Księgarnia

A. Identyfikacja zagadnienia biznesowego

Celem niniejszego projektu jest stworzenie prostego sklepu internetowego - księgarni dla klienta który prowadzi stacjonarną księgarnie. W ramach projektu zostanie przygotowany w pełni funkcjonalny sklep internetowy z książkami wyczerpujący wszystkie podstawowe zagadnienia:

- tworzenie kont przez klientów poprzez rejestrację lokalną
- przeglądanie asortymentu wg. kategorii (gatunków książek)
- filtrowanie książek w asortymencie
- wyświetlanie szczegółów wybranej pozycji tj. informacji o wydawnictwie, autorze, cenie.
- możliwość zamówienia wybranej pozycji
- wyświetlenie listy złożonych zamówień

Dodatkowo należy udostępnić odpowiednie narzędzia umożliwiające zarządzanie sklepem internetowym przez administratora:

- dodawanie nowych pozycji do asortymentu
- dodawanie nowych autorów, wydawnictw
- podgląd zamówień

Całość będzie kompletnym systemem umożliwiającym sprzedaż książek w internecie.

B. Określenie wymagań

Aplikacja internetowa pełniąca rolę księgarnii internetowej zostanie podzielona na dwie niezależne części:

- część serwerowa (backend) wykorzystująca środowisko Node.js wraz z szablonem serwera Express.js zawierająca logikę biznesową oraz API dostępowe. Do przechowywania danych zostanie wykorzystana baza typu NoSQL - MongoDB.
- część kliencka (frontend) oparta na React i korzystająca z API udostępnianego przez backend.

Część serwerowa:
Aplikacja backendowa odpowiedzialna będzie za logikę biznesową, komunikację z bazą danych MongoDB oraz udostępnianie interfejsu niezbędnego do komunikacji z aplikacją kliencką.
Cześć zasobów będzie zasobami otwartymi, nie wymagającymi żadnych dodatkowych uprawnień np. pobieranie listy dostępnych pozycji czy wyświetlenie szczegółów książki. Dostęp do zasobów chronionych, czyli takich które wymagają odpowiednich uprawnień np. klienta do złożenia zamówienia czy administracyjnych do dodania nowej pozycji, zostanie zabezpieczony za pomocą JWT Tokenów wydawanych na okaziciela (Beareer token). Uwierzytelnianie użytkowników oparte będzie na strategii lokalnej czyli informacje o kontach klientów zapisane będą w własnej bazie danych.
Cała aplikacja będzie hostowana na dedykowanej maszynie wyposażonej w środowisko uruchomieniowe Node.js, natomiast baza danych zostanie uruchomiona w chmurze.

Część kliencka:
Aplikacja frontendowa zostanie napisana w React, dzięki czemu dostępna będzie pod określonym adresem w internecie za pomocą dowolnej przeglądarki internetowej. Za jej pomocą klienci będą mogli się uwierzytelniać i korzystać z pełnych możliwości serwisu - przeglądać katalog dostępnych pozycji czy składać zamówienia. Dodatkowo dla użytkowników z prawami administracyjnymi zostanie udostępniona specjalna część pozwalająca na administrowanie sklepem.

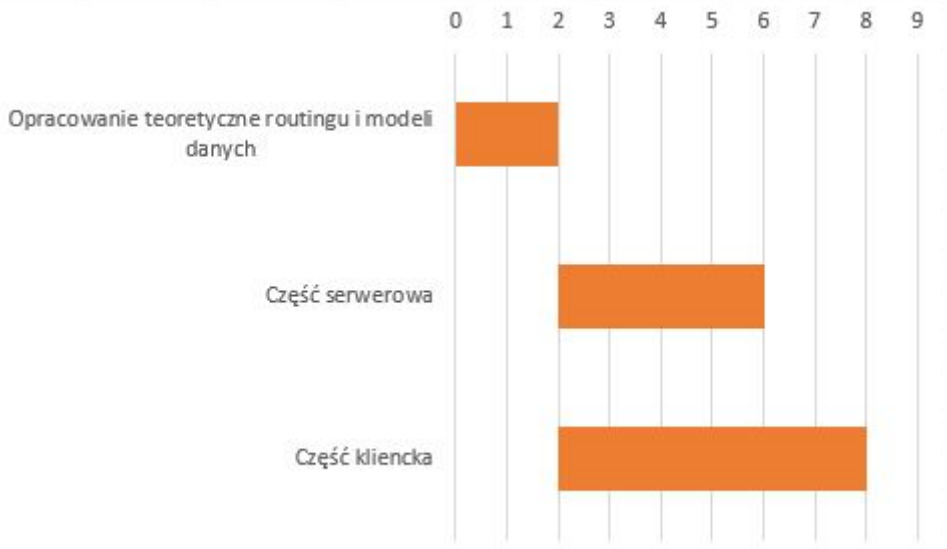
C. Zadaniowy harmonogram prac i podział na członków zespołu

