Dokumentacja projektu "Booka"

Miłosz Białczak (218295)
Mateusz Gniewkowski (218138)
Beata Szeląg (218139)

20 czerwca 2017

Spis treści

Rozdział 1

Wstęp

Celem projektu jest zaproponowanie, zaprojektowanie, implementacja i udokumentowanie systemu opartego o technologię Java EE. System jaki zamierzamy stworzyć ma umożliwić zarządzanie domową biblioteczką przeciętnemu użytkownikowi komputera. System powinien przede wszystkim dawać możliwość katalogowania i kategoryzowania książek z dowolnego punktu dostępowego. Wskazane jest więc użycie technologii pozwalających na stworzenie aplikacji webowej, które to wybraliśmy na podstawie analizy postawionych przez nas wymagań. Projekt obejmuje więc:

1. określenie celów
2. pomysł realizacji postawionych celów
3. analizę wymagań
4. wybór technologii
5. zaprojektowanie systemu
6. implementację systemu
7. dokumentację techniczną (w postaci niniejszego dokumentu)

1.1 Uzasadnienie biznesowe

Wraz ze wzrostem liczby książek coraz większą trudność sprawia utrzymanie ich w porządku, zwłaszcza jeżeli część książek została pożyczona od kogoś lub komuś - stąd pomysł na naszą aplikację. Nasza aplikacja (dalej zwana "Booką") ma umożliwić wygodne zarządzanie księgozbiorem. Daje ona możliwość zaznaczenia kto daną książkę pożyczył i kiedy powinien ją oddać oraz wiele innych przydatnych funkcjonalności, które zostaną opisane dogłębniej w niniejszym dokumencie.

Rozdział 2

Analiza systemu

2.1 Opis działania systemu

Systemem, który stworzono jest prosta aplikacja internetowa. Po uruchomieniu przeglądarki i wpisaniu odpowiedniego adresu użytkownikowi powinna pokazać się strona umożliwiająca zalogowanie się, lub rejestrację do systemu. Po zalogowaniu, strona przenosi użytkownika do strony głównej, z której użytkownik ma dostęp do swojego księgozbioru (rozumie się przez to możliwość dodawania, usuwania książek, jak i przeglądania go), do podglądu swoich znajomych, podglądania książek im i od nich pożyczonych, chatu, widoku umożliwiającego przeszukiwanie okolicznych bibliotek i do swojego konta Google (po uprzednim zalogowaniu się do niego), umożliwiającego przechowywania elektronicznych wersji książek w chmurze.

2.2 Wymagania funkcjonalne

Tablica 2.1: Wymaganie funkcjonalne F_00

Utworzenie konta	
ID	F_00
Opis	Użytkownicy nieposiadający kont mają możliwość ich założenia. Po wybraniu odpowiedniej opcji, na podany adres mailowy zostaje wysłana wiadomość zawierająca link potwierdzający rejestrację.
Priorytet	wymagane
Powiązania	-

Tablica 2.2: Wymaganie funkcjonalne F_01

Logowanie

ID	F_01
Opis	Użytkownik posiadający konto może zalogować się do systemu
Priorytet	wymagane
Powiązania	-

Tablica 2.3: Wymaganie funkcjonalne F_02

Wylogowanie	
ID	F_02
Opis	Zalogowany użytkownik może się wylogować
Priorytet	wymagane
Powiązania	-

Tablica 2.4: Wymaganie funkcjonalne F_03

Konfiguracja	
ID	F_03
Opis	Użytkownik ma możliwość konfiguracji swojego konta, a w szczególności ustawienia i zmiany hasła
Priorytet	wymagane
Powiązania	-

Tablica 2.5: Wymaganie funkcjonalne F_04

Dodawanie pozycji do księgozbioru	
ID	F_04
Opis	Użytkownik może dodać nową pozycję do swojego księgozbioru. Powinien on określić jej tytuł, autora i kategorię. Jeżeli użytkownik posiada elektroniczną kopię książki na dysku lokalnym, lub obsługiwanym dysku sieciowym (serwer FPT), może ją zaimportować automatycznie. Jest możliwe importowanie całych katalogów naraz - aplikacja powinna reagować na zmiany w danym katalogu i automatycznie importować ich zawartość.
Priorytet	wymagane
Powiązania	-

Tablica 2.6: Wymaganie funkcjonalne F_05

Usuwanie pozycji z księgozbioru	
ID	F_05
Opis	Użytkownik może usunąć dowolną pozycję ze swojego księgozbioru.
Priorytet	wymagane

Powiązania	-

Tablica 2.7: Wymaganie funkcjonalne F_06

Podgląd księgozbioru	
ID	F_06
Opis	Użytkownik może przeglądać swój księgozbiór.
Priorytet	wymagane
Powiązania	-

Tablica 2.8: Wymaganie funkcjonalne F_07

Przeszukiwanie księgozbioru	
ID	F_07
Opis	Użytkownik może przeszukiwać swój księgozbiór po nazwie, autorze, kategorii itp.
Priorytet	wymagane
Powiązania	F_06

Tablica 2.9: Wymaganie funkcjonalne F_08

Przeszukiwanie zasobów bibliotecznych	
ID	F_08
Opis	Użytkownik powinien mieć możliwość przeglądania zasobów bibliotek miejskich.
Priorytet	wymagane
Powiązania	-

Tablica 2.10: Wymaganie funkcjonalne F_09

Przeszukiwanie sklepów internetowych	
ID	F_09
Opis	Użytkownik powinien mieć możliwość wyszukiwania książek w zdefiniowanych sklepach
Priorytet	oczekiwane
Powiązania	-

Tablica 2.11: Wymaganie funkcjonalne F_10

Pożyczanie książek

ID	F_10
Opis	Użytkownik powinien mieć możliwość oznaczenia książki jako pożyczonej (z uwzględnieniem komu owa książka została pożyczona). Informacja o pożyczonych komuś i od kogoś książkach powinna być dostępna w widoku księgozbioru.
Priorytet	wymagane
Powiązania	F_06, F_11

Tablica 2.12: Wymaganie funkcjonalne F_11

Oddawanie książek	
ID	F_11
Opis	System powinien dawać możliwość oddawania książek.
Priorytet	wymagane
Powiązania	F_06, F_10

Tablica 2.13: Wymaganie funkcjonalne F_12

System notyfikacji	
ID	F_12
Opis	System powinien automatycznie informować użytkownika o akcjach z nim związanych (np. informacja o zbliżającym się terminie oddania książki). Notyfikacje powinny być możliwe do modyfikacji i wyłączenia.
Priorytet	oczekiwane
Powiązania	-

