DOKUMENTACJA TECHNICZNA PROJEKTU

Fridge Helper

Autor Projektu

30 listopada 2025

Spis treści

1 Przegląd Projektu

Fridge Helper to pełnostackowa aplikacja webowa zaprojektowana do efektywnego wyszukiwania i zarządzania przepisami kulinarnymi w oparciu o dostępne składniki. Głównym celem systemu jest optymalizacja wykorzystania zasobów spożywczych oraz dostarczanie spersonalizowanych rekomendacji przepisów. Architektura rozwiązania opiera się na integracji grafowej bazy danych Neo4j z nowoczesnym interfejsem webowym, co pozwala na modelowanie złożonych relacji w sposób wydajniejszy niż w tradycyjnych bazach relacyjnych.

2 Wykorzystane Technologie i Rozwiązania

Projekt został zrealizowany w oparciu o nowoczesny stos technologiczny (GRAND Stack approach), implementując kluczowe wzorce architektury oprogramowania:

2.1 Frontend

- **Technologia**: React (TypeScript).
- **Kluczowy aspekt (SPA)**: Aplikacja działa jako Single Page Application. Zapewnia to dynamiczne doświadczenie użytkownika bez konieczności przeładowywania całej strony przy każdej interakcji, co znacząco podnosi responsywność interfejsu.
- **Stylizacja**: Modułowy CSS.

2.2 Backend

- **Technologia**: Node.js + Express.js.
- **Kluczowy aspekt (Separacja Warstw)**: Backend funkcjonuje jako niezależne API RESTful. Taka separacja od warstwy prezentacji pozwala na niezależne skalowanie, testowanie oraz rozwój obu części systemu.
- **Sterownik**: `neo4j-driver` obsługujący protokół Bolt.

2.3 Baza Danych

- **Technologia**: Neo4j (Graph Database).
- **Kluczowy aspekt (Wydajność)**: Wykorzystanie natywnego modelu grafowego pozwala na błyskawiczne zapytania o relacje (np. „znajdź przepisy zawierające moje składniki"). W SQL wymagałoby to kosztownych operacji typu `JOIN`, w Neo4j jest to proste przejście po krawędziach grafu.
- **Hosting**: Neo4j AuraDB (DBaaS) - rozwiązanie w pełni chmurowe.

2.4 Infrastruktura i Wdrożenie

- **Technologia**: Docker.
- **Kluczowy aspekt (Cloud-Native)**: Pełna konteneryzacja obu warstw aplikacji zapewnia przenośność, izolację środowisk oraz łatwość wdrożenia na dowolnej platformie chmurowej (AWS, Azure, VPS).

3 Architektura Systemu

Aplikacja opiera się na architekturze klient-serwer z separacją odpowiedzialności (SoC).

3.1 Architektura Frontendu

Warstwa prezentacji komunikuje się z serwerem asynchronicznie.

- Użytkownik wybiera posiadane składniki w interfejsie.
- Wyniki są renderowane dynamicznie (status: „Ready to Cook!" lub „Missing ingredients").
- Kluczowe komponenty: `Nav.tsx`, `SearchTab.tsx`, moduły administracyjne (`AdminLogin.tsx`).

3.2 Architektura Backendu

Backend przetwarza żądania HTTP, zarządza autoryzacją (middleware `adminAuth.js`) i komunikuje się z bazą danych.

3.3 Publiczne Punkty Końcowe API

`GET /api/ingredients`
Zwraca listę wszystkich unikalnych nazw składników zarejestrowanych w systemie.
Odpowiedź: Tablica JSON z nazwami składników.

`POST /api/search`
Wyszukuje przepisy na podstawie dostarczonej listy składników.
Body: `{ "myIngredients": ["Jajka", "Bacon"] }`
Odpowiedź: Lista przepisów posortowana według liczby brakujących składników.

`POST /api/recipes`
Dodaje nowy przepis do bazy danych. Automatycznie tworzy węzły składników, jeśli te nie istnieją (operacja `MERGE`).

3.3.1 Administracyjne Punkty Końcowe API

Wymagają uwierzytelnienia. Token lub hasło przekazywane jest zgodnie z konfiguracją `ADMIN_PASSWORD`.

`POST /api/admin/login`
Uwierzytelnia administratora.
Zwraca: Token sesji (JWT) lub status sukcesu.

`POST /api/admin/recipes`
Chroniony punkt końcowy. Pozwala dodawać lub aktualizować istniejące przepisy bez ograniczeń użytkownika publicznego.

