Laboratorium 2

Interfejsy WWW

version 0.1

Spis treści

Wstęp do laboratorium 2

Temat: Wprowadzenie do Angulara

W poprzednim ćwiczeniu zbudowaliście i uruchomiliście backend webowy oparty o FastAPI, wraz z mechanizmem autoryzacji JWT oraz uruchamianiem aplikacji lokalnie i w kontenerze Docker. Ten etap był potrzebny po to, żebyśmy mieli realne API, z którym frontend będzie mógł rozmawiać – dokładnie tak, jak wygląda to w nowoczesnych projektach webowych, gdzie backend i frontend są niezależnymi aplikacjami.

W ćwiczeniu 2 przechodzimy na „drugą stronę” – budujemy frontend w Angularze. Angular to framework do tworzenia aplikacji typu SPA (Single Page Application), czyli takich, które działają jak klasyczna aplikacja desktopowa: nawigacja pomiędzy ekranami odbywa się bez przeładowywania strony, a dane są pobierane dynamicznie przez HTTP z backendu. Dla użytkownika wygląda to płynnie i „aplikacyjnie”, a dla Was oznacza to, że musicie nauczyć się kilku podstawowych elementów architektury Angulara: komponentów, routingu, formularzy, serwisów oraz komunikacji z API.

Istotne jest to, że dzisiejsze ćwiczenie nie będzie zbiorem oderwanych przykładów. Zrobicie jedną spójną miniaplikację (taskboard / lista zadań), która krok po kroku będzie rozbudowywana: najpierw powstanie szkielet projektu i widoki, potem logowanie do backendu, następnie zabezpieczenie komunikacji tokenem JWT, a na końcu pobieranie danych i praca na prostym modelu „zadań”. Dzięki temu zobaczycie pełny przepływ typowy dla aplikacji komercyjnej: użytkownik loguje się → frontend zapisuje token → kolejne zapytania do API są autoryzowane → backend zwraca dane.

Cel ćwiczenia

W trakcie ćwiczenia 2 macie nauczyć się budować frontend Angular w sposób, który jest zgodny z realnymi praktykami projektowymi. W szczególności:

- utworzycie projekt Angular przez Angular CLI i uruchomicie go w WSL,
- zbudujecie podstawową strukturę aplikacji (komponenty i routing),
- zaimplementujecie logowanie do backendu FastAPI z użyciem JSON,
- zapiszecie i wykorzystacie token JWT po stronie frontendu,
- zabezpieczycie komunikację z API przez interceptor HTTP (Authorization: Bearer …),
- pobierzecie dane z endpointu chronionego JWT i wyświetlicie je w widoku listy.

Zakres i charakter ćwiczenia

Ćwiczenie ma charakter praktyczny. Skupiamy się na tym, żebyście:

- umieli uruchomić środowisko pracy (WSL + VS Code + Angular CLI),
- rozumieli, jak Angular „składa się” z komponentów i tras,
- zobaczyli, jak wygląda komunikacja z backendem w SPA,
- potrafili zrealizować podstawowy scenariusz bezpieczeństwa: logowanie i token JWT.

Rozwiązania – podobnie jak w ćwiczeniu 1 – będą celowo uproszczone tam, gdzie nie jest to kluczowe (np. prosta lista zadań zamiast bazy danych i rozbudowanego CRUD), ale mechanizmy pozostaną takie same, jak w aplikacjach produkcyjnych: separacja odpowiedzialności (komponenty vs serwisy), praca na asynchronicznych requestach HTTP, i autoryzacja oparta o token.

Technologie wykorzystywane w ćwiczeniu

WSL (Ubuntu) + VS Code
Pracujecie w tym samym środowisku co wcześniej: terminal, Node.js i narzędzia uruchamiacie w WSL. VS Code działa jako edytor, ale uruchamia procesy „w środku" Linuxa. To ogranicza problemy wynikające z różnic między środowiskiem Windows a serwerowym.