Tablica 2.14: Wymaganie funkcjonalne F_13

Dostęp do pozycji w wersjach elektronicznych	
ID	F_13
Opis	Jeżeli książka jest dostępna w wersji elektronicznej, powinno być możliwe otworzenie jej z poziomu aplikacji.
Priorytet	oczekiwane
Powiązania	-

Tablica 2.15: Wymaganie funkcjonalne F_14

Udostępnianie księgozbioru	
ID	F_14
Opis	Użytkownik powinien mieć możliwość udostępnienia swojego księgozbioru do podglądu innym użytkownikom (niekoniecznie z opcją czytania książek w wersji elektronicznej)

Priorytet	opcjonalne
Powiązania	-

Tablica 2.16: Wymaganie funkcjonalne F_15

Wysyłanie powiadomień do innych użytkowników	
ID	F_15
Opis	Użytkownik powinien mieć możliwość wysyłania powiadomień innym użytkownikom. Powiadomienia mogą dotyczyć próśb o pożyczenie, oddanie książki, udostępnienie podglądu księgozbioru itp. Powiadomienia mogą być wysyłane poprzez Facebooka, e-maila, lub aplikację.
Priorytet	oczekiwane
Powiązania	-

2.3 Wymagania niefunkcjonalne

Tablica 2.17: Wymaganie niefunkcjonalne N_00

Interfejs użytkownika	
ID	N_00
Opis	Aplikacja powinna posiadać ładny i przejrzysty graficzny interfejs użytkownika, możliwe intuicyjny i łatwy w obsłudze.
Priorytet	oczekiwane
Powiązania	-

Tablica 2.18: Wymaganie niefunkcjonalne N_01

Odporność na utratę danych	
ID	N_01
Opis	Stworzenie mechanizmów w systemie odpowiedzialnych za cykliczną archiwizację stanu bazy danych oraz możliwość wczytania jej z pliku.
Priorytet	oczekiwane
Powiązania	-

Tablica 2.19: Wymaganie niefunkcjonalne N_02

Skalowalność	
ID	N_02
Opis	System powinien zapewnić możliwość jego rozbudowy pod względem rozmiaru
Priorytet	oczekiwane
Powiązania	-

Tablica 2.20: Wymaganie niefunkcjonalne N_03

Bezpieczeństwo danych	
ID	N_03
Opis	System powinien dbać o zapewnienie ograniczonego dostępu do przechowywanych informacji.
Priorytet	wymagane
Powiązania	-

Tablica 2.21: Wymaganie niefunkcjonalne N_04

Wydajność	
ID	N_04
Opis	System powinien reagować na zapytania użytkownika z opóźnieniem nie większym niż pół sekundy.
Priorytet	oczekiwane
Powiązania	-

Tablica 2.22: Wymaganie niefunkcjonalne N_05

Możliwość rozwoju	
ID	N_05
Opis	System powinien być zaplanowany w ten sposób, aby umożliwić dołączanie nowych funkcjonalności.
Priorytet	oczekiwane
Powiązania	-

2.4 Wymagania technologiczne

Tablica 2.23: Wymaganie technologiczne T_00

Java	
ID	T_00
Opis	Aplikacja powinna być stworzona w języku Java, wersja 8 lub wyższa.
Priorytet	wymagane
Powiązania	-

Tablica 2.24: Wymaganie technologiczne T_01

SpringBoot	
ID	T_01

Opis	Serwer aplikacji powinien być napisany z wykorzystaniem framework'a Spring Boot.
Priorytet	wymagane
Powiązania	-

Tablica 2.25: Wymaganie technologiczne T_02

MySQL	
ID	T_02
Opis	Wykorzystywanym systemem bazodanowym powinien być MySQL.
Priorytet	wymagane
Powiązania	-

Tablica 2.26: Wymaganie technologiczne T_03

Aplikacja webowa	
ID	T_03
Opis	Aplikacja webowa powinna być napisana z wykorzystaniem technologii CSS, HTML i JavaScript z framework'iem AngularJS
Priorytet	wymagane
Powiązania	-

2.5 Przypadki użycia

Użyte skróty:

- WP - warunki początkowe
- WK - warunki końcowe

Tablica 2.27: Przypadek użycia PU_00

Zakładanie konta	
ID	PU_00
Cel	Możliwość stworzenia nowego konta do systemu
WP	-
WK	Poprawne zarejestrowanie nowego użytkownika
Przebieg	1. Należy wybrać opcję 'Sign up'. 2. Należy podać imię, nazwisko, email, hasło oraz opcjonalnie wypełnić dodatkowe pola 3. Należy zatwierdzić formularz 4. Na email zostanie przesłany link potwierdzający rejestrację, który należy kliknąć

Tablica 2.28: Przypadek użycia PU_01

Logowanie	
ID	PU_01
Cel	Możliwość zalogowania się do systemu
WP	Poprawnie utworzone konto użytkownika
WK	Zalogowanie się do systemu
Przebieg	1. Należy wybrać opcję 'Sign in'. 2. Należy podać email oraz hasło 3. Należy zatwierdzić dane

Tablica 2.29: Przypadek użycia PU_02

Edycja danych użytkownika	
ID	PU_02
Cel	Możliwość edycji danych użytkownika
WP	Poprawnie zalogowanie się do systemu
WK	Zmiana danych użytkownika na nowe
Przebieg	1. Należy wybrać opcję 'Settings' oraz 'Profile'. 2. Należy wybrać interesujące nas dane, które chcemy zmienić i wybrać opcję 'Edit' 3. Po wprowadzeniu danych należy zatwierdzić zmiany

Tablica 2.30: Przypadek użycia PU_03

Przeglądanie księgozbioru	
ID	PU_03
Cel	Możliwość przeglądania wszystkich pozycji w księgozbiorze
WP	Poprawnie zalogowanie się
WK	Wyświetlenie listy książek
Przebieg	1. Należy wybrać opcję 'Books'

Tablica 2.31: Przypadek użycia PU_04

Wyszukiwanie książek z własnego księgozbioru	
ID	PU_04
Cel	Możliwość przeglądania wybranych pozycji w księgozbiorze
WP	Poprawnie zalogowanie się
WK	Wyświetlenie listy książek
Przebieg	1. Należy wybrać opcję 'Books' 2. Należy wypełnić jeden/kilka filtrów: filtr imion autorów ('Name'), nazwisk autorów ('Surname'), tytułów ('Title'), tagów ('Tag')

