

Projekt Goodreads

Projekt stara się imitować podzbiór funkcjonalności serwisu Goodreads.com. Projekt jest napisany w języku Javascript. Celem projektu jest nauka programowania w języku Javascript oraz nauka różnych technologii webowych.

Funkcjonalności

Projekt posiada następujące funkcjonalności:

- rejestracja i logowanie
- role użytkowników (`admin`, `moderator`, `user`)
- dodawanie/usuwanie/edycja/pobieranie książek
- dodawanie/usuwanie/edycja/pobieranie własnych półek/list w bibliotece^[1]
- dodawanie/usuwanie książek do własnej kopii predefiniowanej listy: `Want to read`, `Currently reading`, `Read`
- dodawanie/usuwanie książek do/z dowolnie indywidualnie utworzonej półki/listy
- nadawanie/usuwanie/zmiana oceny książki
- dodawanie/usuwanie/edycja komentarzy do książki
- wyszukiwanie po fragmencie tekstu książek, komentarzy, użytkowników
- statystyki per użytkownik i globalne (najlepiej/najgorzej oceniane książki, najczęściej dodawane książki, liczba książek przeczytanych, obecnie czytanych, chcę przeczytać,)

Dodatkowo:

- obserwacja zakładanych kont przez administratora
- integracja ze Slackiem (informacje o nowych użytkownikach)
- komentarz do książki może zostać zablokowany przez comment-moderatora (wystarczy, że w komentarzu znajdzie się słowo `fuck` lub `shit`, lub `free`)

Architektura rozwiązania

Aplikacja składa się z następujących komponentów:

- frontend - aplikacja webowa napisana w języku Javascript przy wykorzystaniu React, będąca interfejsem użytkownika i administratora (dostęp do panelu administratora tylko dla administratora)
- backend - aplikacja webowa napisana w języku Javascript - Node.js, będąca serwerem REST API. Obecnie w API zdefiniowane są następujące endpointy:
 - `/auth` - autoryzacja użytkowników, zakładanie kont, weryfikacja zalogowania (3 metody)

- `/users` - operacje na użytkownikach (pełen CRUD)^{[2][3]}
- `/books` - operacje na książkach (pełen CRUD)^[2]
- `/comments` - operacje na komentarzach (pełen CRUD)^[2]. Komentarze zablokowane przez comment-moderatora/moderatora **nie** są widoczne dla użytkowników chyba że mają rolę `admin` lub `moderator`
- `/shelves` - operacje na półkach/listach (pełen CRUD)^[2]
- `/book-details` - operacje na ocenach, dodawanie/usuwanie z półek/list (pełen CRUD)
- `/stats` - statystyki - globalne i per użytkownik (GET - 4 różne zestawienia)
- `/dbs` - operacje na bazie danych (DELETE - usuwanie kolekcji MongoDB - dostępne tylko dla administratora)
- Dodatkowo serwer backend łączy się z brokerem MQTT w celu wysyłania wiadomości do apletów/microserwisów, oraz potrafi przyjąć zlecenia z systemu kolejkowego w celu zablokowania komentarza ocenionego przez `comment-moderator`.
- baza danych - MongoDB
- system kolejkowy - MQTT
- aplety/microserwisy - działające na zasadzie pub/sub, które reagują na zdarzenia z systemu kolejkowego MQTT. W projekcie zaimplementowane są następujące aplety:
 - `slack-sender` - wysyłanie wiadomości do Slacka
 - `account-watcher` - monitorowanie zakładanych kont użytkowników, oraz wysłanie przez MQTT wiadomości do apletu `slack-sender` o utworzeniu nowego konta
 - `comment-moderator` - automatyczne blokowanie komentarzy zawierających słowa `fuck`, `shit`, `free` (przykład rozwiązania - można się integrować z zewnętrznymi usługami, np. z usługą moderacji komentarzy). Po zablokowaniu komentarza wysyłana jest wiadomość do admina na Slacka (następna integracja pomiędzy microserwisami przez MQTT). Zależnie od klasyfikacji treści zwracany jest inny powód blokady komentarza (np. `forbidden words` lub `spam`).

