Laboratorium 3

Interfejsy WWW

version 0.1

Spis treści

Wstęp do laboratorium 2

Temat: Wprowadzenie do Angulara – ciąg dalszy

W poprzednich laboratoriach przygotowaliście kompletne środowisko pracy oparte o WSL i VS Code oraz zbudowaliście backend webowy w Pythonie z wykorzystaniem FastAPI. Backend został wyposażony w mechanizm uwierzytelniania i autoryzacji oparty o JWT, co pozwala zabezpieczać endpointy i kontrolować dostęp do danych w zależności od tego, czy użytkownik posiada poprawny token. To podejście jest dziś standardem w architekturach, gdzie frontend i backend są od siebie niezależne, a komunikacja odbywa się przez REST API.

Laboratorium 3 jest kontynuacją pracy po stronie frontendu. Zamiast pojedynczych, oderwanych przykładów będziecie rozwijać jedną spójną aplikację typu SPA w Angularze. Punkt wyjścia jest prosty: użytkownik loguje się do backendu, otrzymuje token JWT, a następnie wykonuje kolejne operacje na zasobach (lista zadań). Na tej bazie przechodzimy od „działającego demo” do rozwiązania, które przypomina praktyki spotykane w realnych projektach: centralne dołączanie tokena do żądań, obsługa sytuacji braku autoryzacji, ochrona tras, a następnie dodawanie danych przez formularze.

Istotnym elementem tego laboratorium jest uporządkowanie odpowiedzialności w aplikacji Angular. Komponenty odpowiadają za UI i interakcję użytkownika, natomiast komunikacja z API oraz logika związana z autoryzacją powinna znajdować się w serwisach i interceptorach. Dzięki temu kod staje się czytelny, łatwiejszy do rozbudowy i zgodny z typowym sposobem organizacji projektów frontendowych.

W trakcie laboratorium przejdziecie przez trzy kluczowe obszary, które w praktyce występują niemal zawsze w aplikacjach SPA:

- **autoryzacja w komunikacji HTTP** (interceptor JWT i obsługa błędu 401),
- **kontrola dostępu do widoków** (guard i routing),
- **operacje na danych z backendu** (pobranie listy oraz dodawanie nowych rekordów przez formularz).

Zastosowane rozwiązania będą celowo uproszczone tam, gdzie nie jest to kluczowe dydaktycznie (np. „baza” zadań jako lista w pamięci), ale mechanizmy pozostają identyczne jak w projektach produkcyjnych: praca na JSON, kontrola dostępu przez token, rozdzielenie logiki na serwisy, oraz walidacja danych po stronie frontendu.

Ważne: w tym laboratorium frontend korzysta z tokena JWT do autoryzacji, ale bezpieczeństwo systemu zawsze wymusza backend. Frontend jedynie przechowuje token i dołącza go do zapytań, natomiast decyzję „czy użytkownik ma dostęp” podejmuje serwer. Zobaczycie to w praktyce w momencie, gdy celowo usuniecie token albo spróbujecie wejść na widok chroniony bez logowania - backend zwróci 401, a aplikacja zareaguje przekierowaniem na ekran logowania.

Na zakończenie laboratorium będziecie mieli działającą aplikację Angular, która:

- loguje użytkownika przez JSON do backendu,
- automatycznie autoryzuje żądania HTTP tokenem JWT,
- blokuje dostęp do tras wymagających zalogowania,
- pobiera listę zadań z endpointu chronionego,

- pozwala dodać nowe zadanie przez formularz Reactive Forms,
- oraz stanowi bazę pod dalszą rozbudowę o edycję, usuwanie, filtrację i role użytkowników.

Część 1: Interceptor JWT, automatyczna autoryzacja oraz pobranie listy zadań z `/tasks`

W tej części eliminujemy ręczne dopisywanie nagłówków `Authorization` w kodzie komponentów i serwisów. Zamiast tego konfigurujemy interceptor HTTP, który automatycznie dołącza token JWT do wszystkich zapytań do backendu. Następnie tworzymy serwis `TasksService` i pobieramy listę zadań z endpointu chronionego `/tasks`.