I. **Ustala się następujące zadania do wykonania:**
    1. Opracowanie teoretyczne routingu i modeli danych.
    2. Część serwerowa
        a. Autoryzacja lokalna
            i. strategia lokalna dla użytkownika-klienta (logowania, rejestracja)
            ii. strategia lokalna dla administratora księgarni (logowania, rejestracja przez innego administratora)
        b. Modele danych
        c. Kontrolery dla modeli danych
    3. Część kliencka

Czas wykonywania zadania 1. szacuje się na 2 dni, zadania 2. również na 2 dni, a zadania 3. na 4 dni. Zadanie 1. wykonywane jest przed rozpoczęciem jakichkolwiek kolejnych prac, zadanie 2. i 3. zaś będą wykonywane równocześnie. Do zarządzania i wersjonowania kodu używane będzie narzędzie git, którego to repozytorium będzie wysyłane na serwis github.com.

II. **Ustala się następujący podział prac:**
    1. Zadanie 1 będzie wykonywane przez Macieja Książkiewicza i Sebastiana Zieje
    2. Zadanie 2 będzie wykonywane przez Macieja Książkiewicz i Piotra Dunaja
    3. Zadanie 3 będzie wykonywane przez Piotra Dunaja i Sebastiana Zieje

III. **Wykres Gantta:**



D. Analiza zagadnienia i jego modelowanie

Na potrzeby implementacji funkcjonalności sklepów zaimplementowano następujące modele:

1. **Author**
   Model reprezentujący autora książki. Pola modelu:
   a. **surname** - String, wymagane, nazwisko autora
   b. **name** - String, wymagane, imię autora
2. **BookType**
   Model reprezentujący gatunek książki. Pola modelu:
   a. **name** - String, wymagane, nazwa gatunku książki
3. **PublishingHouse**
   Model reprezentujący wydawnictwa. Pola modelu:
   a. **name** - String, wymagane, nazwa wydawnictwa
4. **Book**
   Model reprezentujący książkę. Pola modelu:
   a. **title** - String, wymagane, tytuł książki
   b. **price** - Number, wymagane, cena książki
   c. **publishYear** - Number, wymagane, rok wydania książki
   d. **description** - String, wymagane, opis książki
   e. **author** - ObjectId, wymagane, id autora książki
   f. **bookType** - ObjectId, wymagane, id gatunku książki
   g. **publishingHouse** - ObjectId, wymagane, id wydawnictwa
5. **User**
   Model reprezentujący klienta sklepu. Pola modelu:
   a. **email** - String, unikalne, email użytkownika
   b. **password** - String, hash hasła użytkownika
   c. **googleId** - String, id zwracane przy logowaniu z użyciem google'a
   d. **facebookId** - String, id zwracane przy logowaniu z użyciem facebooka
   e. **isAdmin** - Boolean, flaga oznaczająca administratora
6. **Order**
   Model reprezentujący zamówienie. Pola modelu:
   a. **userId** - ObjectId, wymagane, id użytkownika składającego zamówienie
   b. **books** - tablica (wymagana) zawierająca obiekty o następującej strukturze:
      i. **bookId** - ObjectId, wymagane, id zamówionej książki
      ii. **quantity** - Number, wymagane, ilość zamówionych sztuk książki
      iii. **price** - Number, wymagane, cena książki w momencie zakupu

