zwróć uwagę na:

- Atrybut [FromBody], który specyfikuje że dane nowego studenta będą pobierane z ciała żądania HTTP.
- Sposób zwracania wyniku z pomocą metody CreatedAtRoute, aby odpowiedź dla klienta zawierała nagłówek Location z adresem nowo utworzonego zasobu. Dzięki temu klient będzie mógł uzyskać dostęp do identyfikatora (Id), który będzie automatycznie nadany w bazie danych. Do generacji adresu wykorzystana została nazwa ścieżki prowadzącej do metody pobierającej studenta o podanym identyfikatorze.

17. Do przetestowania działania metody POST (a później również PUT i DELETE) wykorzystamy narzędzie Postman. Można je uruchomić w przygotowanym środowisku z poziomu terminala (otwórz nowe okno) komendą postman.

18. Pracę z narzędziem Postman zaczniemy od sprawdzenia działania metody GET. W pasku adresu głównego okna narzędzia wprowadź adres: http://localhost:5000/api/student i kliknij przycisk Send. Obejrzyj odpowiedź serwera (ciało i nagłówki).

19. W narzędziu Postman zmień metodę HTTP na POST. Jako typ zawartości ciała żądania wybierz „raw body”, a jako treść wprowadź:

```
{"index":213458,"firstName":"Zenon","lastName":"Zawada"}
```

Zmień typ przesyłanej zawartości na application/json i kliknij Send.

Obejrzyj nagłówki odpowiedzi serwera zwracając szczególną uwagę na Status i Location.

20. Dodaj w klasie kontrolera StudentController poniższe metody obsługujące żądania PUT i DELETE:

```
[HttpPut("{id}")]
public IActionResult Update(long id, [FromBody] Student student)
{
   if (student == null || student.Id != id)
   {
      return BadRequest();
   }

   var studentToUpdate = _context.Students.FirstOrDefault(s =>
s.Id == id);
   if (studentToUpdate == null)
   {
      return NotFound();
   }

   studentToUpdate.Index = student.Index;
   studentToUpdate.FirstName = student.FirstName;
   studentToUpdate.LastName = student.LastName;

   _context.Students.Update(studentToUpdate);
   _context.SaveChanges();
   return new NoContentResult();
}

[HttpDelete("{id}")]
public IActionResult Delete(long id)
```

```
    {
        var studentToDelete = _context.Students.FirstOrDefault(s =>
s.Id == id);
        if (studentToDelete == null)
        {
            return NotFound();
        }

        _context.Students.Remove(studentToDelete);
        _context.SaveChanges();
        return new NoContentResult();
    }
```

21. Przetestuj dodane operacje narzędziem Postman:
- zmień imię studentowi o Id = 1
- usuń studenta o Id = 2
- pobierz dane wszystkich studentów

22. Zatrzymaj aplikację i zamknij folder projektu w VS Code. Zamknij Postmana.
23. Uruchom aplikację z linii komend:
dotnet run
24. Przetestuj z poziomu przeglądarki czy aplikacja działa i pozostaw ją uruchomioną.

Ćwiczenie 2 (Front-end)

1. Otwórz nowe okno poleceń (lub wykorzystaj okno po Postmanie).

2. Sprawdź poniższymi komendami czy zainstalowane są potrzebne narzędzia:
node -v
npm -v
ng version
Jeśli brakuje Angular CLI, doinstaluj to oprogramowanie komendą:
npm install -g @angular/cli
Jeśli narzędzie ng nie znajduje się na ścieżce PATH systemu operacyjnego, dodaj jego katalog domowy do ścieżki PATH:

set PATH=%APPDATA%\npm;%PATH%

(składnia dla Windows)

3. Z poziomu katalogu domowego wydaj komendę Angular CLI tworzącą nowy projekt:

ng new StudentsFront

Pojawią się 2 pytania dotyczące konfiguracji projektu:

- Spraw aby dodany został Angular routing
- Wybierz CSS jako format arkuszy stylów

4. Zmień bieżący katalog w terminalu na katalog utworzonego przed chwilą projektu:

cd StudentsFront

5. Uruchom aplikację Angular:

ng serve –open

Opcja „--open” powinna otworzyć stronę w przeglądarce:

http://localhost:4200/

6. Otwórz folder projektu front-endowego w VS Code.

7. Obejrzyj zawartość pliku konfiguracyjnego projektu package.json zwracając uwagę na wersję frameworka i języka TypeScript.