1. Usunięcie tymczasowego testu `/me` z TaskListComponent

W poprzedniej części (Część 3) dodaliście ręczne wywołanie endpointu `/me` z dopisaniem nagłówka w `TaskListComponent`. Był to test kontrolny działania tokena. Teraz nie jest już potrzebny.

1. Otwórz plik:

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

2. Usuń:

- importy związane z `HttpClient`, `HttpHeaders`, `environment`,
- ręczne tworzenie nagłówków `Authorization`,
- cały fragment testowego `this.http.get(.../me...)`.

Po tym kroku `TaskListComponent` może być na chwilę „pusty” (zajmiemy się nim w punkcie 4).

2. Utworzenie interceptora JWT

Interceptor będzie:

- pobierał token z `AuthService`,
- dołączał nagłówek `Authorization: Bearer <token>`,
- reagował na `401 Unauthorized` (wylogowanie i przekierowanie na `/login`).

2.1. Generowanie interceptora

W katalogu głównym projektu wykonaj:

```
ng g interceptor auth/interceptors/jwt
```

Powstanie plik interceptora w katalogu `src/app/auth/interceptors/`.

Jeżeli Angular CLI utworzy plik z inną nazwą niż oczekujesz, to nie jest problem. Najważniejsze, żebyś wiedział gdzie on jest i co zawiera.

2.2. Implementacja interceptora

Otwórz plik interceptora, np.:

src/app/auth/interceptors/jwt.interceptor.ts

Wstaw kod (albo zmodyfikuj istniejący), tak aby wyglądał w sposób jednoznaczny:

```
import { Injectable } from '@angular/core';
import {
  HttpInterceptor,
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpErrorResponse
} from '@angular/common/http';
import { Observable, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';
import { Router } from '@angular/router';
import { AuthService } from '../services/auth.service';

@Injectable()
export class JwtInterceptor implements HttpInterceptor {

  constructor(
    private auth: AuthService,
    private router: Router
  ) {}

  intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
    const token = this.auth.getToken();

    let authReq = req;

    if (token) {
      authReq = req.clone({
        setHeaders: {
          Authorization: `Bearer ${token}`
        }
      });
    }

    return next.handle(authReq).pipe(
      catchError((err: HttpErrorResponse) => {
        if (err.status === 401) {
          this.auth.logout();
          this.router.navigate(['/login']);
        }
        return throwError(() => err);
      })
    );
  }
}
```

Co się dzieje:

- `req.clone(...)` tworzy kopię requestu z dodatkowym nagłówkiem,
- `catchError` przechwytuje błędy odpowiedzi,
- `401` oznacza brak autoryzacji (np. brak tokena, token wygasł, token błędny),
- w reakcji na `401` czyścimy token i wracamy na ekran logowania.

3. Rejestracja interceptora w aplikacji

Interceptor musi zostać „włączony” w konfiguracji Angular.

3.1. Rejestracja w `app.module.ts`

Otwórz:

```
src/app/app.module.ts
```

Dodaj import:

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { JwtInterceptor } from './auth/interceptors/jwt.interceptor';
```

Następnie w `providers` dodaj:

```
providers: [
  {
    provide: HTTP_INTERCEPTORS,
    useClass: JwtInterceptor,
    multi: true
  }
],
```

Co się dzieje:

- Angular będzie przepuszczał każde zapytanie HTTP przez listę interceptorów,
- `multi: true` oznacza, że możesz mieć więcej interceptorów niż jeden.

4. Utworzenie serwisu TasksService

Nie pobieramy danych bezpośrednio w komponencie. Komponent ma odpowiadać za UI, a serwis za komunikację z API.

4.1. Generowanie serwisu

W katalogu projektu wykonaj:

```
ng g s tasks/services/tasks
```

Powstanie plik:

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

4.2. Implementacja `getTasks()`

Otwórz:

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

Wstaw kod:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { environment } from '../../../environments/environment';

export type Task = {
  id: number;
```

```
  title: string;
  done: boolean;
  priority: string;
};

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

  constructor(private http: HttpClient) {}

  getTasks() {
    return this.http.get<Task[]>(`${environment.apiUrl}/tasks`);
  }
}
```

Co się dzieje:

- `getTasks()` pobiera listę z endpointu `/tasks`,
- nie dopisujemy nagłówków ręcznie, ponieważ robi to interceptor,
- typ `Task` porządkuje strukturę danych w TypeScript.

