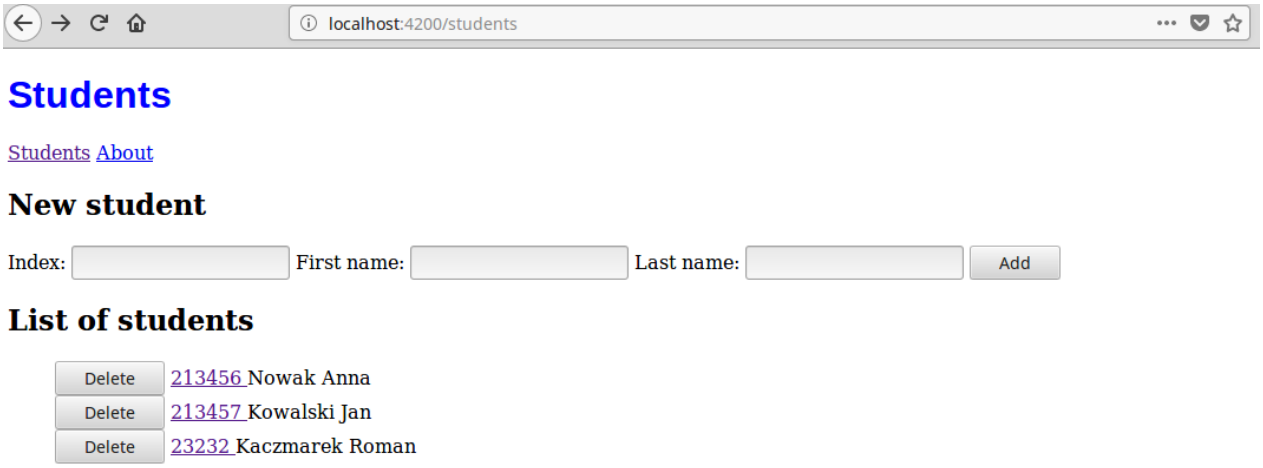
Aplikacja SPA: Angular + .NET Core Web API

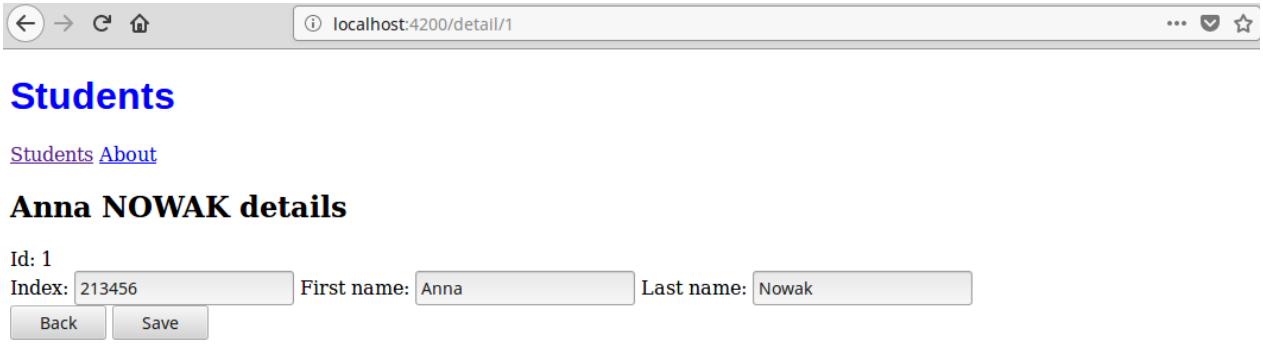
Celem ćwiczeń jest zbudowanie aplikacji typu SPA (Single Page Application), której back-end będzie zaimplementowany na platformie .NET Core z wykorzystaniem ASP.NET Web API, a front-end przy użyciu frameworka Angular. Front-end i back-end będą uruchomione na odrębnych serwerach, w związku z czym do ich 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 SDK 6.0 lub nowsze (może być zainstalowane z Visual Studio lub samodzielnie)
- Visual Studio Code + C# extension (instalowany z poziomu panelu Extensions w VS Code)
- node.js + npm (npm instaluje się wraz z node.js)
- Angular CLI (instalowane z poziomu npm)
- Postman