Obecnie wykorzystywane są następujące tematy (`topic`):

- `goodreads/user/created` - wysyłane z backendu do apletu `account-watcher` - informacja o utworzeniu nowego konta użytkownika
- `goodreads/user/login` - logowanie użytkownika^[4]
- `goodreads/book-details/created` - dodanie książki do półki^[4]
- `goodreads/book-details/updated` - zmiana informacji o półkach, ratingu dla książki^[4]
- `goodreads/slack/send` - wysyłane z apletu `account-watcher` lub `comment-moderator` do apletu `slack-sender` - informacja o utworzeniu nowego konta użytkownika
- `goodreads/comments/created` - wysyłane z backendu do apletu `comment-moderator` - informacja o utworzeniu nowego komentarza
- `goodreads/comments/updated` - wysyłane z backendu do apletu `comment-moderator` - informacja o zmianie komentarza

- `goodreads/comments/blocked` - wysyłane z apletu `comment-moderator` do backendu - informacja o zablokowaniu komentarza. Backend zapisuje powód blokady w bazie danych.

Technologie użyte w projekcie Goodreads.

- backend - API REST serwer
 - operacje CRUD na bazie danych (8 routingów)
 - autoryzacja użytkowników przy pomocy JWT - sesja w headerze HTTP w postaci podpisanego tokena JWT
 - wykorzystanie systemu kolejkowego MQTT - wysyłanie zdarzeń z serwera do kolejki, reakcja mikroserwisów na zdarzenia z kolejki, komunikacja między mikroserwisami
- frontend - aplikacja webowa napisana w języku Javascript - React
 - websockets - do komunikacji z brokerem MQTT
 - wykorzystanie biblioteki React Router do obsługi routingu
 - wykorzystanie biblioteki Redux do zarządzania stanem aplikacji
- baza danych - MongoDB
 - wykorzystanie biblioteki Mongoose do komunikacji z bazą danych
 - operacje na bazie - find, insert, update, delete, aggregate
 - wyszukiwanie w bazie danych jest wykonane na dwa różne sposoby:
 - wyszukiwanie za pomocą wyrażeń regularnych w kilku wybranych polach np. `book.title`, `books.author` - gdzie pola te są połączone za pomocą operatora `$or`. Dzieje się tak w przypadku gdy chcemy przeszukiwać konkretne pola w bazie danych. Tworzony RegExp jest odpowiednikiem `/.search_text./i` - zwraca wszystkie dokumenty w których występuje szukany tekst w dowolnym miejscu tekstu (case insensitive)
 - wyszukiwanie w całym dokumencie za pomocą wyrażenia szukania full-text search (komentarze) - nie ma wtedy opcji wyszukiwania tylko w wybranych polach
 - operacja na samej bazie danych - usuwanie kolekcji (dostępne tylko dla administratora)
- system kolejkowy - MQTT
 - połączenie z frontendu do MQTT przez websockets (broker Mosquitto wystawił swoje API przez websockety). Websockets opakowują protokół MQTT aby umożliwić standardową komunikację z frontendem
 - połączenie z backendu do MQTT przez natywy protokół MQTT

Bezpieczeństwo

Uwierzytelnianie

Uwierzytelnianie jest wykonywane przez endpoint API `/auth/login`. Wymagane jest podanie loginu i hasła użytkownika. W przypadku poprawnego uwierzytelnienia serwer zwraca token JWT. Token

ten jest zapisywany w localStorage przeglądarki. W przypadku każdego zapytania do API serwer sprawdza czy w nagłówku HTTP jest token JWT. Jeśli tak to sprawdza czy token jest poprawny. Jeśli tak to zwraca dane zapytania. Jeśli nie to zwraca błąd 401 Unauthorized. W tokenie JWT zapisywane są dane użytkownika (email, role, username).