5. Pobranie i wyświetlenie zadań w TaskListComponent

5.1. Implementacja logiki pobierania w komponencie

Otwórz:

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

Zaimplementuj:

```
import { Component, OnInit } from '@angular/core';
import { TasksService, Task } from '../../services/tasks.service';

@Component({
  selector: 'app-task-list',
  templateUrl: './task-list.component.html',
  styleUrls: ['./task-list.component.scss']
})
export class TaskListComponent implements OnInit {

  tasks: Task[] = [];

  constructor(private tasksService: TasksService) {}

  ngOnInit(): void {
    this.tasksService.getTasks().subscribe({
      next: (data) => this.tasks = data,
      error: (err) => console.error('Błąd pobierania tasks:', err)
    });
  }
}
```

5.2. Implementacja widoku HTML listy

Otwórz:

```
src/app/tasks/pages/task-list/task-list.component.html
```

Wstaw:

```
<h2>Lista zadań</h2>

<ul>
  <li *ngFor="let t of tasks">
    <strong>{{ t.title }}</strong>
    (priority: {{ t.priority }}, done: {{ t.done }})
  </li>
</ul>
```

6. Test działania całości

6.1. Test pozytywny

1. Uruchom backend (jeśli nie działa).
2. Upewnij się, że Angular działa (`ng serve`).
3. Wejdź na:
    - `http://localhost:4200/login`
4. Zaloguj się:
    - `admin / admin123`
5. Przejdź na:
    - `http://localhost:4200/tasks`

Oczekiwany efekt:

- w widoku listy pojawiają się zadania zwrócone przez backend `/tasks`,
- w Network (DevTools) widać request `GET /tasks` z nagłówkiem `Authorization: Bearer ...`.

6.2. Test negatywny (401 i przekierowanie)

1. Usuń token z Local Storage:
    - DevTools → Application → Local Storage → usuń `access_token`
2. Odśwież stronę `/tasks`.

Oczekiwany efekt:

- backend zwraca `401 Unauthorized`,
- interceptor wykonuje `logout()` i przekierowuje na `/login`.

7. Punkt kontrolny po Części 1

Na tym etapie:

- macie działające logowanie i zapis tokena JWT,

- interceptor automatycznie autoryzuje requesty do API,
- pobieracie listę zadań z endpointu chronionego `/tasks`,
- aplikacja reaguje poprawnie na `401`.

W kolejnej części można przejść do:

- ochrony tras guardem (żeby nie dało się wejść na `/tasks` bez logowania),
- formularza dodawania/edycji zadania (Reactive Forms),
- oraz rozszerzenia backendu o `POST/PUT/DELETE` dla tasks.

Część 2: Ochrona tras (AuthGuard) i ukrywanie elementów UI zależnie od zalogowania

W tej części zabezpieczamy aplikację tak, aby użytkownik nie mógł wejść na widoki zadań bez zalogowania. Do tego służy `AuthGuard`, który blokuje dostęp do wybranych tras. Dodatkowo porządkujemy nawigację w layoucie: linki widoczne są zależnie od tego, czy użytkownik jest zalogowany.

1. Utworzenie guarda autoryzacji

Guard jest mechanizmem Angulara, który pozwala zdecydować, czy dana trasa może zostać aktywowana. Jeśli użytkownik nie ma tokena, to zamiast wejścia na `/tasks` zostanie przekierowany na `/login`.

1.1. Generowanie guarda

W katalogu głównym projektu wykonaj:

```
ng g guard auth/guards/auth
```

W trakcie kreatora:

- jeśli pojawi się pytanie o typ guarda, wybierz `CanActivate`.