Tablica 2.32: Przypadek użycia PU_05

Dodanie nowej książki do księgozbioru	
ID	PU_05
Cel	Możliwość dodania nowej pozycji do księgozbioru
WP	Poprawnie zalogowanie się
WK	Dodanie nowej pozycji do księgozbioru
Przebieg	1. Należy wybrać opcję 'Books' i 'Add book' 2. Należy uzupełnić formularz, podając imię i nazwisko autora, tytuł książki i opcjonalnie wypełnić pozostałe pola 3. Należy zatwierdzić formularz

Tablica 2.33: Przypadek użycia PU_06

Edycja danych książki	
ID	PU_06
Cel	Możliwość edycji informacji o danej książce
WP	Poprawnie zalogowanie się
WK	Zmiana danych dotyczących danej książki
Przebieg	1. Należy wybrać opcję 'Books' 2. Należy wybrać daną książkę poprzez 'See details' 3. Należy wybrać interesujące nas dane, które chcemy zmienić i wybrać opcję 'Edit' 4. Po wprowadzeniu danych należy zatwierdzić zmiany

Tablica 2.34: Przypadek użycia PU_07

Usuwanie książki	
ID	PU_07
Cel	Możliwość usunięcia danej pozycji z księgozbioru
WP	Poprawnie zalogowanie się
WK	Usunięcie pozycji z księgozbioru
Przebieg	1. Należy wybrać opcję 'Books' 2. Należy wybrać daną książkę poprzez 'See details' 3. Należy wybrać opcję 'Delete' 4. W dodatkowym oknie należy potwierdzić chęć usunięcia danej pozycji (opcja 'Yes')

Tablica 2.35: Przypadek użycia PU_08

Podgląd książki elektronicznej	
ID	PU_08

Cel	Możliwość otworzenia książki elektronicznej
WP	Poprawnie zalogowanie się oraz posiadanie książki/książek elektronicznych w księgozbiorze
WK	Otworzenie książki w programie typu Adobe Reader / Foxit
Przebieg	1. Należy wybrać opcję 'Books' 2. Należy wybrać daną pozycję 3. Należy wybrać ikonę książki

Tablica 2.36: Przypadek użycia PU_09

Pożyczenie książki z własnego księgozbioru	
ID	PU_09
Cel	Możliwość oznaczenia własnej książki jako pożyczonej wraz z oznaczeniem komu i kiedy została ona pożyczona
WP	Poprawnie zalogowanie się oraz posiadanie książki/książek w księgozbiorze
WK	Oznaczenie książki jako pożyczonej / Foxit
Przebieg	1. Należy wybrać opcję 'Books' 2. Należy wybrać daną pozycję 3. Należy wybrać opcję 'See details' oraz 'Lend' 4. Należy wybrać komu pożyczamy książkę (podajemy login osoby, jeśli posiada ona konto w systemie, w przeciwnym wypadku - dowolną nazwę) oraz opcjonalnie uzupełniamy resztę informacji (np: określenie daty zwrotu). 5. Zatwierdzamy formularz

Rozdział 3

Projekt systemu

3.1 Serwer aplikacji

Naszą aplikację oparliśmy o REST API (Representational State Transfer Application Programming Interface) - popularną, bezstanową architekturę umożliwiającą na prostą komunikację pomiędzy systemami (bądź podsystemami). REST API, polega na udostępnieniu punktów (tzw. punktów końcowych - endpoints) w postaci adresu URL. Odwołując się na taki adres żądamy określonej akcji od serwera. Poniżej przedstawiono spis wszystkich endpointów.

3.1.1 REST API

Tablica 3.1: Akcje związane z użytkownikami

Użytkownicy /users			
Endpoint	Request	Opis	Dodatkowo
/users/	GET	Pobranie informacji o użytkownikach	-
/users/	PUT	Zmiana danych użytkownika	body: id, login, password, email, name, surname
/users/<user_id>	GET	Wyświetlanie informacji o użytkowniku	-
/users/<user_id>	DELETE	Usuwanie konta użytkownika	HttpServletResponse: response
/users/hash	POST	Haszowanie haseł użytkowników	-
/users/session	GET	Pobranie informacji sesji użytkownika	cookie: sessionToken
/users/search/<query>	GET	Wyszukiwanie użytkowników	-

/users/confirm/<token>	GET	Potwierdzenie założenia konta	-
/users/sign_in	POST	Logowanie	body: login, password HttpServletResponse: response
/users/sign_out	POST	Wylogowywanie	cookie: sessionToken HttpServletResponse: response
/users/sign_up	POST	Tworzenie nowego konta	body: login, password, email, name, surname

Tablica 3.2: Akcje związane z książkami

Książki /books			
Endpoint	Request	Opis	Dodatkowo
/books	PUT	Zmiana danych książki	body: id, title, author, format, path, status, user_type, user, department
/books	POST	Dodanie książki	body: title, author, format, path, status, user_type, user, department
/books/<book_id>	GET	Wyświetlanie informacji o danej książce	-
/books/<book_id>	DELETE	Usuwanie książki	-
/books/addTagToBook	POST	Dodanie tagu do książki	body: tagBook
/books/removeTagFromBook	DELETE	Usuwa tag z książki	body: tagBook
/books/user/<user_id>	GET	Wyświetlanie wszystkich książek użytkownika	cookie: sessionToken
/books/lend	PUT	Edycja danych o wypożyczeniu	body: id, book, borrower, message, date_start, date_stop, name, email, facebook
/books/lend	POST	Wypożyczanie książki	body: id, book, borrower, message, date_start, date_stop, name, email, facebook
/books/lend/user	GET	Wyświetlenie informacji o książkach które użytkownik wypożyczył	cookie: sessionToken
/books/lend/<lend_id>	GET	Wyświetlenie informacji o wypożyczeniu	-
/books/lend/<land_id>	DELETE	Usuwa dane wypożyczenie	-
/books/lend/book/<lend_id>	GET	Wyświetlenie informacji o wypożyczeniu książki	-

/books/borrowed/user	GET	Wyświetlenie informacji książkach wypożyczonych przez użytkownika	cookie: sessionToken

Tablica 3.3: Akcje związane z tagami

Tagi /tags			
Endpoint	Request	Opis	Dodatkowo
/tags/	GET	Zmiana wszystkie tagi	-
/tags/	POST	Dodanie tagu	body: title
/tags/	DELETE	Usunięcie tagu	body: title