Dodatkowo przyjęto następujące endpointy do obsługi zapytań:

1. **<u>Uwierzytelnianie:</u>**
    a. **POST /api/auth/register**
    Rejestracja nowego klienta sklepu. Jako ciało przyjmuje się model User.
    b. **POST /api/auth/login**
    Logowanie lokalne klienta sklepu. Jako ciało przyjmuje się model User. Zwracany jest token użytkownika.
2. **<u>Dostęp do książek:</u>**
    a. **GET /api/book**
    Zwraca wszystkie książki zgodnie z modelem Book.
    b. **GET /api/book/:id**
    Zwraca książkę o podanym **id** zgodnie z modelem Book.
    c. **POST /api/book**
    Dodaje nową książkę. Jako ciało przyjmuje się model Book. Wymagane jest podanie tokenu administratora w nagłówku x-access-token.
    d. **DELETE /api/book**
    Usuwa książkę. Jako ciało przyjmuje się podanie w polu **id** id książki. Wymagane jest podanie tokenu administratora w nagłówku x-access-token.
    e. **PATCH /api/book**
    Edytuje książki. Jako ciało przyjmuje się model Book wraz z uzupełnionym polem **id**. Wymagane jest podanie tokenu administratora w nagłówku x-access-token.
3. **<u>Dostęp do gatunków książek</u>**
    a. **GET /api/bookType**
    Zwraca wszystkie gatunki książek zgodnie z modelem BookType.
    b. **GET /api/bookType/:id**
    Zwraca gatunek książki o podanym **id** zgodnie z model BookType.
    c. **POST /api/bookType**
    Dodaje nowy gatunek książki. Jako ciało przyjmuje się model BookType. Wymagane jest podanie tokenu administratora w nagłówku x-access-token.
    d. **DELETE /api/bookType**
    Usuwa gatunek książki. Jako ciało przyjmuje się podanie w polu **id** id gatunku książki. Wymagane jest podanie tokenu administratora w nagłówku x-access-token.
    e. **PATCH /api/bookType**
    Edytuje gatunek książki. Jako ciało przyjmuje się model BookType wraz z uzupełnionym polem **id**. Wymagane jest podanie tokenu administratora w nagłówku x-access-token.
4. **<u>Dostęp do wydawnictw</u>**
    a. **GET /api/publishingHouse**
    Zwraca wszystkie wydawnictwa zgodnie z modelem PublishingHouse.

b. **GET /api/publishingHouse/:id**
   Zwraca wydawnictwo o podanym **id** zgodnie z modelem PublishingHouse.
c. **POST /api/publishingHouse**
   Dodaje nowe wydawnictwo. Jako ciało przyjmuje się model PublishingHouse. Wymagane jest podanie tokenu administratora w nagłówku x-access-token.
d. **DELETE /api/publishingHouse**
   Usuwa wydawnictwo. Jako ciało przyjmuje się podanie w polu **id** id wydawnictwa. Wymagane jest podanie tokenu administratora w nagłówku x-access-token.
e. **PATCH /api/publishingHouse**
   Edytuje wydawnictwo. Jako ciało przyjmuje się model PublishingHouse wraz z uzupełnionym polem **id**. Wymagane jest podanie tokenu administratora w nagłówku x-access-token.