`DELETE /api/admin/recipes/:id`
Chroniony punkt końcowy. Usuwa trwale przepis o wskazanym identyfikatorze z grafu.

`POST /api/admin/ingredients`
Zarządzanie węzłami składników (np. edycja literówek w nazwach, łączenie duplikatów).

3.4 Model Danych (Graf)

Neo4j wykorzystuje intuicyjny model węzłów i relacji:

- **Węzły**:
  - `:Ingredient {name: string}`
  - `:Recipe {name: string}`
- **Relacje**:
  - `(:Recipe)-[:CONTAINS]->(:Ingredient)`

4 Wdrożenie i Konfiguracja

4.1 Konfiguracja Neo4j AuraDB

1. Utworzenie darmowej instancji w serwisie Neo4j AuraDB.
2. Pobranie danych dostępowych (URI, User, Password).
3. Konfiguracja pliku `.env` w katalogu `backend`:

```
NEO4J_URI=neo4j+s://<instancja>.auradb.ondb.io
NEO4J_USER=neo4j
NEO4J_PASSWORD=<haslo>
ADMIN_PASSWORD=<haslo_admina>
```

4. (Opcjonalnie) Inicjalizacja bazy dostarczonym skryptem Cypher.

4.2 Uruchomienie Lokalne (Docker)

Projekt jest w pełni skonteneryzowany. Aby uruchomić środowisko:

```
docker-compose up --build
```

Usługi będą dostępne pod adresami:

- Backend: http://localhost:2067
- Frontend: http://localhost:3000

4.3 Wersja Produkcyjna

Aplikacja została wdrożona w środowisku produkcyjnym i jest dostępna publicznie pod adresem:

https://fridgehelper.benito.dev

5 Instrukcja Budowania (Bez Dockera)

5.1 Serwer Backendowy

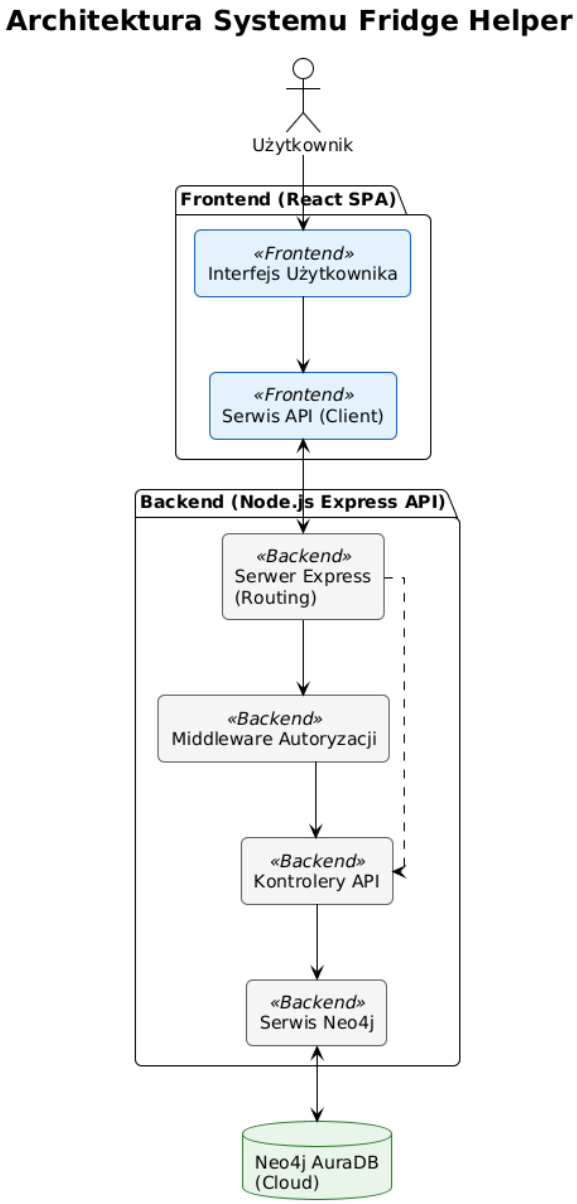
```
cd backend
npm install
npm start
```

5.2 Aplikacja Frontendowa

```
cd frontend
npm install
npm start
```

6 Diagramy UML

Poniższy diagram komponentów ilustruje wysokopoziomową architekturę systemu oraz przepływ danych między warstwami.



Rysunek 1: Architektura Komponentów Systemu Fridge Helper