Tablica 3.4: Akcje związane z wyszukiwaniem

Wyszukiwanie /search			
Endpoint	Request	Opis	Dodatkowo
/search/	GET	Generowanie linku do strony biblioteki	body: title, author, department
/search/library	GET	Pobieranie książek ze strony biblioteki	body: title, author, department

Tablica 3.5: Akcje związane z instytucjami

Institutions /institutions			
Endpoint	Request	Opis	Dodatkowo
/institutions/	GET	Lista instytucji	-
/institutions/<institution_id>	GET	Informacje o instytucji	-

Tablica 3.6: Akcje związane ze znajomymi

Znajomi /friends			
Endpoint	Request	Opis	Dodatkowo
/friends/	GET	Lista znajomych	cookie: sessionToken
/friends/<user_id>	POST	Dodanie znajomego	cookie: sessionToken
/friends/confirmFriendship	POST	Potwierdzenie znajomości	body: friend1, friend2

/friends/changeAuthorizedState/<user_id>	POST	Zmiana pozwolenia na wgląd w książki użytkownika	cookie: sessionToken

3.2 Aplikacja internetowa

3.3 Baza danych

W naszym systemie korzystaliśmy z systemu PostgreSQL - jednego z popularniejszych relacyjnych systemów bazodanowych. Odeszliśmy jednak od klasycznego podejścia projektowania takich baz. Korzystając z JPA (Java Persistence API) - oficjalnego standardu mapowania obiektowo-relacyjnego, umożliwiającego na operowanie obiektami zwanymi encjami, które dzięki tej technologii są automatycznie mapowanie do bazy danych (a więc tabele i relacje między nimi tworzą się automatycznie na podstawie kodu w Java'ie). Żeby z czegoś takiego skorzystać potrzebowaliśmy określić typy encji (POJO - proste obiekty) i relacji między nimi. Poniżej przedstawiono listę takich obiektów, oraz diagram bazy danych jaki otrzymaliśmy w wyniku mapowania (relacje są przestawione względem opisywanej encji).

3.3.1 Encje

Tablica 3.7: Encja: Address

Nazwa encji: Address			
Pole	Typ atrybutu	Typ relacji	Opis
id	Integer	-	klucz główny
country	String	-	-
province	String	-	-
city	String	-	-
street	String	-	-
buildNr	String	-	-
apartNr	String	-	-
code	String	-	-

Tablica 3.8: Encja: Book

Nazwa encji: Book			
Pole	Typ	Typ relacji	Opis
id	Integer	-	klucz główny
title	String	-	-
author	String	-	-

format	Character	-	b - książka fizyczna, e - książka w wersji elektronicznej
path	String	-	ścieżka url do pliku
status	Boolean	-	wypożyczona/niewypożyczona
ownerType	Character	-	typ właściciela (u - user, l - library)
user	User (encja)	wiele do jednego	właściciel książki (null jeżeli właścicielem jest biblioteka)
department	Department (encja)	wiele do jednego	właściciel książki (null jeżeli właścicielem jest osoba fizyczna)

Tablica 3.9: Encja: Borrowed

Nazwa encji: Borrowed			
Pole	Typ	Typ relacji	Opis
id	Integer	-	-
book	Book (encja)	jeden do jednego	książka której dot. wypożyczenie
borrower	User (encja)	jeden do jednego	pożyczający
name	String	-	-
email	String	-	-
message	String	-	-
facebook	String	-	-
dateStart	Date	-	-
dateStop	Date	-	-

Tablica 3.10: Encja: Friend

Nazwa encji: Friend			
Pole	Typ	Typ relacji	Opis
friendId	FriendId	-	klucz złożony
friend1Allow	Boolean	-	pierwsza os. w relacji pozwala na podgląd księgozbioru
friend2Allow	Boolean	-	druga os. w relacji pozwala na podgląd księgozbioru
friendshipConfirmed	Boolean	-	przyjaźń potwierdzona

Tablica 3.11: Klasa pomocnicza - klucz złożony: FriendId

Nazwa encji: FriendId			
Pole	Typ	Typ relacji	Opis
friend1	User (encja)	wiele do jednego	pierwszy z relacji user - user (mniejsze id)
friend2	User (encja)	wiele do jednego	drugi z relacji user - user (większe id)

Tablica 3.12: Encja: Institution

Nazwa encji: Institution

Pole	Typ	Typ relacji	Opis
id	Integer	-	-
name	String	-	-
url	String	-	-
contact	String	-	-
type	Character	-	-

Tablica 3.13: Encja: VerificationToken

Nazwa encji: VerificationToken			
Pole	Typ	Typ relacji	Opis
id	Integer	-	-
token	String	-	-
user	User (encja)	jeden do jednego	użytkownik którego dot. token
expiryDate	Date	-	-

Tablica 3.14: Encja: User

Nazwa encji: User			
Pole	Typ	Typ relacji	Opis
id	Integer	-	-
login	String	-	-
password	String	-	-
email	String	-	-
salt	String	-	'sól' do hasła
name	String	-	-
surname	String	-	-
facebook	String	-	-
isConfirmed	Boolean	-	flaga czy użytkownik jest potwierdzony
Address	address (encja)	jeden do jednego	adres użytkownika

Tablica 3.15: Encja: Tag

Nazwa encji: Tag			
Pole	Typ	Typ relacji	Opis
title	String	-	Nazwa tagu, klucz główny