5. **<u>Dostęp do autorów</u>**
   a. **GET /api/author**
      Zwraca wszystkich autorów zgodnie z modelem Author.
   b. **GET /api/author/:id**
      Zwraca autora o podanym **id** zgodnie z modelem Author.
   c. **POST /api/author**
      Dodaje nowego autora. Jako ciało przyjmuje się model Author. Wymagane jest podanie tokenu administratora w nagłówku x-access-token.
   d. **DELETE /api/author**
      Usuwa autora. Jako ciało przyjmuje się podanie w polu **id** id autora. Wymagane jest podanie tokenu administratora w nagłówku x-access-token.
   e. **PATCH /api/author**
      Edytuje autora. Jako ciało przyjmuje się model Author wraz z uzupełnionym polem **id**. Wymagane jest podanie tokenu administratora w nagłówku x-access-token.
6. **<u>Dostęp do zamówień</u>**
   a. **GET /api/order**
      Zwraca wszystkie zamówienia zgodnie z modelem Order. Wymagane jest podanie tokenu użytkownika w nagłówku x-access-token.
   b. **GET /api/order/:id**
      Zwraca zamówienie o podanym **id** zgodnie z modelem Order. Wymagane jest podanie tokenu użytkownika w nagłówku x-access-token.
   c. **POST /api/order**
      Dodaje nowe zamówienie. Jako ciało przyjmuje się obiekt zgodny z polem books modelu Order. Wymagane jest podanie tokenu użytkownika w nagłówku x-access-token.
   d. **DELETE /api/order**

Usuwa zamówienie. Jako ciało przyjmuje się podanie w polu **id** id zamówienia. Wymagane jest podanie tokenu użytkownika w nagłówku x-access-token.

E. Implementacja

Część serwerowa została napisana w środowisku Node.js wykorzystując szablon serwera Express. Jako system utrwalania danych wybrano bazę danych NoSQL - MongoDB.

a) mongoose - biblioteka mapująca obiekty bazy danych (kolekcje) na obiekty JS. Umożliwia wykonywania zapytań na bazie danych za pomocą metod udostępnianych przez obiekty mapujące.

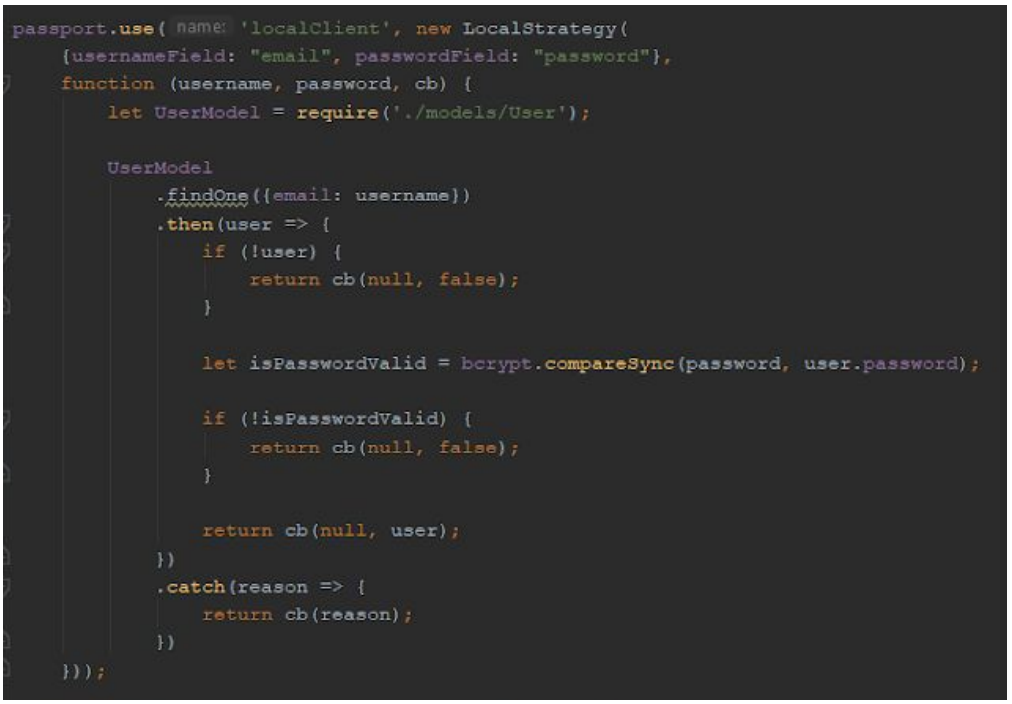
```
let mongoose = require('mongoose');

let AuthorSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true
    },
    surname: {
        type: String,
        required: true
    }
});

mongoose.model( name: 'Author', AuthorSchema);

module.exports = mongoose.model( name: 'Author');
```