8. W panelu eksploratora plików projektu rozwiń gałąź src/app

9. Obejrzyj klasę modułu w pliku app.module.ts zwracając uwagę na odwołania do komponentu AppComponent w dekoratorze którym opatrzona jest klasa modułu.

10. Otwórz plik index.html i zwróć uwagę na niestandardowy element HTML, który wskazuje miejsce zagnieżdżenia na stronie głównego komponentu aplikacji.

11. Otwórz pliki ts, html i css komponentu AppComponent. Odszukaj definicję nazwy znacznika (selektora), który reprezentuje ten komponent w HTML-u.

12. Zmień w klasie komponentu tytuł na "Students"

13. Całą zawartość szablonu komponentu (*.html) zastąp poniższym wierszem:

<h1>{{title}}</h1>

14. W arkuszu stylów komponentu umieść regułę CSS:

```
h1 {
   color: blue;
}
```

15. Otwórz globalny arkusz stylów aplikacji (styles.css) i umieść w nim regułę:

```
h1 {
   font-family: Arial, Helvetica, sans-serif;
   font-size: 200%;
  }
```

16. Z poziomu linii komend będąc w katalogu głównym projektu frontendowego utwórz nowy komponent Angulara:

```
ng generate component students
```

17. Obejrzyj w VS Code kod wygenerowanego komponentu. Zauważ też, że został on automatycznie uwzględniony w głównym pliku modułu (app.module.ts).

18. W folderze src/app utwórz plik student.ts i umieść w nim poniższy kod klasy Student:

```
export class Student {
   id: number;
   index: number;
   firstName: string;
   lastName: string;
  }
```

Jest to klasa modelu stanowiąca odpowiednik klasy modelu po stronie back-endu.

19. Przejdź do edycji klasy komponentu Students i dodaj w jej pliku instrukcję import:

```
import { Student } from '../student';
```

20. W ciele klasy StudentsComponent umieść definicję tablicy obiektów klasy Student:

```
  students: Student[] = [
   {id: 1, index: 123456, firstName: 'Marek', lastName: 'Wojciechowski'},
```

```
  {id: 2, index: 123457, firstName: 'Krzysztof', lastName: 'Jankiewicz'},
 ]
```

21. Jako szablon komponentu StudentsComponent wprowadź:

```
<h2>List of students</h2>
<ul>
 <li *ngFor="let student of students">
  <span>{{student.index}}</span>
  <span>{{student.lastName}}</span>
  <span>{{student.firstName}}</span>
 </li>
</ul>
```

22. Na końcu szablonu komponentu AppComponent dodaj znacznik:

```
<app-students></app-students>
```

23. Obejrzyj aplikację front-endową w przeglądarce. (Serwer jest uruchomiony w trybie deweloperskim, aplikacja jest na bieżąco budowana i odświeżana w przeglądarce.)

24. Utwórz w projekcie front-endowym klasę usługową (service) korzystając z Angular CLI:

```
ng generate service student
```

25. Przenieś definicję tablicy studentów z klasy komponentu do klasy serwisu.

26. Dodaj w klasie serwisu import klasy Student.

27. Dodaj w klasie serwisu poniższą metodę udostępniającą studentów:

```
 getStudents(): Student[] {
  return this.students;
 }
```

28. Aby usługa była dostępna dla mechanizmu wstrzykiwania zależności Angulara, dodaj ją do tablicy providers modułu (app.module.ts):

...
providers: [StudentService]
...

29. Dodaj odpowiednią instrukcję import w pliku klasy modułu.

30. Przejdź do edycji klasy komponentu StudentsComponent i dokonaj następujących zmian
a) dodaj import:
import { StudentService } from '../student.service';
b) dodaj w klasie właściwość:
students: Student[];
c) wstrzyknij usługę poprzez parametr konstruktora:
constructor(private studentService: StudentService) { }
Taki sposób deklaracji parametru konstruktora (z kwalifikatorem widzialności)
jednocześnie tworzy prywatną właściwość w klasie.