Angular + TypeScript
Angular jest frameworkiem, który narzucW poprzednim ćwiczeniu zbudowaliście i uruchomiliście backend webowy oparty o FastAPI, wraz z mechanizmem autoryzacji JWT oraz uruchamianiem aplikacji lokalnie i w kontenerze Docker. Ten etap był potrzebny po to, żebyśmy mieli realne API, z którym frontend będzie mógł rozmawiać – dokładnie tak, jak wygląda to w nowoczesnych projektach webowych, gdzie backend i frontend są niezależnymi aplikacjami.

lab 1

W ćwiczeniu 2 przechodzimy na „drugą stronę" – budujemy frontend w Angularze. Angular to framework do tworzenia aplikacji typu SPA (Single Page Application), czyli takich, które działają jak klasyczna aplikacja desktopowa: nawigacja pomiędzy ekranami odbywa się bez przeładowywania strony, a dane są pobierane dynamicznie przez HTTP z backendu. Dla użytkownika wygląda to płynnie i „aplikacyjnie", a dla Was oznacza to, że musicie nauczyć się kilku podstawowych elementów architektury Angulara: komponentów, routingu, formularzy, serwisów oraz komunikacji z API.

Istotne jest to, że dzisiejsze ćwiczenie nie będzie zbiorem oderwanych przykładów. Zrobicie jedną spójną miniaplikację (taskboard / lista zadań), która krok po kroku będzie rozbudowywana: najpierw powstanie szkielet projektu i widoki, potem logowanie do backendu, następnie zabezpieczenie komunikacji tokenem JWT, a na końcu pobieranie danych i praca na prostym modelu „zadań". Dzięki temu zobaczycie pełny przepływ typowy dla aplikacji komercyjnej: użytkownik loguje się → frontend zapisuje token → kolejne zapytania do API są autoryzowane → backend zwraca dane.

Cel ćwiczenia

W trakcie ćwiczenia 2 macie nauczyć się budować frontend Angular w sposób, który jest zgodny z realnymi praktykami projektowymi. W szczególności:

- utworzycie projekt Angular przez Angular CLI i uruchomicie go w WSL,
- zbudujecie podstawową strukturę aplikacji (komponenty i routing),
- zaimplementujecie logowanie do backendu FastAPI z użyciem JSON,
- zapiszecie i wykorzystacie token JWT po stronie frontendu,
- zabezpieczycie komunikację z API przez interceptor HTTP (Authorization: Bearer …),
- pobierzecie dane z endpointu chronionego JWT i wyświetlicie je w widoku listy.

Zakres i charakter ćwiczenia

Ćwiczenie ma charakter praktyczny. Skupiamy się na tym, żebyście:

- umieli uruchomić środowisko pracy (WSL + VS Code + Angular CLI),
- rozumieli, jak Angular „składa się" z komponentów i tras,
- zobaczyli, jak wygląda komunikacja z backendem w SPA,
- potrafili zrealizować podstawowy scenariusz bezpieczeństwa: logowanie i token JWT.

Rozwiązania – podobnie jak w ćwiczeniu 1 – będą celowo uproszczone tam, gdzie nie jest to kluczowe dydaktycznie (np. prosta lista zadań zamiast bazy danych i rozbudowanego CRUD), ale mechanizmy pozostaną takie same, jak w aplikacjach produkcyjnych: separacja odpowiedzialności (komponenty vs serwisy), praca na asynchronicznych requestach HTTP, i autoryzacja oparta o token.

Technologie wykorzystywane w ćwiczeniu

WSL (Ubuntu) + VS Code
Pracujecie w tym samym środowisku co wcześniej: terminal, Node.js i narzędzia uruchamiacie w WSL. VS Code działa jako edytor, ale uruchamia procesy „w środku" Linuxa. To ogranicza problemy wynikające z różnic między środowiskiem Windows a serwerowym.

Angular + TypeScript
Angular jest frameworkiem, który narzuca pewną architekturę: aplikacja jest zbudowana z komponentów, a logika komunikacji z backendem zwykle trafia do serwisów. TypeScript daje typowanie i lepszą kontrolę nad kodem – szczególnie przy pracy z danymi z API.

HTTP + REST API
Frontend nie ma własnej bazy danych. Wszystkie dane pobieracie przez HTTP z backendu (REST). To oznacza, że musicie umieć:

- wykonać zapytanie HTTP (GET/POST),
- obsłużyć odpowiedź,
- zareagować na błędy (np. 401 Unauthorized).

JWT (JSON Web Token) po stronie klienta
JWT jest kluczowy, bo bez niego nie macie dostępu do endpointów chronionych. Po zalogowaniu backend zwróci Wam token, a frontend będzie go przechowywał i automatycznie dołączał do kolejnych requestów. Bardzo ważne: w tym ćwiczeniu token służy do autoryzacji, ale nie traktujcie frontendu jako „miejsca bezpieczeństwa" – prawdziwa kontrola dostępu zawsze jest po stronie backendu.