Tablica 3.16: Encja: TagBook

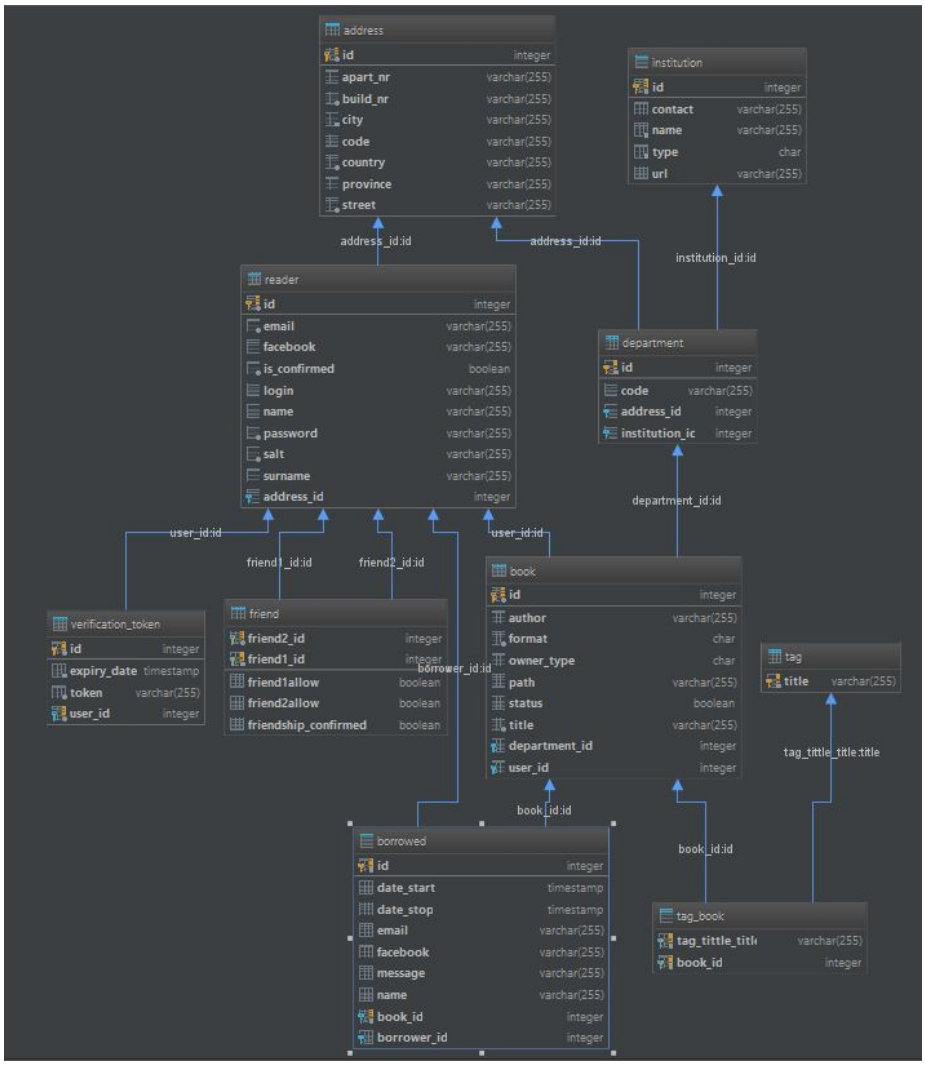
Nazwa encji: TagBook

Pole	Typ	Typ relacji	Opis
tagBook	TagBookId	-	klucz złożony

Tablica 3.17: Klasa pomocnicza - klucz złożony: TagBookId

Nazwa encji: TagBookId			
Pole	Typ	Typ relacji	Opis
tagTittle	Tag (encja)	wiele do jednego	tag dla książki
book	Book (encja)	wiele do jednego	książka jakiej dot. tag

3.3.2 Diagram bazy danych



Rozdział 4

Opis techniczny

4.1 Serwer aplikacji

4.1.1 Podział aplikacji

W naszej aplikacji możemy wyszczególnić trzy podstawowe części - jest to baza danych, serwer aplikacji i serwer WWW (nginx). Rozdzielenie dwóch ostatnich świadczy o tym, że nie jest to klasyczne podejście do aplikacji w architekturze MVC, jednak mimo to jest ona widoczna. W niniejszym podrozdziale skupimy się na podziale aplikacji z perspektywy serwera.

Kontroler

Kontroler jest elementem, który przyjmuje dane od użytkownika, reaguje na jego poczynania. W naszej aplikacji jest to oczywiście kontroler REST'owy. Z serwera WWW wysyłane jest zapytanie na konkretny endpoint, który następnie odpowiada w sposób przez nas przewidziany (więcej o pkt. końcowych w punkcie 3.1.1). Na przykład w przypadku zapytania typu GET na adres <adres-ip>:8080/users uprawniony użytkownik powinien otrzymać informacje o wszystkich użytkownikach w bazie w postaci listy json'ów. Poniżej przedstawiono implementację jednego z kontrolerów (najprostszego z nich).

```
@RestController
@RequestMapping(value = "/api/v1/tags")
public class TagController {

    @Autowired
    public TagService tagService;

    @RequestMapping(method = RequestMethod.GET)
    public Collection<Tag> getTags(){
        return tagService.getTags();
    }
```

```
    @RequestMapping(method = RequestMethod.POST)
    public ResponseEntity<Tag> addTag(@RequestBody
        Tag tag){
        try {
            return
                ResponseEntity.ok(tagService.addTag(tag));
        }
        catch (IllegalArgumentException e) {
            return new
                ResponseEntity<>(HttpStatus.CONFLICT);
        }
    }

    @RequestMapping(method = RequestMethod.DELETE)
    public ResponseEntity<Tag> deleteTag(@RequestBody
        String title) {
        Tag tag =
            tagService.getTagByTitle(title).orElse(null);

        if (tag == null)
        {
            return new
                ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
        try {
            tagService.deleteTag(title);
        } catch (DataAccessException e) {
            return new
                ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
        return new ResponseEntity<>(HttpStatus.OK);

    }
}
```

Adnotacje @RestController i @RequestMapping służą kolejno do oznaczenia, że dana klasa jest kontrolerem (typu REST'owego) oraz ścieżki od jakiej zaczynać się będą adresy wszystkich kontrolerów w tej klasie. @Autowired służy do podpięcia serwisu (jako, że serwis jest obiektem wyjątkowym, jedynym) - Spring Boot szuka tych adnotacji, tworzy i podpina do obiektu oznaczonego tą adnotacją. @RequestMapping użyte nad metodą wskazuje na dalsze rozwinięcie adresu i metodę HTTP jaką można się na ten adres odwoływać.

Serwis

Serwis jest kolejnym poziomem abstrakcji odpowiedzialnym za obliczenia i pośredniczącym (jeżeli zachodzi taka potrzeba) pomiędzy kontrolerem a repozytorium. W warstwie serwisów są np. kodowane i dekodowane hasła, czy też ustalane linki do mapy na podstawie określonego adresu. Poniżej przedstawiono interfejs oraz implementację jednego z serwisów.

```
@Service
public interface TagService {

    Collection<Tag> getTags();

    Optional<Tag> getTagByTitle(String title);

    Tag addTag(Tag tag);

    void deleteTag(String title);

}
```