31. Dodaj w klasie komponentu poniższą metodę pobierającą dane studentów z usługi:

```
getStudents(): void {
  this.students = this.studentService.getStudents();
}
```

32. Wywołaj powyższą metodę w ciele metody ngOnInit() komponentu.

33. Zapisz wszystkie zmiany i sprawdź czy aplikacja w przeglądarce nadal prezentuje listę z danymi studentów.

34. Ponieważ docelowo dane będą pobierane żądaniem HTTP, zmienimy mechanizm pobierania danych przez serwis na asynchroniczny, wykorzystując do tego celu Observable z biblioteki RxJS. W tym celu:

a) Dodaj importy do klasy serwisu:
import { Observable } from 'rxjs';
import { of } from 'rxjs';
b) Zmień implementację by zwracała Observable emitujące pojedynczą wartość - tablicę studentów (w takiej formie zwróci dane klient HTTP Angulara):

```
getStudents(): Observable<Student[]> {
```

```
  return of(this.students);
 }
```

35. Dostosuj do dokonanych zmian w usłudze metodę pobierającą dane z usługi w komponencie (subskrypcja ze wskazaniem callbacka):

```
 getStudents(): void {
  this.studentService.getStudents()
  .subscribe(students => this.students = students);
 }
```

36. Zapisz zmiany i sprawdź działanie aplikacji w przeglądarce.

37. Przejdź do edycji pliku modułu app.module.ts i dokonaj następujących zmian:

a) dodaj poniższy import:

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

b) Do tablicy imports modułu dopisz:

HttpClientModule

38. W klasie usługi:

a) Dodaj import:

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

b) Wstrzyknij jako parametr konstruktora klasy serwisu:

private http: HttpClient

c) Dodaj prywatne pole:

private studentsApiUrl = 'http://localhost:5000/api/student';

d) Zmień wartość zwrotną metody getStudents() na

this.http.get<Student[]>(this.studentsApiUrl)

e) Usuń niepotrzebną już tablicę ze studentami.

39. Zapisz zmiany i obejrzyj ich efekt w przeglądarce. Czy dane pobrały się z back-endu?

40. Spróbuj zdiagnozować problem wyświetlając konsolę przeglądarki. Powinien być w niej pokazany błąd naruszenia zasady „Same Origin Policy”.

41. Aby front-end uruchomiony na innym serwerze (wystarcza inny port!) mógł skutecznie komunikować się z back-endem trzeba będzie skonfigurować mechanizm CORS na poziomie aplikacji wystawiającej API. W tym celu:

a) Zatrzymaj aplikację back-endową (Ctrl+C)

b) Otwórz do edycji plik Startup.cs (w dowolnym edytorze tekstowym - nie musimy dla tak drobnej modyfikacji otwierać całego projektu w IDE).

c) W metodzie ConfigureServices przed instrukcją

```
services.AddMvc();
```

dodaj instrukcję

```
services.AddCors();
```

d) W metodzie Configure przed instrukcją

```
app.UseMvc();
```

dodaj instrukcję

```
app.UseCors( options =>
options.WithOrigins("http://localhost:4200").AllowAnyMethod().AllowAnyHeader());
```

e) Zapisz zmiany i uruchom z linii komend aplikację back-endową:

```
dotnet run
```

42. Sprawdź czy teraz aplikacja front-endowa wyświetla studentów pobranych z back-endu.

43. Wróć do pracy nad projektem front-endowym i dodaj w klasie usługowej poniższą metodę:

```
getStudent(id: number): Observable<Student> {
  const url = `${this.studentsApiUrl}/${id}`;
  return this.http.get<Student>(url);
}
```

44. Metody obsługujące inne żądania niż GET będą wysyłały nagłówek Content-Type w żądaniu. Aby ułatwić sobie implementację tych metod zdefiniuj w pliku z klasą usługową (przed definicją klasy) poniższą stałą:

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

45. Dodaj w klasie usługowej poniższe brakujące metody:

```
updateStudent(student: Student): Observable<any> {
  const url = `${this.studentsApiUrl}/${student.id}`;
  return this.http.put(url, student, httpOptions);
```

```
  }

  createStudent(student: Student): Observable<Student> {
    return this.http.post<Student>(this.studentsApiUrl, student, httpOptions);
  }

  deleteStudent(student: Student | number): Observable<Student> {
    const id = typeof student === 'number' ? student : student.id;
    const url = `${this.studentsApiUrl}/${id}`;
    return this.http.delete<Student>(url, httpOptions);
  }
```

Przeanalizuj kod dopisanych metod.

Uwaga: W rzeczywistej aplikacji należałoby jeszcze oprogramować obsługę błędów komunikacji.