Co będzie dla Was najważniejsze dzisiaj

Po tym ćwiczeniu powinniście rozumieć trzy rzeczy, które regularnie wracają w prawdziwych projektach:

1. **Angular to architektura, nie tylko "ładne widoki".**
   Komponenty są odpowiedzialne za UI, ale komunikację z API realizuje się serwisami. To porządkuje projekt i pozwala go rozwijać.

2. **Routing i podział aplikacji na ekrany to fundament SPA.**
   Nawet prosta aplikacja szybko robi się nieczytelna bez tras: `/login`, `/tasks`, `/tasks/new`.
3. **JWT + interceptor to standardowy sposób autoryzacji w SPA.**
   Nie chcecie dopisywać ręcznie nagłówków w każdym miejscu – interceptor rozwiązuje to centralnie.

Część 1: Utworzenie projektu Angular i przygotowanie frontendu do integracji z backendem

W tej części rozpoczynamy pracę nad frontendem aplikacji webowej w Angularze. Backend API, przygotowany w ramach poprzedniego ćwiczenia, traktujemy jako gotowy i uruchomiony. Naszym zadaniem jest stworzenie projektu Angular, uruchomienie go w środowisku WSL oraz przygotowanie podstawowej struktury aplikacji, która w kolejnych krokach będzie integrowana z backendem FastAPI.

1. Utworzenie katalogu projektu frontendowego

Pracujemy w środowisku WSL, w tym samym katalogu roboczym, w którym znajdował się backend.

1. Otwórz terminal WSL (najlepiej w VS Code).
2. Przejdź do katalogu roboczego, np.:

```
cd ~/lab_iwww
```

3. Utwórz katalog na frontend:

```
mkdir frontend
cd frontend
```

Rozdzielamy backend i frontend na osobne katalogi. Jest to standardowa praktyka w projektach, w których frontend i backend są niezależnymi aplikacjami.

2. Sprawdzenie środowiska Node.js

Zanim utworzymy projekt Angular, upewniamy się, że Node.js oraz npm działają poprawnie w WSL.

W terminalu wpisz:

```
node -v
npm -v
```

Oczekiwany wynik:

- wyświetlona wersja Node.js (LTS),
- wyświetlona wersja npm.

Jeżeli polecenia nie działają, oznacza to, że środowisko z Laboratorium 1 nie jest poprawnie przygotowane – w takiej sytuacji należy wrócić do poprzedniego ćwiczenia.

3. Instalacja Angular CLI

Angular CLI (Command Line Interface) to narzędzie, które:

- generuje strukturę projektu,

- uruchamia serwer developerski,
- generuje komponenty, serwisy i inne elementy aplikacji.

Instalujemy je globalnie w środowisku WSL.

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

Po zakończeniu instalacji sprawdź wersję:

```
ng version
```

Jeżeli zobaczysz informacje o Angular CLI, Node.js i TypeScript – instalacja przebiegła poprawnie.

4. Utworzenie nowego projektu Angular

Będąc w katalogu `frontend`, utwórz nowy projekt Angular:

```
ng new taskboard --routing --style=scss
```

W trakcie działania kreatora odpowiadaj zgodnie z wartościami domyślnymi na pytania konfiguratora – wartości ważne dla nas wymusiliśmy w linii poleceń.

Co się dzieje:

- Angular CLI generuje kompletną strukturę projektu,
- dodaje obsługę routingu (SPA),
- konfiguruje SCSS jako format stylów.

Po zakończeniu polecenia powstanie katalog:

```
frontend/
 └── taskboard/git c
```

Przejdź do katalogu projektu:

```
cd taskboard
```

5. Uruchomienie aplikacji Angular

Uruchamiamy serwer developerski Angulara:

```
ng serve --host 0.0.0.0 --port 4200
```

Parametry:

- `--host 0.0.0.0` – pozwala na dostęp do aplikacji z przeglądarki Windows,
- `--port 4200` – domyślny port Angulara.

Otwórz przeglądarkę (Windows) i wejdź na adres:

```
http://localhost:4200
```

Oczekiwany efekt:

- wyświetla się domyślna strona startowa Angular,

- brak błędów w konsoli przeglądarki.

To potwierdza, że Angular działa poprawnie w środowisku WSL.