Przykładowy schemat kolekcji dla autora. Definiuje pola imię oraz nazwisko typu String. Są to pola zdefiniowane jako obowiązkowe. Bez ich określenia nie będzie możliwe utworzenie wpisu w kolekcji.

```
AuthorModel
    .findOne({"_id": authorId})
    .then(author => {
        return res
            .status(200)
            .json(author)
    })
    .catch(reason => {
        sendApiError(res, code: 500, message: "Couldn't download author: " + reason.message)
    });
```

Przykładowe zapytania zwracające autora o konkretnym ID.

b) Passport - biblioteka dostarczająca mechanizmy autoryzacji w środowisku Node.js. W projekcie użyto dwóch strategii lokalnych - dla użytkownika-klienta oraz administratora.



```
passport.use( name: 'localClient', new LocalStrategy(
    {usernameField: "email", passwordField: "password"},
    function (username, password, cb) {
        let UserModel = require('./models/User');

        UserModel
            .findOne({email: username})
            .then(user => {
                if (!user) {
                    return cb(null, false);
                }

                let isPasswordValid = bcrypt.compareSync(password, user.password);

                if (!isPasswordValid) {
                    return cb(null, false);
                }

                return cb(null, user);
            })
            .catch(reason => {
                return cb(reason);
            })
    }));
```

Implementacja strategii lokalnej dla klienta.

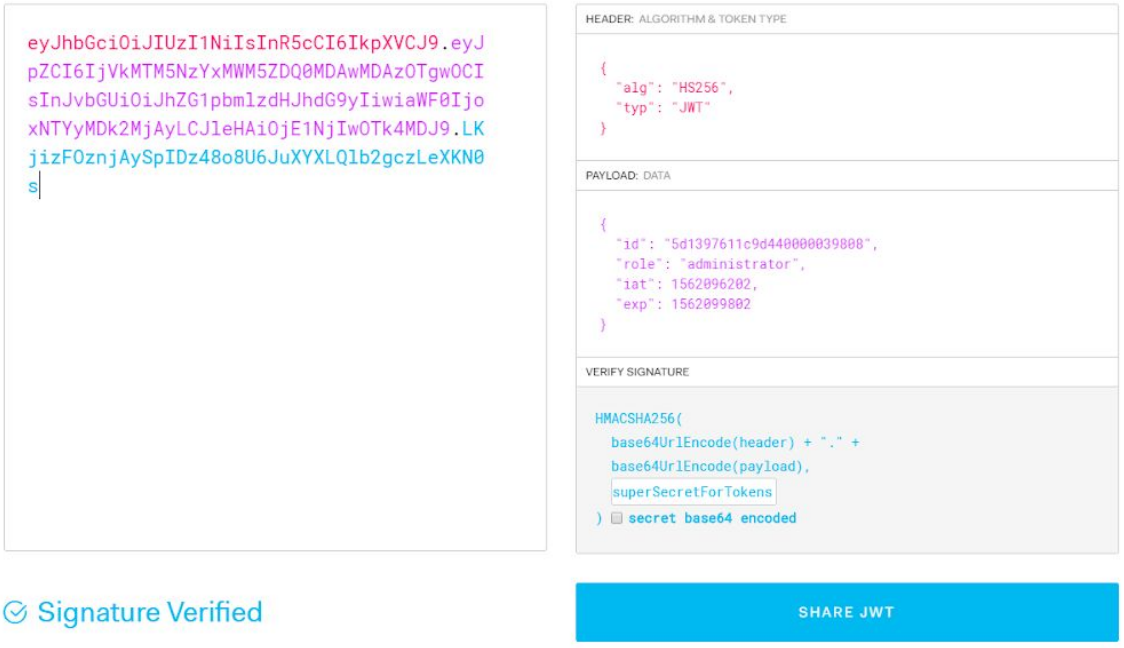
c) jsowebtoken - biblioteka umożliwiająca pracę z tokenami JWT. Dostęp do API został zabezpieczony przez zastosowanie tokenów JWT - wykonanie pewnych endpointów wymaga zidentyfikowanie się tokenem na okaziciela, który jest wydawany po poprawnym zalogowaniu.

```
router.post('/administrator/login', passport.authenticate( strategy: 'localAdministrator', options: {failureRedirect
    let userId = req.user._id;

    let token = jwt.sign({id: userId, role: "administrator"}, config.jwtSecret, {expiresIn: config.jwtTime});

    sendApiToken(res, token)
});
```

Metoda odpowiedzialna za wydanie tokenu po poprawnym zalogowaniu przez administratora. W tokenie zapisywany jest id użytkownika oraz jego rola w serwisie. Taki token następnie jest szyfrowany za pomocą sekretu oraz wydawany na pewien określony czas.