46. Na początku szablonu komponentu z listą studentów dodaj poniższą formatkę do dodawania nowego studenta:

```
<h2>New student</h2>
<div>
  <label>Index:
    <input #studIndex />
  </label>
  <label>First name:
    <input #studFirstName />
  </label>
  <label>Last name:
    <input #studLastName />
  </label>
  <button (click)="create(studIndex.value, studFirstName.value,
studLastName.value); studIndex.value=''; studFirstName.value='';
studLastName.value=''">
    Add
  </button>
</div>
```

47. W klasie komponentu dopisz metodę create(), która podpięta jest pod zdarzenie click dodanego przed chwilą przycisku:

```
 create(index: number, firstName: string, lastName: string): void {
   this.studentService.createStudent({ index: index, firstName:
firstName, lastName: lastName } as Student)
     .subscribe(student => {
       this.students.push(student);
     });
 }
```

48. W szablonie przed wyświetlanym indeksem studenckim, dodaj przycisk do usuwania danych:

```
   <button title="Delete"
     (click)="delete(student)">Delete</button>
```

49. Dodaj w klasie komponentu metodę na którą powołuje się dodany przycisk:

```
 delete(student: Student): void {
   this.students = this.students.filter(s => s.id !== student.id);
   this.studentService.deleteStudent(student).subscribe();
 }
```

50. Przetestuj w przeglądarce dodawania i usuwanie studentów.

Włącz monitor aktywności sieciowej (Ctrl+Shift+E w Firefox), aby zobaczyć że w ramach mechanizmu CORS żądania POST i DELETE (również PUT) poprzedzane są żądaniem OPTION (preflight request).

51. Po dodaniu przycisku do usuwania pozycji listy studentów niezręcznie wygląda kropka rozpoczynająca pozycję listy. Wykorzystaj arkusz stylów komponentu by lista była wyświetlana bez punktorów (klasą CSS).

52. Edycję danych studentów zrealizujemy w komponencie, który będzie zastępował na stronie komponent z listą studentów. Aby możliwa była nawigacja między komponentami współdzielącymi ten sam obszar na stronie, skonfigurujemy routing.

Rozpoczniemy od dodania do projektu modułu, który będzie pełnił rolę modułu routingu:

ng generate module app-routing --flat --module=app

Wyjaśnienie składni komendy:

Opcja --flat umieszcza kod komponentu bezpośrednio w katalogu src/app, a nie w dedykowanym podkatalogu.

Opcja --module=app zleca rejestrację tworzonego komponentu w tablicy imports modułu AppModule.

53. Ponieważ moduły routingu nie korzystają z komponentów, usuń z dekoratora klasy modułu tablicę declarations i odwołania do CommonModule (z tablicy imports i instrukcję import też):

```
@NgModule({
  imports: []
})
export class AppRoutingModule { }
```

54. Dodaj w dekoratorze modułu routing tablicę exports udostępniającą RouterModule komponentom, które go będą wymagały. Po zmianach kod modułu routingu powinien wyglądać następująco:

```
import { NgModule }             from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

@NgModule({
  exports: [ RouterModule ]
})
export class AppRoutingModule {}
```

55. Dodaj w module routingu trasę (route) prowadzącą do komponentu z listą studentów. W tym celu:

a) Zaimportuj komponent:

```
import { StudentsComponent }      from './students/students.component';
```

b) Zdefiniuj tablicę tras jako stałą (powyżej klasy):

```
const routes: Routes = [
  { path: 'students', component: StudentsComponent }
];
```

56. Zaicjalizuj router, aby nasłuchiwał zmian adresu w przeglądarce. Dodaj RouterModule do tablicy imports w dekoratorze klasy modułu routingu w poniższy sposób:

```
imports: [ RouterModule.forRoot(routes) ],
```

57. W szablonie komponentu AppComponent zastąp znacznik reprezentujący komponent z listą studentów, poniższym znacznikiem reprezentującym miejsce na stronie, gdzie router będzie podmieniał komponenty zależnie od aktualnej ścieżki adresu:

<router-outlet></router-outlet>

58. Sprawdź działanie aplikacji w przeglądarce. Lista studentów pokaże się dopiero po ręcznej zmianie adresu w pasku adresu na:

http://localhost:4200/students

59. Dodaj poniższe linki przed elementem <router-outlet> (drugi nie będzie na razie działał):

```
<nav>
  <a routerLink="/students">Students</a>
  <a routerLink="/about">About</a>
</nav>
```

60. Sprawdź w przeglądarce działanie pierwszego z linków.

61. Zdefiniuj domyślną ścieżkę routing, tak aby po wejściu na adres startowy aplikacji od razu pojawiła się w przeglądarce lista studentów. Dodaj w tym celu poniższą ścieżkę do tablicy AppRoutingModule.Routes.