6. Otworzenie projektu w Visual Studio Code (WSL)

Jeżeli nie masz jeszcze otwartego projektu w VS Code, wykonaj w katalogu `taskboard`:

```
code .
```

VS Code:

- uruchomi się w trybie **Remote – WSL**,
- terminale, Node.js i Angular CLI będą działały po stronie Linuxa,
- edycja plików będzie odbywała się w środowisku zgodnym z backendem.

Sprawdź w lewym dolnym rogu VS Code, czy widnieje informacja:

```
WSL: Ubuntu
```

7. Krótkie omówienie struktury projektu Angular

Przejrzyj strukturę katalogu `src/app`:

```
src/
 └── app/
     ├── app.ts
     ├── app.html
     ├── app.scss
     ├── app.config.ts
     ├── app.routes.ts
     └── app.spec.ts
```

Najważniejsze elementy:

- **app.ts** – główny komponent aplikacji,
- **app.routes.ts** – konfiguracja tras,
- **app.config.ts** – konfiguracja samej aplikacji.

Na tym etapie **nie modyfikujemy jeszcze kodu** – chodzi o zrozumienie, jak Angular organizuje aplikację.

8. Przygotowanie konfiguracji backendu (environment)

Frontend będzie komunikował się z backendem FastAPI. Adres API zapisujemy w konfiguracji środowiska. Nie wykorzystujemy do tego bezpośrednio app.config – bo tam jest konfiguracja techniczna samej aplikacji, a nie konfiguracja jej logiki biznesowej. Natomiast wykorzystamy go do prawidłowego wstrzyknięcia konfiguracji do innych modułów.

Utwórz nowy plik :

```
src/app/app.settings.ts
```

Dodaj tam interfejs do ustawień wykorzystywany przez wszystkie zainteresowane moduły:

```
import { InjectionToken } from '@angular/core';

export interface AppSettings {
  apiBaseUrl: string;
}

export const APP_SETTINGS = new InjectionToken<AppSettings>('app.settings');
```

Teraz pozostaje nam uzupełnić app.config.ts – zmieńcie go do następującej postaci:

```
import { ApplicationConfig, provideBrowserGlobalErrorListeners } from
'@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';
import { APP_SETTINGS, AppSettings } from './app.settings';

const settings: AppSettings = {
  apiBaseUrl: 'http://localhost:8000',
};

export const appConfig: ApplicationConfig = {
  providers: [
    provideBrowserGlobalErrorListeners(),
    provideRouter(routes),
    { provide: APP_SETTINGS, useValue: settings },
  ],
};
```

Co to oznacza:

- wszystkie zapytania HTTP do backendu będą korzystały z jednej, centralnej konfiguracji,
- w przyszłości można łatwo zmienić adres API (np. środowisko produkcyjne),
- połączyliśmy konfigurację wewnętrzną i logiki zewnętrznej, korzystając z mechanizmu wstrzykiwania zależności - **injection**

9. Sprawdzenie punktu kontrolnego

Na zakończenie tej części upewnij się, że:

- Angular CLI jest zainstalowany i działa,
- projekt `taskboard` został utworzony,
- aplikacja uruchamia się pod `http://localhost:4200`,
- projekt jest otwarty w VS Code w trybie WSL,
- w pliku `app.config.ts` ustawiony jest adres backendu FastAPI,
- backend API (z Lab 1) działa niezależnie pod `http://localhost:8000`.

Na tym etapie frontend **jeszcze nie komunikuje się z backendem** – został jedynie przygotowany grunt pod kolejne kroki.

Część 2: Struktura aplikacji, routing oraz ekran logowania (standalone)

W tej części przechodzicie od „pustego” projektu Angular do aplikacji, która posiada wyraźny podział na widoki oraz pierwszy realny scenariusz użytkownika: **ekran logowania**. Na tym etapie frontend nie pobiera jeszcze danych domenowych (zadań), ale potrafi już komunikować się z użytkownikiem oraz przygotowuje się do integracji z backendem w zakresie uwierzytelniania.

Projekt jest realizowany w architekturze **standalone**, co oznacza, że:

- nie korzystamy z `AppModule`,
- routing i konfiguracja aplikacji znajdują się w plikach `app.routes.ts` oraz `app.config.ts`,
- komponenty są definiowane jako standalone.

1. Przygotowanie struktury katalogów aplikacji

Aby zachować porządek w projekcie, rozdzielamy odpowiedzialności na logiczne obszary funkcjonalne. Angular nie narzuca struktury katalogów, jednak jest to powszechna i zalecana praktyka.