Adnotacja @Service wskazuje, że powyższy kod dotyczy serwisu.

```
@Component
public class TagServiceImpl implements TagService{

    @Autowired
    private TagRepository tagRepository;

    @Override
    public Collection<Tag> getTags() {
        return
           StreamSupport.stream(tagRepository.findAll().spliterator(),
           false).
                collect(Collectors.toList());
    }

    @Override
    public Optional<Tag> getTagByTitle(String title) {
        return
           Optional.ofNullable(tagRepository.findOne(title));
    }

    @Override
    public Tag addTag(Tag tag)throws
       IllegalArgumentException {
        try {
            return tagRepository.save(tag);
```

```
        }
        catch (DataAccessException e) {
            throw new IllegalArgumentException(e);
        }
    }

    @Override
    public void deleteTag(String title) throws
        IllegalArgumentException {
        if (title == null)
            throw new
                IllegalArgumentException("Cannot
                delete tag with unspecified title");

        if (!tagRepository.exists(title))
            throw new
                IllegalArgumentException("Cannot
                delete tag that does not exist");

        try {
            tagRepository.delete(title);
        }
        catch (DataAccessException e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```

Powyżej widzimy implementację wcześniej opisanego interfejsu. @Component sprawia, że spring automatycznie znajdzie tą klasę i stworzy niezbędny obiekt. W powyższym przykładzie widzimy odwołania do repozytorium (tagRepository), o którym w następnym punkcie.

Repozytorium

Repozytorium jest pewnym typem klasy, który zapewnia komunikację z bazą danych. W naszej aplikacji korzystaliśmy z repozytorium CRUDowego (CRUD - create, read, update and delete - podstawowa metoda komunikacji z bazą danych) rozszerzonego o hibernate - framework'a ułatwiającego realizację dostępu do bazy danych w szczególnych przypadkach (hibernate pozwala na pisanie własnych, bardziej skomplikowanych zapytań). Uznaliśmy, że standard JPA jest nam nie potrzebny. Poniżej przedstawiono jedną z klas typu repozytorium - każda taka klasa pozwala na dostęp do jednego z typów obiektów (więcej o typach obiektów i o modelach w rozdziale poświęconym bazie danych).

```
public interface UserRepository extends
    CrudRepository<User, Integer> {
```

```
    User findByLogin(String login);

    User findByEmail(String email);

    @Query("SELECT u" +
            " FROM User u" +
            " WHERE UPPER(u.name) like
               UPPER(CONCAT('%',:any,'%')) " +
            " OR UPPER(u.surname) like
               UPPER(CONCAT('%',:any,'%')) " +
            " OR UPPER(u.login) like
               UPPER(CONCAT('%',:any,'%'))" +
            " OR UPPER(u.email) like
               UPPER(CONCAT('%',:any,'%'))")
    Collection<User> findByAny(@Param("any") String
       any);
}
```

Implementacja repozytorium jest bardzo prosta, a w najlepszym przypadku nie wymaga pisania ani jednej metody. CrudRepository posiada kilka wbudowanych zapytań (takich jak findOne(), czy delete()), które działają na podstawie klucza głównego obiektu (więcej w 3.3 i 4.3), jednak pozwala ono również na pisanie własnych zapytań poprzez odpowiednie nazywanie metod - np. findByLogin. Więcej informacji na ten temat można znaleźć pod następującym adresem http://docs.spring.io/spring-data/jpa/docs/1.5.1.RELEASE/reference/html/jpa.repositories.html#jpa.query-methods.query-creation. W przypadku bardziej skomplkowanych zapytań korzystaliśmy z adnotacji @Query (dostarczonej przez hibernate), która pozwala na pisanie zapytań w pseudosqlowym języku (HQL). Więcej informacji na stronie https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html.

4.1.2 System kont użytkowników

Logowanie i sesja

Dane wpisane przez użytkownika w celu zalogowania się są wysyłane na odpowiedni endpoint. W celu ich weryfikacji stworzyliśmy specjalny serwis odpowiedzialny za autoryzację:

```
@Component
public class AuthorizationServiceImpl implements
    AuthorizationService {

    private final Map<UUID, Session> sessions = new
       HashMap<>();
```

```
    @Autowired
    public HashingService hashingService;

    @Autowired
    private UserService userService;

    @Override
    public Optional<Session> getSession(UUID token) {
        return Optional.of(sessions.compute
                        (token, (t, s) ->
                           Optional.ofNullable(s).
                    filter(es ->
                       es.getExpiration().isAfter(Instant.now())).
                    orElse(null)));
    }

    @Override
    public boolean invalidateSession(UUID token) {
        return sessions.remove(token) != null;
    }

    @Override
    public Optional<Session> authenticate(Credentials
       credentials) {
        val session =
           userService.getByLogin(credentials.getLogin()).
                filter(u ->
                   u.getPassword().equals(Base64.getEncoder().
                        encodeToString(hashingService.hash(credentials.getPa
                        toCharArray(),
                           Base64.getDecoder().decode(u.getSalt())))))).
                map(u -> new Session(u));

        session.ifPresent(s ->
           sessions.put(s.getToken(), s));

        return session;
    }
}
```

Jak widać w powyższym fragmencie kodu, do tego serwisu został podpięty serwis odpowiedzialny za haszowanie - hasło jest przez niego kodowane (na podstawie 'soli' przypisanej do użytkownika) i porównywane z tym co znajduje się w bazie. Taka implementacja spełnia podstawowe wymagania dot. bezpieczeństwa. Kolejnym elementem na który warto zwrócić uwagę jest tablica asocjacyjna zawierająca pary wartości tokenu i sesji. W punkcie końcowym od-

powiedzialnym za logowanie za każdym razem tworzona jest taka para, z której klucz wysyłany jest w ciasteczku do użytkownika. Pozwala to na utrzymywanie sesji i weryfikację użytkownika w innych punktach końcowych.

System pocztowy

Kiedy użytkownik tworzy konto, jego dane pojawiają się w bazie danych (po uprzednim wygenerowaniu soli i zakodowaniu hasła). Jednak jest to użytkownik niepotwierdzony przez system (co oznaczamy za pomocą flagi isConfirmed). W celu potwierdzenia konta na adres użytkownika wysyłany jest email. Dzieje się to dzięki klasie EmailSender.

```
@Component
public class EmailSender {


    @Autowired
    private JavaMailSender javaMailSender;

    public EmailStatus sendPlainText(String to,
       String subject, String text) {
        return sendM(to, subject, text, false);
    }

    public EmailStatus sendHtml(String to, String
       subject, String htmlBody) {
        return sendM(to, subject, htmlBody, true);
    }

    private EmailStatus sendM(String to, String
       subject, String text, Boolean isHtml) {
        try {
            MimeMessage mail =
               javaMailSender.createMimeMessage();
            MimeMessageHelper helper = new
               MimeMessageHelper(mail, true);
            helper.setTo(to);
            helper.setSubject(subject);
            helper.setText(text, isHtml);
            javaMailSender.send(mail);

            return new EmailStatus(to, subject,
               text).success();
        } catch (Exception e) {
            return new EmailStatus(to, subject,
               text).error(e.getMessage());
        }
```

```
    }
}
```

Jak widać w powyższym fragmencie kodu klasa ta korzysta z kilku podstawowych klas wykorzystywanych do obsługi poczty (MimeMessage, MimeMessageHelper) jak i z gotowego serwisu javaMailSender, którego konfiguracje można znaleźć w pliku application.properties.

