Zadanie domowe z Vue.js/React + backend (.NET/Express)

W ramach otwartych danych ZTM w Gdańsku mamy następujące API:

- https://ckan.multimediagdansk.pl/dataset/c24aa637-3619-4dc2-a171-a23eec8f2172/resource/4c4025f0-01bf-41f7-a39f-d156d201b82b/download/stops.json – **lista przystanków**, zawiera **stopId** – identyfikator słupka przystankowego unikalny w skali Trójmiasta.
- http://ckan2.multimediagdansk.pl/delays?stopId={stopId} – **estymowane czasy przyjazdów na przystanek**, gdzie {stopId} jest identyfikatorem słupka – wartość stopId z zasobu Lista przystanków. Przykładowe zapytanie dla przystanku na Miszewskiego (stopId=2019): http://ckan2.multimediagdansk.pl/delays?stopId=2019. Dostajemy kolekcję pojazdów które pojawią się na przystanku w najbliższym okresie. Dane są zapisane z użyciem formatu JSON. Mamy tam kilka właściwości zawierających interesujące aktualne informacje, takie jak: numer linii:"routeId", opóźnienie:"delayInSeconds", czas przyjazdu wg rozkładu: "theoreticalTime", rzeczywisty czas odjazdu: "estimatedTime". Dane można je porównać z wybrana podstroną ZTM: https://mapa.ztm.gda.pl/?stop=2019.


```
{
"lastUpdate": "2024-11-16 19:09:13",
"delay": [
{
"id": "T82R9",
"delayInSeconds": 12,
"estimatedTime": "19:09",
"headsign": "Strzyża PKM",
"routeId": 9,
"tripId": 82,
"status": "REALTIME",
"theoreticalTime": "19:09",
"timestamp": "19:07:20",
"trip": 3577762,
"vehicleCode": 1022,
"vehicleId": 416
}, …
```


Rysunek 1 – JSON z estymowanymi czasami przyjazdów na przystanek Miszewskiego.

W ramach zadania laboratoryjnego proszę zaimplementować prostą aplikację pozwalającą na rejestrację i zalogowanie się użytkownika. Hasło zarejestrowanego użytkownika powinno być przechowywane w bazie danych w postaci hasha, np. z wykorzystaniem funkcji bcrypt. Logowanie odbywa się w oparciu o Bearer authentication. Po zalogowaniu wyświetla użytkownikowi jego spersonalizowane dane (w przykładowym kodzie jest to lista wybranych przez niego tablic przystankowych – osobna dla każdego użytkownika), która zawiera listę niebawem odjeżdżających z tego przystanku, pojazdów ZTM. Użytkownik powinien mieć możliwość dodawania, przeglądania, edycji i usuwania tych danych.

Wymagania techniczne (backend):

- Encje modelu danych muszą wykorzystywać zamiast typów prostych obiekty wartościowe, zawierające logikę przynajmniej w postaci walidacji wartości.
- Dane użytkownika (login i hash hasła) przechowywane w bazie danych dokumentowej (np. MongoDB) albo relacyjnej (np. SQLite),

- Wykorzystanie odpowiedniego JavaScript ODM (np. Mongoose) albo ORM (np. EF Core) dla wybranej bazy danych.
- Autoryzacja z wykorzystaniem Bearer authentication.
- Zbudowanie API w node.js (np. z wykorzystaniem frameworka Express.js bądź Sails.js) albo w .NET (np. z wykorzystaniem ASP.NET Web API albo Minimal API) obsługującego utożsamianie użytkownika.
- Zapytania API dotyczące danych zalogowanego użytkownika powinny być chronione przez token JWT (Bearer authentication).
- API bazuje na danych z bazy danych oraz z publicznego API ZTM.
- Długotrwałe zapytanie z API ZTM, np. o listę wszystkich dostępnych przystanków zwracające ciężki plik stops.json powinno być cash'owane.
- Dokumentacja API z wykorzystaniem Swagger'a z możliwością zalogowania się użytkownika (zapamiętania tokenu JWT).
- (dodatkowe punkty) wizualizacja położenia autobusów/przystanków/tras z wykorzystaniem map, np.: Google/OSM itp…

Wymagania techniczne (frontend w Vite + Vue.JS/React):