Docelowy wygląd aplikacji:





Ćwiczenie 1 (Back-end)

1. Otwórz okno terminala (cmd w Windows). Utwórz katalog dla projektu back-endowego. Uczyń go katalogiem bieżącym:

```
mkdir StudentsApi
cd StudentsApi
```

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

```
dotnet new webapi
```

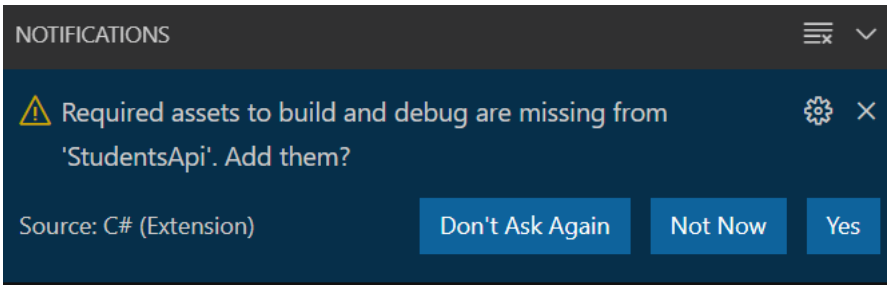
3. Z poziomu interfejsu linii komend .NET wykonaj dwa poniższe polecenia w celu dodania wymaganych pakietów do projektu:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.InMemory
```

Komentarz: Pierwszy pakiet zawiera Entity Framework (wraz ze wsparciem dla MS SQL Server, z którego w projekcie nie skorzystamy). Drugi dodaje pamięciową bazę danych, której użyjemy zamiast prawdziwej bazy danych, aby uprościć konfigurację środowiska.

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

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



5. Otwórz plik projektu (*.csproj) w VS Code i odszukaj w nim referencje do dwóch pakietów, które zostały dodane do projektu.

6. Otwórz w VS Code plik Program.cs do edycji. Odszukaj instrukcję włączającą przekierowanie do HTTPS i zakomentuj ją.

```
//app.UseHttpsRedirection();
```

Komentarz: Przy włączonych przekierowaniach do HTTPS, po odwołaniu się do API protokołem HTTP nastąpiłoby przekierowanie do HTTPS i przeglądarka poskarżyłaby się na niewłaściwy certyfikat. .NET dostarcza deweloperski certyfikat, którego przeglądarki nie uznają za godny zaufania.

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

8. Uruchom aplikację (kombinacją klawiszy Ctrl+F5 z poziomu VS Code lub komendą `dotnet run` z linii poleceń).

Uwaga: Nie przejmuj się gdy adres URI wykorzystujący HTTPS zostanie automatycznie otwarty w przeglądarce (ma to miejsce gdy aplikacja jest uruchamiana z poziomu VS Code), skutkując ostrzeżeniem o niebezpieczeństwie. Będziemy ręcznie podmieniać domyślny, problematyczny adres URI.

9. W przeglądarce wprowadź poniższy adres URI aby sprawdzić czy aplikacja działa (ewentualnie popraw numer portu, jeśli twój serwer wykorzystuje inny port niż 5000 do nasłuchu HTTP):

`http://localhost:5000/weatherforecast`

10. Zatrzymaj aplikację (kombinacją klawiszy Shift+F5 w VS Code lub Ctrl-C z poziomu wiersza poleceń, zależnie od sposobu uruchomienia aplikacji).

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

12. 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; }
    }
}
```

13. 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; }

    }
```

```
}
```

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

a) Przed instrukcją tworzącą aplikację wstaw:

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

Ustawiona opcja zleca wykorzystanie pamięciowej bazy danych do przechowywania danych poprzez zdefiniowany kontekst bazodanowy.

b) Dodaj na początku poniższe instrukcje using:

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

15. Ponieważ planujemy wykorzystać scaffolding do generacji kontrolera API obsługującego dane studentów, najpierw musimy dodać klika pakietów i narzędzi. Wróć do linii komend .NET i wykonaj następujące kroki:

a) Dodaj dwa kolejne pakiety do projektu:

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
```

b) Zainstaluj generator kodu obsługujący scaffolding:

```
dotnet tool install --global dotnet-aspnet-codegenerator
```