W katalogu `src/app` utwórzcie następującą strukturę:

```
src/app/
 ├── core/
 │   └── layout/
 ├── auth/
 │   └── pages/
 │       └── login/
 └── tasks/
     └── pages/
         ├── task-list/
         └── task-form/
```

Znaczenie katalogów:

- **core** – elementy wspólne dla całej aplikacji (layout, nawigacja),
- **auth** – logowanie i autoryzacja użytkownika,
- **tasks** – widoki związane z zadaniami (wykorzystane w kolejnych częściach).

Struktura ta odpowiada przyszłym obszarom funkcjonalnym aplikacji.

2. Generowanie komponentów Angular (standalone)

W katalogu głównym projektu (`taskboard`) generujcie komponenty za pomocą Angular CLI.

2.1. Layout aplikacji

```
ng g c core/layout
```

Komponent ten będzie zawierał:

- nagłówek aplikacji,
- nawigację,
- `<router-outlet>`.

W nowym Angularze komponent zostanie wygenerowany automatycznie jako **standalone**.

2.2. Komponent logowania

```
ng g c auth/pages/login
```

To będzie pierwszy widok aplikacji, który zobaczy użytkownik.

2.3. Widoki zadań (na później)

```
ng g c tasks/pages/task-list
ng g c tasks/pages/task-form
```

Na tym etapie komponenty te mogą pozostać puste – przygotowujemy je pod kolejne części ćwiczenia.

3. Konfiguracja routingu aplikacji (standalone)

Routing w aplikacji typu SPA pozwala przełączać widoki bez przeładowania strony.
W architekturze standalone **nie korzystamy z `app-routing.module.ts`**.

3.1. Definicja tras

Otwórzcie plik:

```
src/app/app.routes.ts
```

Zdefiniujcie trasy:

```
import { Routes } from '@angular/router';
import { Login } from './pages/login/login';
import { TaskList } from './pages/task-list/task-list';
import { TaskForm } from './pages/task-form/task-form';

export const routes: Routes = [
  { path: 'login', component: Login },
  { path: 'tasks', component: TaskList },
  { path: 'tasks/new', component: TaskForm },
  { path: 'tasks/:id/edit', component: TaskForm },
  { path: '', redirectTo: 'login', pathMatch: 'full' },
  { path: '**', redirectTo: 'login' }
];
```

3.2. Rejestracja routingu w aplikacji

Otwórzcie plik:

```
src/app/app.config.ts
```

Upewnijcie się, że routing jest podpięty:

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';
```

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes)
  ]
};
```

- `app.routes.ts` definiuje trasy,
- `app.config.ts` rejestruje routing w aplikacji,
- pełni on rolę dawnego `AppModule`.

4. Implementacja layoutu aplikacji

4.1. Szablon layoutu

Otwórzcie plik:

```
src/app/core/layout/layout.html
```

Wstawcie:

```
<header>
  <h1>TaskBoard</h1>
  <nav>
    <a routerLink="/login">Login</a>
    <a routerLink="/tasks">Tasks</a>
    <a routerLink="/tasks/new">Add task</a>
  </nav>
</header>

<main>
  <router-outlet></router-outlet>
</main>
<footer>
  <p>&copy; 2026 IWWW Team</p>
</footer>
```

Layout będzie wspólny dla wszystkich widoków aplikacji. VS powinien Wam zaproponować zmiany także w **layout.ts**. Upewnijcie się że będzie on podobny do tego:

```
import { Component } from '@angular/core';
import { RouterOutlet, RouterLink } from '@angular/router';

@Component({
  selector: 'app-layout',
  standalone: true,
  imports: [RouterOutlet, RouterLink],
  templateUrl: './layout.html',
  styleUrl: './layout.scss',
})
export class Layout {}
```

W aplikacjach standalone istotne jest, by każdy komponent miał odpowiedni import wykorzystywanych modułów / innych komponentów.

4.2. Użycie layoutu w AppComponent

Otwórzcie:

```
src/app/app.html
```

Zastąpcie całą zawartość:

```
<app-layout></app-layout>
```

Od tej pory wszystkie trasy będą renderowane wewnątrz layoutu.

5. Przygotowanie formularza logowania (standalone)

5.1. Import ReactiveFormsModule

W aplikacji standalone **nie importujemy modułów w `AppModule`**, lecz bezpośrednio w komponencie.