Przy zakładaniu użytkownika (endpoint `POST /auth/signup`) serwer sprawdza czy podany email nie jest już zajęty. Jeśli jest to zwraca błąd 409 Conflict. Jeśli nie jest to tworzy nowego użytkownika w bazie danych.

Hasło użytkownika w bazie danych jest hashowane przy pomocy biblioteki `bcrypt` wraz z dodanym `salt`em. Hasło jest hashowane przy rejestracji użytkownika oraz przy logowaniu. W przypadku logowania serwer porównuje hashowane hasło z logowania z hashowanym hasłem zapisanym w bazie danych. Hasło jest niemożliwe do "odzyskania" z hashowanej wersji ponieważ funkcja `hash` jest jednostronna.

Hashowane hasło z bazy danych nie jest zwracane w żadnym zapytaniu do API.

Autoryzacja

Autoryzacja jest wykonywana przez middleware. Zależnie od endpointu oraz zapytania sprawdzane są różne uprawnienia.

Na przykład:

- dla endpointu `GET /users` umożliwiającego otrzymanie listy wszystkich użytkowników uprawnienia pozwalają na pobranie tej listy przez użytkownika z rolą `admin`, ale zwracają błąd przy roli `user`. Przy próbie pobrania danych jednego użytkownika (endpoint `GET /users/:id`) - rola `user` może pobrać tylko swoje dane, a rola `admin` może pobrać dane dowolnego użytkownika. Jeżeli uprawnienia nie są spełnione zwracany jest błąd 403 Forbidden.
- dla endpointu `PATCH /users/:id` umożliwiającego zmianę danych użytkownika - rola `user` pozwala zmienić tylko część swoich danych (password, username, avatar_url etc.). Rola `admin` pozwala także na zmianę atrybutu `role` dla każdego użytkownika. Użytkownik z oczywistych powodów nie ma uprawnień do zrobienia tego samodzielnie.

Cookies

Przy logowaniu ustawiane jest cookie o nazwie `last_login` z czasem życia 1 rok. Cookie to jest używane do wyświetlania informacji o ostatnim logowaniu użytkownika.

Przykładowy response:

```
{
  "message": "OK",
  "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjp7ImVtYWlsIjoiyW5pYUBsb2NhbGhvc3QiLCJyb2x1IjoiaXNlciJ9LCJpYXQiOiJlE2NzUwMjExMDMsImV4cCI6MTY3NTYyNTkwM30uYX08D9xM6ANaIX0AaqZX5RwYY4G7wEF_d1Sl-o0ppo0"
}
```

Scenario: Get comments

```
Given I make a "GET" request to "/comments"
When I receive a response
Then I expect response should have a status 200
And I expect response should have a json like
    """
    [
    {
    "book_id": ${BookId},
    }
    ]
    """
```

W tym scenariuszu testowym sprawdzamy czy po wykonaniu żądania **GET** na adres **/comments** otrzymamy odpowiedź z kodem 200 oraz czy w odpowiedzi znajduje się tablica z obiektami, których pola **book_id** mają wartość zapisaną w zmiennej **BookId** z poprzedniego żądania. Uruchomienie testów odbywa się za pomocą komendy **npm run test** w katalogu **./src**.

Inny przykład testu:

Scenario Outline: Check valid authorization

```
Given I make a "<method>" request to "<endpoint>"
When I receive a response
Then I expect response should have a status <status>
```

Examples:

method	endpoint	status
GET	/users	403
GET	/comments	200
GET	/books	200
GET	/book-details	200
GET	/shelves	200
GET	/stats	200

[1] administrator może dodawać/usuwać/edytować standardowe półki dostępne dla każdego użytkownika: np. **Want to read**, **Currently reading**, **Read**

[2] pełen CRUD oznacza: get all, get by id, create (post), update (patch), delete)

[3] różne uprawnienia dla różnych ról użytkowników

[4] obecnie nic nie nasłuchuje na tym topicu