Powstanie plik w stylu:

```
src/app/auth/guards/auth.guard.ts
```

1.2. Implementacja AuthGuard

Otwórz plik guarda i ustaw treść w sposób jednoznaczny:

```
import { Injectable } from '@angular/core';
import { CanActivate, Router, UrlTree } from '@angular/router';
import { AuthService } from '../services/auth.service';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {

  constructor(
    private auth: AuthService,
    private router: Router
  ) {}

  canActivate(): boolean | UrlTree {
    if (this.auth.isLoggedIn()) {
      return true;
    }
    return this.router.parseUrl('/login');
  }
}
```

Co się dzieje:

- guard pyta `AuthService`, czy użytkownik jest zalogowany,
- jeśli tak, przepuszcza trasę,
- jeśli nie, zwraca przekierowanie na `/login`.

2. Zabezpieczenie tras w routingu

Teraz podpinamy guarda do tras związanych z zadaniami.

2.1. Podłączenie w `app-routing.module.ts`

Otwórz:

```
src/app/app-routing.module.ts
```

Dodaj import:

```
import { AuthGuard } from './auth/guards/auth.guard';
```

Następnie zmodyfikuj trasy tak, aby wszystkie `/tasks...` były chronione:

```
const routes: Routes = [
  { path: 'login', component: LoginComponent },

  { path: 'tasks', component: TaskListComponent, canActivate: [AuthGuard] },
  { path: 'tasks/new', component: TaskFormComponent, canActivate:
[AuthGuard] },
  { path: 'tasks/:id/edit', component: TaskFormComponent, canActivate:
[AuthGuard] },

  { path: '', redirectTo: 'login', pathMatch: 'full' },
  { path: '**', redirectTo: 'login' }
];
```

Co się dzieje:

- użytkownik bez tokena nie wejdzie na listę ani formularz,
- nawet jeśli wpisze adres ręcznie w pasku przeglądarki, guard zadziała.

3. Modyfikacja layoutu: linki zależne od zalogowania

W tej części poprawiamy UX:

- jeśli użytkownik nie jest zalogowany, nie pokazujemy linków do zadań,
- jeśli użytkownik jest zalogowany, pokazujemy link „Wyloguj”.

3.1. Dodanie logiki w LayoutComponent

Otwórz:

```
src/app/core/layout/layout.component.ts
```

Dodaj import:

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

Zaimplementuj konstruktor i metody:

```
export class LayoutComponent {

  constructor(
    public auth: AuthService,
    private router: Router
  ) {}

  logout(): void {
    this.auth.logout();
    this.router.navigate(['/login']);
  }
}
```

Uwaga:

- `public auth` pozwala używać `auth.isLoggedIn()` w HTML bez dodatkowych getterów.

3.2. Warunkowe linki w `layout.component.html`

Otwórz:

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

Zastąp nawigację następującą wersją:

```
<header>
  <h1>TaskBoard</h1>

  <nav>
    <a routerLink="/login" *ngIf="!auth.isLoggedIn()">Login</a>

    <a routerLink="/tasks" *ngIf="auth.isLoggedIn()">Tasks</a>
    <a routerLink="/tasks/new" *ngIf="auth.isLoggedIn()">Add task</a>

    <button type="button" (click)="logout()"
*ngIf="auth.isLoggedIn()">Wyloguj</button>
  </nav>
</header>

<main>
  <router-outlet></router-outlet>
</main>
```

Co się dzieje:

- linki do zadań są widoczne tylko po zalogowaniu,
- wylogowanie czyści token i wraca na `/login`.

4. Test działania guarda i UI

4.1. Test negatywny (brak tokena)

1. Usuń token z Local Storage:
    - DevTools → Application → Local Storage → usuń `access_token`
2. Wpisz ręcznie w pasku adres:

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

Oczekiwany efekt:

- następuje przekierowanie na `/login`,
- linki do zadań w navbarze nie są widoczne.