Otwórzcie:

```
src/app/auth/pages/login/login.ts
```

Upewnijcie się, że komponent wygląda następująco:

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators, ReactiveFormsModule } from
'@angular/forms';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './login.html',
  styleUrls: ['./login.scss']
})
export class Login {

  form: FormGroup;

  constructor(private fb: FormBuilder) {
    this.form = this.fb.group({
      username: ['', Validators.required],
      password: ['', Validators.required]
    });
  }

  submit(): void {
    if (this.form.invalid) {
      return;
    }

    console.log(this.form.value);
  }
}
```

- `ReactiveFormsModule` jest importowany bezpośrednio do komponentu,
- komponent działa niezależnie od modułów.

5.2. Szablon formularza logowania

Otwórzcie:

```
src/app/auth/pages/login/login.component.html
```

Wstawcie:

```
<h2>Logowanie</h2>

<form [formGroup]="form" (ngSubmit)="submit()">
  <div>
    <label>Login</label>
    <input type="text" formControlName="username">
  </div>

  <div>
    <label>Hasło</label>
    <input type="password" formControlName="password">
  </div>

  <button type="submit">Zaloguj</button>
</form>
```

6. Test działania formularza

1. Upewnijcie się, że `ng serve` działa.
2. Otwórzcie przeglądarkę:

   ```
   http://localhost:4200/login
   ```

3. Wpiszcie dowolne dane i kliknijcie **Zaloguj**.
4. Otwórzcie konsolę przeglądarki (F12).

Oczekiwany efekt:

```
{ "username": "...", "password": "..." }
```

To potwierdza, że:

- routing działa poprawnie,
- layout jest renderowany,
- formularz działa,
- Reactive Forms są poprawnie skonfigurowane w trybie standalone.

7. Wygląd

Doprowadziliśmy do częściowego działania aplikacji – tyle że wygląda ona paskudnie. Spróbujmy coś z tym zrobić. Przejdźcie do terminala, i tam zainstalujcie bootstrap:

```
npm install bootstrap @popperjs/core
```

Potem otwieramy plik styles.scss, i tam dodajemy:

```
@import 'bootstrap/dist/css/bootstrap.min.css';
```

I dodajemy odpowiednie znaczniki stylów w naszym layoucie:

```
<header class="navbar navbar-expand-lg navbar-dark bg-primary">
  <div class="container">
    <a class="navbar-brand" href="#">TaskBoard</a>
    <nav class="navbar-nav ms-auto">
```

```
      <a class="nav-link" routerLink="/login">Login</a>
      <a class="nav-link" routerLink="/tasks">Tasks</a>
      <a class="nav-link" routerLink="/tasks/new">Add task</a>
    </nav>
  </div>
</header>

<main class="container mt-4">
  <router-outlet></router-outlet>
</main>
<footer>
  <p>&copy; 2026 IWWW Team</p>
</footer>
```

Jeszcze formatka logowania nie wygląda najlepiej – postarajmy się zmienić ją na nową, zrobioną zgodnie z bootstrap:

```
<div class="row justify-content-center">
  <div class="col-md-6">
    <div class="card">
      <div class="card-header">
        <h2 class="card-title mb-0">Logowanie</h2>
      </div>
      <div class="card-body">
        <form [formGroup]="form" (ngSubmit)="submit()">
          <div class="mb-3">
            <label for="username" class="form-label">Login</label>
            <input type="text" id="username" class="form-control"
formControlName="username" />
          </div>

          <div class="mb-3">
            <label for="password" class="form-label">Hasło</label>
            <input type="password" id="password" class="form-control"
formControlName="password" />
          </div>

          <button type="submit" class="btn btn-primary w-100"
[disabled]="form.invalid">
            Zaloguj
          </button>
        </form>
      </div>
    </div>
  </div>
</div>
```

Niektóre zmiany wyglądu najwygodniej jest wprowadzać w plikach stylów. Przesuńmy jeszcze podpis na sam spód ekranu. W tym celu otwórz plik layout.scss, i tam podaj:

```
:host {
  display: flex;
  flex-direction: column;
  min-height: 100vh;
}

main {
  flex: 1;
}

footer {
  margin-top: auto;
  text-align: center;
```

```
  padding: 1rem;
  background-color: #f8f9fa;
  border-top: 1px solid #dee2e6;
}
```

7. Punkt kontrolny po Części 2

Na tym etapie aplikacja frontendowa:

- posiada logiczną strukturę katalogów,
- ma skonfigurowany routing (standalone),
- wyświetla layout z nawigacją,
- posiada działający ekran logowania,
- poprawnie zbiera dane użytkownika w formularzu.

Frontend **nie komunikuje się jeszcze z backendem** – zostanie to zrealizowane w kolejnej części laboratorium.

Część 3: Logowanie do backendu (JSON), zapis JWT i pierwszy request do endpointu chronionego

W tej części podłączamy frontend Angular do backendu FastAPI. Implementujemy logowanie przez JSON (POST `/login`), zapisujemy token JWT po stronie frontendu i wykonujemy pierwsze wywołanie do endpointu chronionego (`GET /me`) z użyciem tokena.

1. Import HttpClient w aplikacji

Angular do wykonywania zapytań HTTP używa `HttpClient`. Musi być on włączony w konfiguracji aplikacji.

1.1. Włączenie obsługi HTTP w aplikacji (standalone)

Otwórzcie plik:

- `src/app/app.config.ts`

Dodajcie import:

```
import { provideHttpClient } from '@angular/common/http';
```

i dopiszcie do `providers` linijkę `provideHttpClient()`:

```
...
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes),
    provideHttpClient(),
    ...
  ]
};
```

Angular rejestruje usługi HTTP globalnie i od tej chwili możecie wstrzykiwać `HttpClient` w serwisach i komponentach.

2. Utworzenie AuthService

Logika logowania i trzymania tokena nie powinna siedzieć w komponencie. Komponent odpowiada za UI, a serwis za komunikację i stan autoryzacji.

2.1. Wygeneruj serwis

W katalogu głównym projektu wykonaj:

```
ng g s services/auth
```

Powstanie plik:

```
src/app/services/auth.service.ts
```

2.2. Implementacja AuthService (login JSON + token)

Otwórz:

```
src/app/services/auth.service.ts
```

Wstaw kod:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { tap } from 'rxjs';
import { Inject } from '@angular/core';
import { APP_SETTINGS, AppSettings } from '../app.settings';

type LoginResponse = {
  access_token: string;
  token_type: string;
};

@Injectable({
  providedIn: 'root',
})
export class AuthService {
  private readonly tokenKey = 'access_token';

  constructor(private http: HttpClient, @Inject(APP_SETTINGS) private settings:
AppSettings) {}

  login(username: string, password: string) {
    return this.http
      .post<LoginResponse>(`${this.settings.apiBaseUrl}/login`, { username,
password })
      .pipe(
        tap((res) => {
          localStorage.setItem(this.tokenKey, res.access_token);
        })
      );
  }

  logout(): void {
    localStorage.removeItem(this.tokenKey);
  }

  getToken(): string | null {
    return localStorage.getItem(this.tokenKey);
  }

  isLoggedIn(): boolean {
    return !!this.getToken();
  }
}
```

Co się dzieje:

- `login()` wysyła JSON do `/login`,
- po poprawnym logowaniu token jest zapisywany w `localStorage`,
- `getToken()` zwraca token do użycia w kolejnych częściach,
- `logout()` czyści token.

3. Podłączenie logowania w LoginComponent

Teraz komponent logowania przestaje wypisywać dane w konsoli — zamiast tego wywołuje API.

3.1. Zmiany w LoginComponent

Otwórz:

```
src/app/pages/login/login.ts
```

Dodaj importy:

```
import { Router } from '@angular/router';
import { AuthService } from '../../services/auth';
```

Zmień konstruktor:

```
constructor(
  private fb: FormBuilder,
  private auth: AuthService,
  private router: Router
) {
  this.form = this.fb.group({
    username: ['', Validators.required],
    password: ['', Validators.required]
  });
}
```

Zmień `submit()`:

```
submit(): void {
  if (this.form.invalid) {
    return;
  }

  const { username, password } = this.form.value;

  this.auth.login(username, password).subscribe({
    next: () => {
      this.router.navigate(['/tasks']);
    },
    error: () => {
      alert('Błędny login lub hasło');
    }
  });
}
```

Co się dzieje:

- po poprawnym logowaniu zapisuje się token,
- użytkownik jest przekierowany do `/tasks`,
- błędy logowania obsługujemy prostym komunikatem.