{ path: '', redirectTo: '/students', pathMatch: 'full' },

62. Sprawdź w przeglądarce zachowanie się aplikacji po wejściu na stronę http://localhost:4200

63. Wygeneruj komponent, który docelowo będzie prezentował szczegóły konkretnego studenta z możliwością ich edycji:

ng generate component student-detail

64. Dodaj w klasie nowego komponentu pole, które będzie zawierało studenta do edycji:

student: Student;

Uzupełnij import klasy Student.

65. W tej samej klasie dodaj kolejne importy:

import { ActivatedRoute } from '@angular/router';

import { Location } from '@angular/common';

```
import { StudentService }  from '../student.service';
```

66. Wstrzyknij poprzez konstruktor (w tej samej klasie) obiekty niezbędne do identyfikacji edytowanego studenta, skorzystania z API i programowego powrotu do poprzedniego ekranu:

```
constructor(
  private route: ActivatedRoute,
  private studentService: StudentService,
  private location: Location
) {}
```

67. Dodaj w komponencie ze szczegółami metodę do pobrania danych studenta wskazanego ścieżką adresu, która wywołała component:

```
  getStudent(): void {
    const id = +this.route.snapshot.paramMap.get('id');
    this.studentService.getStudent(id)
      .subscribe(student => this.student = student);
  }
```

68. Wywołaj powyższą metodę w metodzie ngOnInit():

```
  ngOnInit() {
    this.getStudent();
  }
```

69. Dodaj metodę do powrotu do poprzedniego ekranu:

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

70. Dodaj metodę do zapisu edytowanego studenta:

```
 save(): void {
    this.studentService.updateStudent(this.student)
      .subscribe(() => this.goBack());
  }
```

71. Jako szablon komponentu wklej poniższy kod:

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

```
  <h2>{{ student.firstName}} {{ student.lastName | uppercase }}
details</h2>
  <div><span>Id: </span>{{student.id}}</div>
  <div>
    <label>Index:
      <input [(ngModel)]="student.index" placeholder="index"/>
     </label>
     <label>First name:
       <input [(ngModel)]="student.firstName" placeholder="first name"/>
     </label>
       <label>Last name:
    <input [(ngModel)]="student.lastName" placeholder="last name"/>
   </label>
  </div>
  <button (click)="goBack()">Back</button>
  <button (click)="save()">Save</button>
 </div>
```

Zwróć uwagę na:

- Mechanizm zamiany wielkości liter (pipe)
- Odwołanie do [(ngModel)], czyli mechanizmu dwukierunkowego wiązania danych Angulara.

72. Aby mechanizm ngModel był dostępny w aplikacji, dokonaj poniższych uzupełnień w kodzie głównego modułu aplikacji (app.module.ts):

a) Dodaj import:

```
import { FormsModule } from '@angular/forms';
```

b) Dodaj FormsModule do tablicy imports.

73. Do tablicy Routes w module routing dodaj ścieżkę wskazującą studenta do edycji:

```
{ path: 'detail/:id', component: StudentDetailComponent },
```

Dodaj wymagany import.

74. W szablonie komponentu z listą linię:

```
<span>{{student.index}}</span>
```

zastąp przez:

```
<a routerLink="/detail/{{student.id}}">
  <span>{{student.index}}</span>
</a>
```

75. Sprawdź nawigację do szczegółów i z powrotem oraz edycję danych.

76. Sprawdź też zachowanie się aplikacji dla ręcznie wpisanych adresów:

http://localhost:4200/detail/1

http://localhost:4200/detail/12

Ćwiczenie do samodzielnego wykonania

1. Zdefiniuj w projekcie front-endowym komponent AboutComponent, który będzie prezentował:
   a. Nagłówek <h1> z tekstem „About”
   b. Nagłówek <h3> z tekstem „Angular tutorial completed on …” zawierający bieżącą datę i czas (można wykorzystać standardowy obiekt Date języka JavaScript)

2. Powiąż zdefiniowany komponent z utworzonym wcześniej linkiem „About”