- Funkcjonalny podział GUI na komponenty jedno-plikowe.
- Implementacja przynajmniej jednego multikomponentu.
- Implementacja i wykorzystanie własnej dyrektywy.
- Implementacja przynajmniej jednej funkcji wielokrotnego użytku (Composable), np. do pobieranie danych z serwera.
- Nawigacja za pomocą dynamicznego komponentu (vue-router@4).
- Wykorzystać przynajmniej 2 dodatkowe biblioteki lub narzędzia (oprócz Pinia i Router), np.:
  - wtyczkę `vue-good-table-next` (omówiona w instrukcji),
  - bibliotekę `Axios` **LUB** wbudowane w przeglądarkę `Fetch API`.
    - **Ważne:** W przypadku wyboru `Fetch API`, należy zaimplementować własny mechanizm (np. rozbudowując funkcję composable `useFetch` lub tworząc dedykowany serwis), który będzie **automatycznie dołączał token JWT** (pobrany z magazynu Pinia) do nagłówka `Authorization` dla wszystkich chronionych żądań do Twojego API.
- Zarządzanie stanem aplikacji z wykorzystaniem magazynu Pinia.
- Implementacja i użycie własnej wtyczki.
- Wykorzystanie frameworka Tailwind CSS.
- Kilka testów jednostkowych, w tym przynajmniej jednej funkcji wielokrotnego użytku (Composable).
- Kilka testów komponentów.
- Kilka testów E2E w wybranym frameworku (Selenium, Nightwatch …).

Podpowiedzi:

- **Backend:**
  - Do hashowania haseł w Express/Node.js najprościej użyć biblioteki bcryptjs (działa bez kompilacji natywnych modułów).
  - Problem cache'owania pliku stops.json można rozwiązać w prosty sposób: stwórzcie serwis, który pobiera plik raz, zapisuje go (np. w pamięci serwera lub jako plik tymczasowy) i ustawia datę wygaśnięcia (np. na 24 godziny). Każde kolejne żądanie do API będzie dostawało dane z pamięci podręcznej, dopóki nie wygaśnie.

- **Frontend:**
    - Użyjcie Pinia do przechowywania stanu zalogowania (tokena JWT i danych użytkownika).
    - Stwórzcie funkcję composable (na wzór useFetch) lub instancję Axios, która automatycznie dołącza token JWT (pobrany z Pinia) do nagłówka Authorization dla wszystkich zapytań do waszego chronionego API.

Można realizować projekt w grupach 2 osobowych ale wtedy należy wykorzystać architekturę micro-frontends. Do hostowana frontendu można wykorzystać serwer developerski Vite, tutaj przykład: https://dev.to/abhi0498/react-micro-frontends-using-vite-30ah.

Punktacja:

Backend (50 punktów)

- Uwierzytelnianie i Użytkownicy (20 pkt):
    - Poprawna rejestracja z hashowaniem haseł (`bcrypt`). (5 pkt)
    - Endpoint logowania zwracający poprawny token JWT. (5 pkt)
    - Middleware weryfikujący JWT i chroniący trasy. (5 pkt)
    - Konfiguracja Swaggera z obsługą autoryzacji Bearer. (5 pkt)
- Logika Biznesowa i API (20 pkt):
    - Pełne API CRUD do zarządzania zapisanymi przystankami użytkownika. (5 pkt)
    - Główny endpoint API, który pobiera zapisane przystanki z bazy *oraz* łączy je z danymi na żywo z ZTM. (10 pkt)
    - Implementacja cache'owania dla stops.json. (5 pkt)
- Architektura i Baza Danych (10 pkt):
    - Poprawna konfiguracja bazy danych (MongoDB/SQLite) z ODM/ORM (Mongoose/EF Core). (5 pkt)
    - Zastosowanie Obiektów Wartościowych (Value Objects) dla encji (np. dla loginu, hasha hasła). (5 pkt)

Frontend (50 punktów)

- Architektura i Funkcjonalność (20 pkt):
    - Poprawna struktura komponentów (SFC, multikomponenty). (5 pkt)
    - Implementacja routingu (Vue Router) i nawigacji. (5 pkt)
    - Zarządzanie stanem (Pinia) do obsługi logowania i danych użytkownika. (5 pkt)
    - Komponenty poprawnie wyświetlające spersonalizowane dane (CRUD) oraz dane na żywo z ZTM. (5 pkt)
- Zaawansowane Funkcje Vue (15 pkt):
    - Implementacja własnej dyrektywy (np. do zmiany koloru opóźnienia). (5 pkt)
    - Implementacja własnej funkcji Composable (np. `useZtmData`). (5 pkt)
    - Implementacja własnej wtyczki (pluginu). (5 pkt)
- Narzędzia i Testy (15 pkt):
    - Użycie Tailwind CSS do stylizowania. (5 pkt)

- Wykorzystanie 2 dodatkowych wtyczek (np. Axios + biblioteka do dat). (5 pkt)
- Obecność testów (Unit, Component lub E2E) . (5 pkt)

Punkty Dodatkowe (25 pkt):

- Wizualizacja na mapie (Google/OSM): +10 pkt
- Implementacja architektury Micro-frontends (obowiązkowe dla grup 2-osobowych): **+15 pkt (tylko dla studentów realizujących projekt samodzielnie)**