16. Pozostając w wierszu komend, wygeneruj konstruktor StudentsController na podstawie klasy modelu Student (wszystkie opcje podaj w jednej linii):

```
dotnet aspnet-codegenerator controller -name StudentsController
-async -api -m Student -dc StudentContext -outDir Controllers
```

17. Wróć do VS Code i otwórz wygenerowaną klasę kontrolera StudentController, aby przeanalizować jej kod:

a) Odszukaj metody obsługujące żądania GET, PUT, POST i DELETE.

b) Wskaż atrybuty routingu definiujące adresy URI prowadzące do poszczególnych metod.

c) Zlokalizuj instrukcję zwracającą wynik działania metody obsługującej żądania POST, aby zobaczyć w jaki sposób ASP.NET zwraca klientowi adres URI nowo utworzonej instancji zasobu. Zwróć uwagę na odniesienie do metody GetStudent.

d) Znajdź miejsce wstrzyknięcia kontekstu bazy danych do kontrolera.

18. Zapisz wszystkie zmiany. Zatrzymaj aplikację jeśli aktualnie jest uruchomiona. Uruchom aplikację z poziomu linii komend (Tym razem istotne jest uruchomienie aplikacji z poziomu linii komend, aby można było zamknąć projekt w VS Code pozostawiając aplikację działającą.):

```
dotnet run
```

19. Sprawdź z poziomu przeglądarki czy poniższy adres URI działa (ewentualnie popraw numer portu na właściwy):

```
http://localhost:5000/api/students
```

Uwaga: Odpowiedzą z API powinna być pusta tablica w formacie JSON, w związku z tym, że baza danych jest pusta.

20. Uruchom narzędzie Postman, którego będziemy używać do przetestowania API i jednocześnie wypełnienia bazy danych danymi.

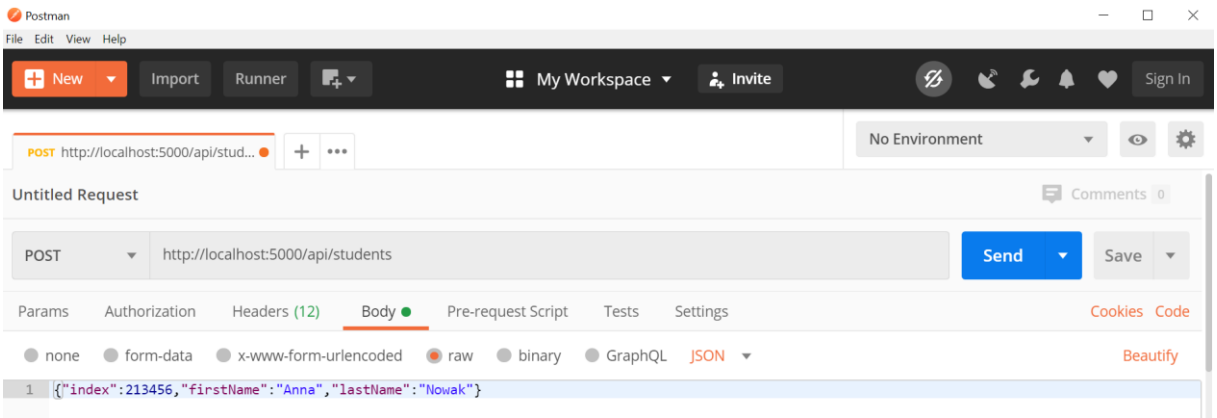
21. Sesję w narzędziu Postman rozpoczniemy od żądania GET. W polu adresu narzędzia wprowadź (ewentualnie poprawiając nr portu): `http://localhost:5000/api/students` i kliknij Send. Przeanalizuj odpowiedź serwera (ciało i nagłówki).

Uwaga: Znowu oczekujemy pustej tablicy w formacie JSON.