Niestety – to nie wystarczy. Ponieważ tymczasowo mamy API i aplikację serwowaną z różnych adresów (różnią się portem) będziemy mieli błąd cross scripting. By go uniknąć – musimy wrócić do API z poprzedniego lab, i tam zrobić zmiany. Po pierwsze, zaimportować wsparcie dla CORS:

```
from fastapi.middleware.cors import CORSMiddleware
```

A po drugie – zmienić plik main.py – tak by uwzględniał wysyłanie czystego json a nie url encoded form:

```
from fastapi import FastAPI, Depends, HTTPException
from fastapi.security import OAuth2PasswordRequestForm # to nie jest potrzebne
```

```
from fastapi.middleware.cors import CORSMiddleware

from app.auth import authenticate_user, create_access_token, get_current_user
from pydantic import BaseModel

class LoginRequest(BaseModel):
    username: str
    password: str

app = FastAPI(title="FastAPI JWT Demo")

origins = [
    "http://localhost:4200",
    "http://127.0.0.1:4200",
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.post("/login")
def login(data: LoginRequest):
    # tu zmieniony payload z OAuth2PasswordRequestForm na LoginRequest
    user = authenticate_user(data.username, data.password)
    if not user:
        raise HTTPException(status_code=401, detail="Invalid credentials")

    token = create_access_token({"sub": user["username"]})
    return {"access_token": token, "token_type": "bearer"}

@app.get("/me")
def read_me(current_user: dict = Depends(get_current_user)):
    return {
        "username": current_user["username"],
        "role": current_user["role"],
    }
```

4. Test logowania i weryfikacja tokena w przeglądarce

4.1. Test w UI

1. Wejdź na:

```
http://localhost:4200/login
```

2. Zaloguj się:
   - `admin / admin123`
   - albo `user / user123`
3. Po logowaniu aplikacja przekieruje na `/tasks`.

4.2. Sprawdzenie tokena w Local Storage

W przeglądarce:

1. Otwórz DevTools (F12)
2. Wejdź w zakładkę **Application** (lub Storage)
3. Otwórz **Local Storage**
4. Wybierz `http://localhost:4200`

Oczekiwany efekt:

- istnieje wpis `access_token` z długim ciągiem znaków (JWT).

Jeżeli token się nie pojawia:

- sprawdź w Network, czy `/login` zwraca 200,
- sprawdź, czy backend przyjmuje JSON (a nie form-urlencoded).

5. Pierwsze wywołanie endpointu chronionego: /me (tymczasowo)

Na razie nie mamy jeszcze interceptora.
Żeby sprawdzić, że token działa, wykonamy test ręczny, dopisując nagłówek w kodzie.

5.1. Dodaj test w TaskListComponent

Otwórz:

```
src/app/tasks/pages/task-list/task-list.ts
```

Dodaj importy:

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { AuthService } from '../../../services/auth';
```

Zmień komponent tak, żeby w konstruktorze wstrzyknąć serwisy:

```
constructor(
  private http: HttpClient,
  private auth: AuthService
) {}
```

Dodaj `ngOnInit()`:

```
ngOnInit(): void {
  const token = this.auth.getToken();

  const headers = new HttpHeaders({
    Authorization: `Bearer ${token}`
  });

  this.http.get(`${this.settings.apiBaseUrl}/me`, { headers })
    .subscribe({
      next: (res) => console.log('ME:', res),
      error: (err) => console.error('ME ERROR:', err)
    });
}
```

Uwaga:

- to jest **test tymczasowy**,
- w następnej części zastąpimy to interceptor-em.

5.2. Uruchomienie testu

1. Zaloguj się ponownie (żeby mieć token).
2. Przejdź na:

```
http://localhost:4200/tasks
```

3. Otwórz konsolę przeglądarki.

Oczekiwany efekt:

- pojawia się obiekt użytkownika, np.:

```
{ "username": "admin", "role": "admin" }
```

Jeżeli pojawia się 401:

- token nie został zapisany,
- Jest błąd CORS
- token jest pusty,
- backend jest uruchomiony na innym porcie/adresie .

6. Punkt kontrolny po Części 3

Na tym etapie:

- frontend loguje się do backendu przez JSON (`POST /login`),
- token JWT zapisuje się w localStorage,
- potraficie wykonać request do endpointu chronionego (`GET /me`) używając JWT.

W kolejnej części:

- dodamy **HTTP Interceptor**, żeby nagłówek Authorization był dodawany automatycznie,
- obsłużymy błąd 401 (wylogowanie + przekierowanie),
- podłączymy endpoint `/tasks` i pobierzemy listę zadań „prawdziwie”, bez ręcznych nagłówków.