Taki token jest następnie przesyłany w headerze do zabezpieczonego endpointa i weryfikowany w celu dopuszczenia użytkownika do chronionego zasobu.

```
function verifyToken(req, res, next) {
    let token = req.headers[config.jwtHeader];

    if (!token) {
        res
            .status(403)
            .send(ApiUtils.getApiError("There is no token in headers"));
        return;
    }

    jwt.verify(token, config.jwtSecret, function (err, decoded) {
        if (err) {
            res
                .status(500)
                .send(ApiUtils.getApiError("Token is incorrect. Reason: " + err.message));
            return;
        }
        req.userId = decoded.id;
        next();
    });
}

module.exports = verifyToken;
```



Struktura JWT tokenu

F. Testowanie

Nasze oprogramowanie przeszło szereg niezwykle wymagających testów manualnych na GUI. Samo API było również testowane za pomocą Postmana (https://www.getpostman.com/). Do tego każda zmiana musiała przejść przez code review na platformie github.

G. Instalacja/deployment/instrukcja użytkownika

I. Instalacja
   1. Należy ściągnąć repozytorium z aplikacją (https://github.com/Raveor/bookstore.git)
   2. W tym repozytorium należy wywołać komendę `npm/yarn install`
   3. Następnie należy przejść do katalogu client (`cd client`) i tam również wywołać komendę `npm/yarn install`

II. Uruchomienie na systemie Windows

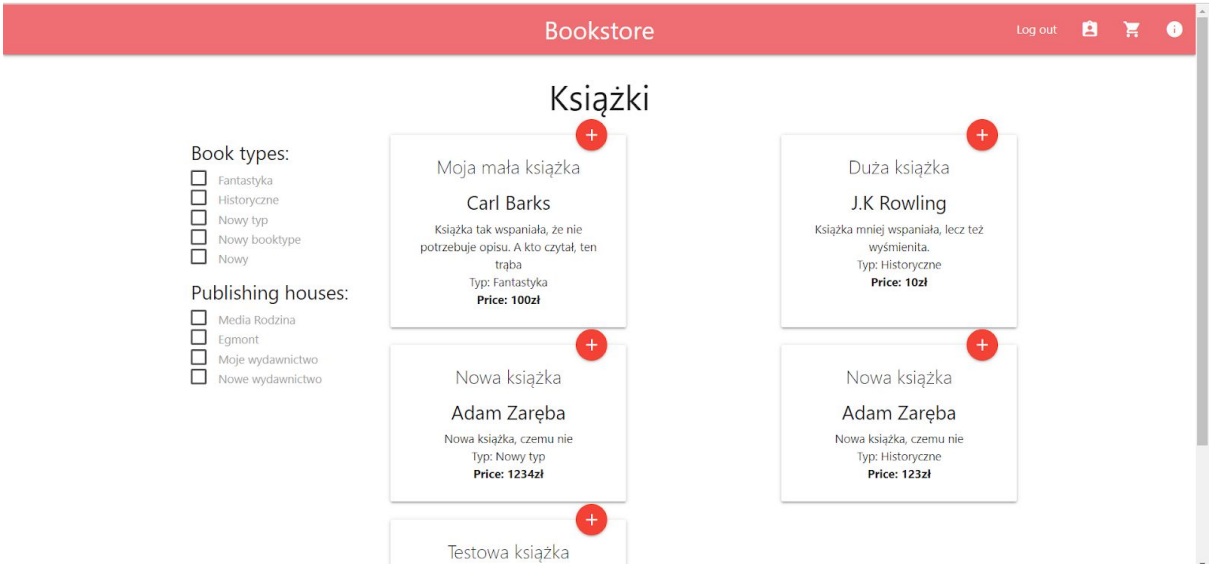
W konsoli w głównym folderze repozytorium należy wywołać komendę `npm run-script dev` Aplikacja powinna samoczynnie się uruchomić po zbudowaniu (jeżeli tak się nie stanie, powinna być dostępna pod adresem http://localhost:5000/).