22. Pozostając w narzędziu Postman zmień metodę HTTP na POST. Upewnij się, że nagłówek Content-Type żądania jest ustawiony na application/json. Przełącz się na zakładkę body I wybierz "raw" jako typ ciała żądania. Jako treść ciała wprowadź:

```
{"index":213456,"firstName":"Anna","lastName":"Nowak"}
```

Kliknij Send.



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

23. W ten sam sposób (jako dwa oddzielne żądania) prześlij metodą POST dane kolejnych dwóch studentów:

```
{"index":213457,"firstName":"Jan","lastName":"Kowalski"}
```

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

24. Aby zweryfikować, że wszystkie dane zostały poprawnie zapamiętane po stronie back-endu, wyślij następujące dwa żądania GET do API za pomocą Postmana:

a) pobierz wszystkich studentów

b) pobierz studenta o id = 1

25. Sprawdź za pomocą Postmana czy metody PUT i DELETE również działają poprawnie:

a) zmień imię studenta o id = 2

b) usuń studenta o id = 3

c) pobierz wszystkich studentów

26. Zamknij folder projektu w VS Code. Pozostaw aplikację działającą.

Ćwiczenie 2 (Front-end)

1. Otwórz nowe okno poleceń.

2. Sprawdź poniższymi komendami czy zainstalowane są potrzebne narzędzia:

```
node -v
npm -v
ng version
```

Jeśli ostatnia komenda zakończy się niepowodzeniem z powodu niezainstalowania Angular CLI, doinstaluj to oprogramowanie komendą:

```
npm install -g @angular/cli
```

Jeśli Angular CLI jest zainstalowane, ale 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 kolejno następujące pytania dotyczące konfiguracji projektu:

- Czy ma być dodany Angular routing? Odpowiedz, że tak.

```
? Would you like to add Angular routing? Yes
```

- Jaki format arkuszy stylów ma być używany? Wybierz CSS.

```
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
  SCSS   [ https://sass-lang.com/documentation/syntax#scss                ]
  Sass   [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less   [ http://lesscss.org                                             ]
```

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%;
}
```

Zapisz wszystkie zmiany i zaobserwuj zmianę wyglądu aplikacji w przeglądarce.

16. Pozostaw uruchomioną aplikację front-endową i otwórz nowe okno terminala aby móc uruchamiać kolejne komendy Angular CLI. Zmień katalog bieżący w nowym oknie na główny katalog projektu front-endowego. Utwórz nowy komponent Angulara poniższym poleceniem:

```
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;

    constructor(index: number, firstName: string, lastName: string,
                id?: number) {
        this.index = index;
        this.firstName = firstName;
        this.lastName = lastName;
        this.id = id;
    }
}
```

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

Pole id zostało oznaczone jako opcjonalne, gdyż planujemy nadawać identyfikatory po stronie backendu. Operacja dodawania nowych studentów będzie działać na instancjach klasy Student z niezdefiniowanym (undefined) identyfikatorem. Analogicznie parametr konstruktora reprezentujący identyfikator jest parametrem opcjonalnym. Został umieszczony na końcu listy parametrów, gdyż parametry opcjonalne muszą występować po obowiązkowych.

19. Przejdź do edycji klasy komponentu StudentsComponent 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 pole:

```
students!: Student [];
```

Wykrzyknik stanowi informację dla kompilatora TypeScript, że pole będzie zainicjalizowane przed użyciem. W naszym wypadku kompilator tego sam by nie wydedukował, gdyż inicjalizacji nie dokonamy w konstruktorze, tylko w zwykłej metodzie.

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 z back-endu, 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 klienta HTTP do klasy serwisu poprzez parametr konstruktora:

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

c) Dodaj prywatne pole (ewentualnie popraw numer portu na właściwy):

```
private studentsApiUrl = 'http://localhost:5000/api/students';
```

d) Zmień wartość zwrotną metody getStudents() na

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

e) Usuń niepotrzebną już tablicę z zapisanymi w kodzie danymi studentów.

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 Program.cs (w dowolnym edytorze tekstowym - nie musimy dla tak drobnej modyfikacji otwierać całego projektu w IDE).

c) Przed instrukcją