```
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=teambooka
spring.mail.password=TeamBookaTeamBooka
spring.mail.protocol=smtp
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.default-encoding=UTF-8
```

Maile, które wysyłamy zawierają w sobie link odwołujący się do specjalnego endpointa, który w pasku zapytań zawiera specjalny token identyfikujący użytkownika. Tokeny są przechowywane w bazie danych. Poniżej przedstawiono klasę typu encji (więcej w rozdziale dot. bazy danych) dla tokenów.

```
@Entity
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class VerificationToken {
    public static final int EXPIRATION = 60 * 24;

    @Id
    @GeneratedValue(strategy =
       GenerationType.IDENTITY)
    private Integer id;

    @Column(nullable = false)
    private String token;

    @OneToOne(optional = false)
    private User user;

    @Column(nullable = false)
    private Date expiryDate;

    public Date calculateExpiryDate(int
       expiryTimeInMinutes) {
        Calendar cal = Calendar.getInstance();
        cal.setTime(new
           Timestamp(cal.getTime().getTime()));
```

```
        cal.add(Calendar.MINUTE, expiryTimeInMinutes);
        return new Date(cal.getTime().getTime());
    }

}
```

Żeby móc wysłać użytkownikowi hiper-link potrzebowaliśmy jeszcze czegoś co pozwalałoby nam modyfikować kod HTML'a na podstawie zmiennych z aplikacji. W tym celu użyliśmy silnika thymeleaf. Poniżej przedstawiono przygotowanego kod wykorzystujący wspomniany silnik i template'a

```
@Component
public class EmailHtmlSender {

    @Autowired
    private EmailSender emailSender;

    @Autowired
    private TemplateEngine templateEngine;

    public EmailStatus send(String to, String
       subject, String templateName, Context context)
       {
        String body =
           templateEngine.process(templateName,
           context);
        return emailSender.sendHtml(to, subject,
           body);
    }
}
```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title th:remove="all">Confirmation</title>
    <meta http-equiv="Content-Type"
       content="text/html; charset=UTF-8"/>
</head>
<body>
Your account was successfully created. Please click
   the link below<br>
to confirm your email address and activate your
   account:
<p>
    <a th:text=${link} th:href="@{'http://' +
       ${link}}">"${link}"</a>
</p>
```

```
<p>
    --<br>
    Questions? Comments? We don't care
</p>
</body>
</html>
```

4.1.3 Przeszukiwanie zasobów bibliotecznych

Formułowanie zapytań do zasobów bibliotecznych

Konstrukcja podstawowych zapytań do zasobów bibliotecznych jest stosunkowo prosta. Używany jest do tego język poleceń CCL (Common Command Language), umieszczany w linku wraz z komendą wyszukania książek. Przykładowo:

$$((wau = George\ or\ Martin) and (wti = Pieśń?))$$

Pola dopuszczalne w zapytaniu

Poniżej lista możliwych zmiennych poprzez które możemy się odwoływać bezpośrednio do bazy danych.

- TIT - Titles - Tytuły
- AUT - Authors - Autorzy
- IMP - Imprint - Wydano
- SUB - Subjects - Hasła przedmiotowe
- SRS - Series - Serie
- LOC - Location - Sygnatura

Wykorzystanie powyższych pól wymaga większej precyzji w umieszczanych tam danych. Lepszym rozwiązaniem wydają się zmienne typu słowa, odpowiadają one zawartości pól uzupełnianych ręcznie przez użytkownika na stronie co sugeruje mniej rygorystyczne zasady ich uzupełniania.

- WRD - Words - Słowa
- WTI - Words in title field - Słowa z pola tytułu
- WAU - Words in author field - Słowa z pola autora
- WPE - Words in personal author field - Słowa z pola autora indywidualnego
- WCO - Words in corporate author field - Słowa z pola autora korporatywnego

- WME - Words in meeting field - Słowa z pola imprez
- WUT - Words in uniform title field - Słowa z pola tytułu ujednoliconego
- WPL - Words in place field - Słowa z pola miejsca wydania
- WPU - Words in publisher field - Słowa z pola wydawcy
- WSU - Words in subject field - Słowa z pola haseł przedmiotowych (odpowiednik tagów)
- WSM - Words in MeSH subjects - Słowa z haseł MeSH
- WST - Words in status - Słowa w statusie
- WGA - Words in geographical area - Słowa z haseł geograficznych
- WYR - Year of publication - Rok publikacji

Operatory logiczne i formatowanie zapytań

Jak widać zapytanie posiada operatory logiczne do składania jego treści. Każde z nich może być zastąpione odpowiednikiem podanym poniżej.

- AND = +/&
- OR = |
- NOT = ~

Oprócz powyższych podstawowych zasad tworzenia zapytań są jeszcze dodatkowe:

Znak ? lub * (gwiazdka), mogą być stosowane w słowie, ale nie mogą występować częściej niż raz w zapytaniu.

Symbol # może służyć do wyszukiwania różnych wersji pisowni, gdy jedna wersja słowa posiada jeden znak więcej niż druga. Na przykład, colo#r spowoduje wyszukanie zarówno słowa color jak i colour; a arch#eology spowoduje wyszukanie zarówno słowa archaeology jak i archeology.

Znak ! może służyć do wyszukiwania różnych wersji pisowni, w przypadkach gdy pisownia może różnić się jednym znakiem. Na przykład, wom!n spowoduje wyszukanie zarówno słowa woman jak i women.

Znak ! poprzedzający numer, można umieszczać pomiędzy słowami, aby określić maksymalny odstęp pomiędzy słowami, z zachowaniem kolejności występowania słów. W tym przypadku, ballads !3 england spowoduje wyszukanie Ballads of England i Ballads of Merry Olde England ale nie England and Her Ballads.

Symbol % poprzedzający numer, można umieszczać pomiędzy słowami, aby określić maksymalny odstęp pomiędzy słowami, niezależnie od ich kolejności. na przykład, england %3 ballads spowoduje wyszukiwanie Ballads of England , Ballads of Merry Olde England i England and Her Ballads.