4.2. Test pozytywny (zalogowany)

1. Zaloguj się (np. `admin / admin123`).
2. Przejdź na `/tasks`.

Oczekiwany efekt:

- guard przepuszcza trasę,
- w navbarze widzisz linki do zadań oraz przycisk „Wyloguj”.

4.3. Test wylogowania

1. Kliknij „Wyloguj”.
2. Spróbuj ponownie wejść na `/tasks`.

Oczekiwany efekt:

- wracasz na `/login`,
- trasy chronione są niedostępne.

Część 3: Formularz zadania (Reactive Forms) oraz dodanie endpointu `POST /tasks` w backendzie

W tej części rozbudowujemy aplikację o możliwość dodawania nowego zadania. Po stronie backendu dodajemy endpoint `POST /tasks`, który przyjmuje dane w JSON i wymaga JWT. Po stronie Angulara implementujemy formularz w `TaskFormComponent`, walidację oraz wysłanie danych do API. Dzięki interceptorowi JWT nie dopisujemy nagłówków ręcznie.

1. Rozszerzenie backendu: `POST /tasks` (JSON + JWT)

Zakładamy, że backend FastAPI działa i ma już:

- `/login` (JSON),
- `/me` (JWT),
- `/tasks` (GET, JWT).

Teraz dodajemy możliwość tworzenia nowych zadań.

1.1. Dodanie modeli Pydantic dla zadań

Otwórz plik backendu, w którym masz endpointy (np. `app/main.py`).

Dodaj import:

```
from pydantic import BaseModel
```

Dodaj modele:

```
class TaskCreate(BaseModel):
    title: str
    done: bool = False
    priority: str = "medium"

class Task(BaseModel):
    id: int
    title: str
    done: bool
    priority: str
```

Co się dzieje:

- `TaskCreate` opisuje dane, które frontend wysyła przy tworzeniu zadania,
- `Task` opisuje pełny rekord z `id`, który zwracamy w odpowiedzi.

1.2. Zmiana przechowywania danych w `tasks_db`

Upewnij się, że `tasks_db` istnieje (lista słowników). Przykładowo:

```
tasks_db = [
    {"id": 1, "title": "Przygotować backend API", "done": True, "priority":
"high"},
    {"id": 2, "title": "Zaimplementować logowanie w Angularze", "done": False,
"priority": "medium"},
```

```
    {"id": 3, "title": "Dodać interceptor JWT", "done": False, "priority":
"high"},
]
```

1.3. Implementacja endpointu `POST /tasks`

Dodaj endpoint:

```
@app.post("/tasks")
def create_task(data: TaskCreate, current_user: dict =
Depends(get_current_user)):
    new_id = max([t["id"] for t in tasks_db], default=0) + 1
    task = {
        "id": new_id,
        "title": data.title,
        "done": data.done,
        "priority": data.priority
    }
    tasks_db.append(task)
    return task
```

Co się dzieje:

- endpoint wymaga tokena JWT (przez `Depends(get_current_user)`),
- tworzymy nowe `id` na podstawie aktualnej listy,
- dopisujemy nowe zadanie do `tasks_db`,
- zwracamy utworzone zadanie jako JSON.

1.4. Test endpointu `POST /tasks` (curl)

1. Zaloguj się i pobierz token:

```
curl -X POST http://localhost:8000/login \
  -H "Content-Type: application/json" \
  -d '{"username":"admin","password":"admin123"}'
```

2. Utwórz zadanie (zmień TWOJ_TOKEN na token otrzymany z poprzedniego wywołania):

```
curl -X POST http://localhost:8000/tasks \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer TWOJ_TOKEN" \
  -d '{"title":"Nowe zadanie z curl","done":false,"priority":"low"}'
```

3. Sprawdź listę:

```
curl -X GET http://localhost:8000/tasks \
  -H "Authorization: Bearer TWOJ_TOKEN"
```

Oczekiwany efekt:

- po `POST` dostajesz obiekt nowego zadania z `id`,
- po `GET` widzisz nowe zadanie na liście.