```
builder.Services.AddControllers();
```

wstaw instrukcję

```
builder.Services.AddCors();
```

d) Przed instrukcją

```
app.UseAuthorization();
```

wstaw instrukcję

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

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

```
dotnet run
```

f) Używając narzędzia Postman dodaj kilku studentów do bazy danych poprzez API. (Niestety informacje z pamięciowej bazy danych zostały utracone gdy aplikacja została zatrzymana.)

42. Sprawdź czy teraz aplikacja front-endowa wyświetla dane studentów pobrane 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, zaraz po instrukcjach import) poniższą stałą:

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

45. Dodaj w klasie usługowej poniższe brakujące metody dla funkcjonalności CRUD:

```
  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>
```

Uwaga: Operator plus poprzedzający wartość indeksu studenta odczytaną z pola formularza wymusza konwersję tej wartości do typu liczbowego.

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(
                        new Student( index, firstName, lastName))
        .subscribe(student => { this.students.push(student); });
  }
```

Uwaga: API w odpowiedzi na żądanie POST odsyła URI nowo utworzonego zasobu w nagłówku Location HTTP. Klient HTTP Angulara na tej podstawie zwraca pobraną instancję już z ustawionym atrybutem id. Tę instancję umieszczamy w lokalnej tablicy zawierającej obiekty studentów. Dzięki temu operacja utworzenia nowego studenta skutkuje jednocześnie dodaniem go po stronie back-endu jak i w interfejsie użytkownika.

48. W szablonie przed wyświetlanym indeksem studenta, 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). (Uwaga: Chrome domyślnie ukrywa żądania preflight w swoim monitorze aktywności sieciowej.)

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. Moduł routingu został utworzony i wstępnie skonfigurowany przez kreator projektu. Odszukaj klasę modułu routingu w strukturze projektu i otwórz ja do edycji. Obejrzyj eksporty i importy modułu routingu. Zauważ że tablica, która będzie zawierać definicje tras jest na razie pusta.

53. 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) Dodaj do tablicy tras poniższą trasę:

```
  { path: 'students', component: StudentsComponent }
```

54. 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>
```

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

`http://localhost:4200/students`

56. 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>
```

57. Sprawdź w przeglądarce działanie pierwszego z linków.

58. Zdefiniuj domyślną ścieżkę routingu, 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' },
```

59. Sprawdź w przeglądarce zachowanie się aplikacji po wejściu na stronę

`http://localhost:4200`

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

```
ng generate component student-detail
```

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

```
student?: Student;
```

Pole jest opcjonalne na wypadek wywołania komponentu z niepoprawnym identyfikatorem studenta zawartym w adresie.

Zaimportuj klasę Student.

62. W tej samej klasie dodaj kolejne importy:

```
import { ActivatedRoute } from '@angular/router';
import { Location } from '@angular/common';

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

63. 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
) {}
```

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

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

Identyfikator przekazany jako parametr ścieżkowy zostanie odczytany ze ścieżki jako string. Dlatego przekazując go do metody komponentu usługowego konwertujemy go do liczby operatorem plus.

65. Wywołaj powyższą metodę w metodzie ngOnInit():

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

66. Dodaj metodę do powrotu do poprzedniego ekranu:

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

67. Dodaj metodę do zapisu edytowanego studenta:

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

Parametr metody został oznaczony wykrzyknikiem, aby poinformować kompilator, że w momencie wywołania metody bieżący student w komponencie na pewno będzie ustawiony. Zapewnimy to w kolejnym punkcie ćwiczenia poprzez wyświetlanie formularza do edycji danych studenta (wraz z przyciskiem wywołującym metodę) tylko gdy bieżący student jest ustawiony.

68. 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.

69. 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.

70. Do tablicy Routes w module routingu dodaj ścieżkę wskazującą studenta do edycji:

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

Dodaj wymagany import.

71. W szablonie komponentu z listą studentów linię:

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

zastąp przez:

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

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

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

`http://localhost:4200/detail/1`

`http://localhost:4200/detail/42`

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”