III. Uruchomienie na systemie Linux

W konsoli w głównym folderze repozytorium należy wywołać komendę `npm start`
W drugiej konsoli zaś należy przejść do folderu client repozytorium i wywołać tam komendę `npm start` Aplikacja powinna samoczynnie się uruchomić po zbudowaniu (jeżeli tak się nie stanie, powinna być dostępna pod adresem http://localhost:5000/).

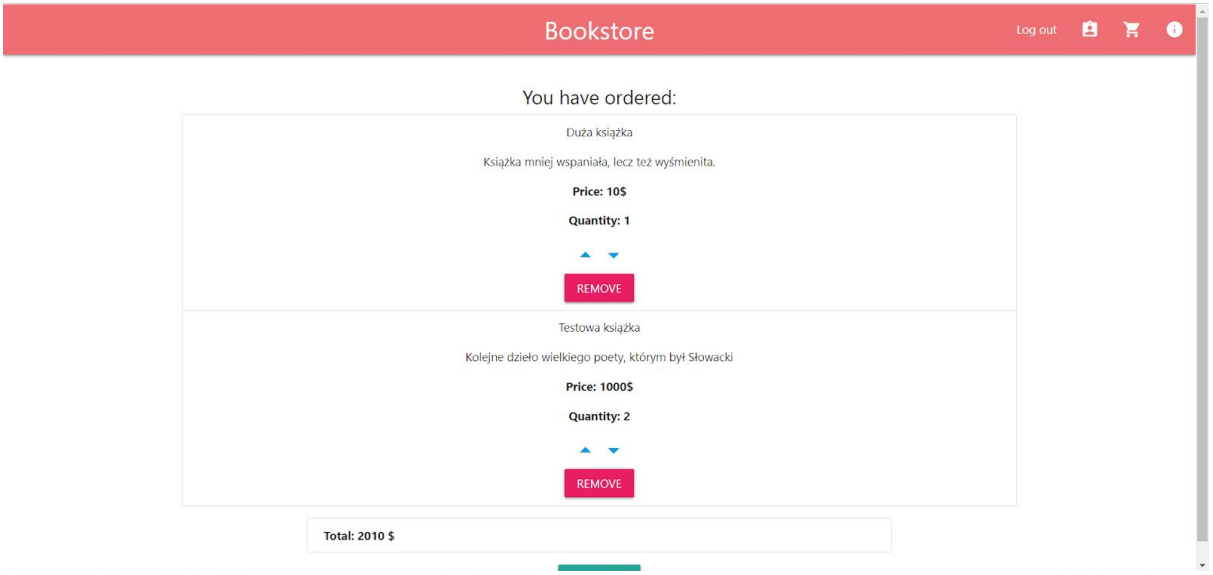
IV. Skrócona instrukcja użytkownika

Po otwarciu strony ze sklepem naszym oczom powinien się ukazać poniższy ekran:



W tym miejscu możemy obejrzeć dostępne książki, przefiltrować je (opcje po lewej stronie) oraz dodać jakąś do koszyka naciskając + przy którejś książce. Za pomocą przycisków w prawym górnym rogu możemy kolejno - zobaczyć listę zamówień, przejrzeć koszyk i przejść do panelu administratora.

Po przejściu do ekranu z koszykiem wyświetla nam się poniższy ekran:



Tutaj możemy zmienić ilość zamawianych książek, usunąć je z koszyka i co najważniejsze wysłać zamówienie.

Po przejściu do panelu administratorskiego widzimy poniższy ekran z opcjami przejrzenia wszystkich zamówień, dodania nowych książek, gatunków, autorów i wydawnictw.