2. Rozszerzenie Angular: `TasksService.createTask()`

W Angularze dodajemy metodę tworzenia zadania.

2.1. Aktualizacja `TasksService`

Otwórz:

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

Dodaj typ dla tworzenia zadania:

```
export type TaskCreate = {
  title: string;
  done: boolean;
  priority: string;
};
```

Dodaj metodę:

```
createTask(data: TaskCreate) {
  return this.http.post<Task>(`${environment.apiUrl}/tasks`, data);
}
```

Co się dzieje:

- request idzie do `/tasks`,
- interceptor dołącza JWT,
- backend zwraca utworzone zadanie.

3. Formularz dodawania zadania w `TaskFormComponent`

3.1. Implementacja formularza (TypeScript)

Otwórz:

```
src/app/tasks/pages/task-form/task-form.component.ts
```

Zaimplementuj:

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { TasksService, TaskCreate } from '../../services/tasks.service';

@Component({
  selector: 'app-task-form',
  templateUrl: './task-form.component.html',
  styleUrls: ['./task-form.component.scss']
})
export class TaskFormComponent {

  form: FormGroup;

  constructor(
    private fb: FormBuilder,
    private tasksService: TasksService,
    private router: Router
  ) {
    this.form = this.fb.group({
      title: ['', [Validators.required, Validators.minLength(3)]],
      priority: ['medium', Validators.required],
      done: [false]
    });
```

```
  }

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

    const data: TaskCreate = this.form.value;

    this.tasksService.createTask(data).subscribe({
      next: () => this.router.navigate(['/tasks']),
      error: (err) => console.error('Błąd tworzenia task:', err)
    });
  }
}
```

Co się dzieje:

- formularz zawiera 3 pola: `title`, `priority`, `done`,
- `title` ma walidację: wymagane + min długość 3,
- po wysłaniu wywołujemy `createTask()`,
- po sukcesie wracamy na listę.

3.2. Szablon formularza (HTML)

Otwórz:

```
src/app/tasks/pages/task-form/task-form.component.html
```

Wstaw:

```
<h2>Dodaj zadanie</h2>

<form [formGroup]="form" (ngSubmit)="submit()">

  <div>
    <label>Tytuł</label>
    <input type="text" formControlName="title">
    <div *ngIf="form.get('title')?.touched && form.get('title')?.invalid">
      <small *ngIf="form.get('title')?.errors?.['required']">Pole jest
wymagane</small>
      <small *ngIf="form.get('title')?.errors?.['minlength']">Min. 3
znaki</small>
    </div>
  </div>

  <div>
    <label>Priorytet</label>
    <select formControlName="priority">
      <option value="low">low</option>
      <option value="medium">medium</option>
      <option value="high">high</option>
    </select>
  </div>

  <div>
    <label>
      <input type="checkbox" formControlName="done">
      Wykonane
    </label>
```

```
  </div>

  <button type="submit" [disabled]="form.invalid">Zapisz</button>
</form>
```

4. Test działania dodawania zadań z UI

1. Upewnij się, że backend działa.
2. Upewnij się, że `ng serve` działa.
3. Zaloguj się w aplikacji.
4. Wejdź na:
    - `http://localhost:4200/tasks/new`
5. Wpisz tytuł, wybierz priorytet, kliknij „Zapisz”.
6. Aplikacja powinna wrócić na `/tasks`.

Oczekiwany efekt:

- nowe zadanie pojawia się na liście,
- w Network widać request:
    - `POST /tasks` (z `Authorization: Bearer ...`).

Jeżeli dostajesz `401`:

- token nie jest zapisany (sprawdź Local Storage),
- interceptor nie jest zarejestrowany,
- backend działa pod innym adresem niż `environment.apiUrl`.

5. Zadania do samodzielnego wykonania:

- Rozszerzcie projekt o możliwość edycji istniejących zadań
- dodajcie usuwanie zadanń,
- Dodajcie filtrację i paginację wyników
- Dodajcie obsługę ról użytkowników (przeglądający / edytujący)