Pomiędzy słowami mogą być umieszczane symbole -> (myślnik i znak większości). Oznacza to, że chcemy wyszukać rekordy, które zawierają słowa od pierwszego słowa (włącznie) aż do drugiego. To wyszukiwanie jest szczególnie przydatne do ograniczenia zestawu rekordów ze względu na rok wydania. Np: 1993 -> 1996.

Przeszukiwanie konkretnych bibliotek

Możliwy jest też wybór biblioteki w której szukamy książki, polega to na dodaniu na końcu zapytania formuły $\&local\_base = \underline{nr}$, gdzie $\underline{nr}$ oznacza numer szukanej biblioteki w systemie. Poniżej tabela oddziałów i ich numery systemowe.

Tablica 4.1: Oddziały biblioteki miejskiej

Numer Filii	Lokacja
01	ul. Sztabowa 98 (ALEPH)
02	ul. Eluarda 51-55 (ALEPH)
03	ul. Morelowskiego 43 (ALEPH)
04	ul. Wieczysta 105 (ALEPH)
05	ul. Namysłowska 8 - Grafit (ALEPH)
06	ul. Lwowska 21 (ALEPH)
07	ul. Suwalska 3 (ALEPH)
08	ul. Głogowska 9-11
09	ul. Wielkopolska 16
10	ul. Jelenia 7 (ALEPH)
11	ul. Krzywoustego 286 - Fama (ALEPH)
15	ul. Rękodzielnicza 1
16	ul. Kolista 17 (ALEPH)
17	ul. Gałczyńskiego 8 (ALEPH)
18	ul. Grabiszyńska 236a (ALEPH)
20	ul. Majakowskiego 34/1a
22	ul. Chociebuska 8-10 (ALEPH)
23	bul. Ikara 29-31 (ALEPH)
26	ul. Olszewskiego 85a (ALEPH)
27	ul. Łokietka 13 (ALEPH)
29	ul. Reja 1-3 (ALEPH)
31	ul. Strachocińska 88
32	ul. 9 Maja 26
37	ul. Sempołowskiej 54a (ALEPH)
40	ul. Przybyszewskiego 59 (ALEPH)
41	ul. Sołtysowicka 48
42	ul. Serbska 5a (ALEPH)
44	ul. Powstańców Śl. 210 (ALEPH)
45	ul. Traugutta 82 (ALEPH)

46	ul. Tarnogórska 1 (ALEPH)
47	ul. Polna 4 (ALEPH)
53	ul. Hercena 13
54	ul. Jesienna 22 (ALEPH)
55	ul. Zielińskiego 2 (ALEPH)
57	ul. Szewska 78 (ALEPH)
58	pl. Teatralny 5 - Mediateka (ALEPH)
64	ul. Osobowicka 127
69	ul. Wróblewskiego 1-5 (ZOO)
70	ul. Wittiga 10 (Biblioteka Parafialna)

Filtrowanie otrzymywanych wyników

Możliwe sa również opcje filtrowania ustawiane już w zapytaniu do serwera. Zapytania takie są kolejnym członem podstawowego zapytnia CCL. Poniżej możliwe elementy takiego zapytania oraz dopuszczalne parametry.

- WFT - szukany format
  - BK - książki
  - SE - czasopismo
  - MU - aubiobook
  - MP - mapa
  - CF - ebook
  - VM - film
  - PD - pomoc dydaktyczna
- WLN - wybór języka poszukiwanego dzieła
  - POL - polski
  - ENG - angielski
  - FRE - francuski
  - GER - niemiecki

Zaimplementowane mechanizmy formułowania zapytań

Najprostszym sposobem jest utworzenie zapytania składającego się jedynie z wypełnionego pola WRD, gdzie można umieścić wszystkie możliwe elementy zapytania, jednak zrezygnowaliśmy z tego na rzecz bardziej szczegółowego zapytania składanego z pół odpowiednich danych, kolejno WAU jako pole z szukanym autorem, WTI jako pole z tytułem książki oraz numer systemowy oddziału biblioteki. Oczywiście jeśli któryś z wymienionych elementów jest nieobecny w zapytaniu, przychodzącym do serwera, zostaje on również pominięty przy tworzeniu zapytania.

Poniżej znajduje się przykład całego złożonego zapytania z podkreślonym edytowanym przez nas fragmentem.

$$https://katalog.biblioteka.wroc.pl/F?func=find-c\&ccl\_term=((WAU=GeorgeORMartin)AND(WTI=Pieśń?))and(WFT=(BK))\&local\_base=31$$

Pobieranie danych z systemu bibliotecznego

Z powodu braku możliwości uzyskania bezpośredniego dostępu do bazy danych biblioteki publicznej w Wrocławiu, dane musieliśmy uzyskiwać w inny sposób. Wykorzystaliśmy powyższe metodykę zapytań oraz bibliotekę Jsoup udało się nam tego dokonać. Należy tutaj wytłumaczyć iż Jsoup posiada API do wydobywania i manipulowania danymi uzyskiwanymi z plików HTML.

Przy każdym zapytaniu na nasz serwer, tworzymy zapytanie do sytemu bibliotecznego zgodnie z wcześniej opisaną formuła. Następnie schodzimy pod uzyskane adresu z poziomu serwera, który również otrzymuje zwrotny HTML ze stroną zawierającą odpowiedź biblioteki na wysłane zapytanie. Tutaj otrzymana strona jest parsowana w odpowiedni sposób, a uzyskane dane dostosowywane do modelu książki opisanego w naszym systemie i zwracane do strony z której przyszło zapytanie.

4.2 Aplikacja internetowa

4.2.1 Kolejka komunikatów

4.2.2 Google Drive

4.3 Baza danych

Jak znaczna część konfiguracji tak i zmienne potrzebne do zalogowania się do bazy danych znajdują się w pliku ąpplication.properties":

- spring.datasource.url - zmienna przyjmująca adres bazy danych. Domyślną wartość stanowi "jdbc:postgresql://localhost:5432/BookaDB"
- spring.datasource.username - nazwa użytkownika bazy danych. Domyślnie jest to "postgres"
- spring.datasource.password - hasło do bazy danych. Domyślnie jest to "postgres"

Każdy obiekt typu encji (więcej w poprzednim rozdziale) został stworzony wg. powtarzalnego schematu, który zostanie objaśniony na podstawie poniższego fragmentu kodu.

```
@Entity
@Table(name="Reader")
@Data
```

```
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class User implements Serializable {

    @Id
    @GeneratedValue(strategy =
       GenerationType.IDENTITY)
    private Integer id;

    private String login;

    @Column(nullable = false)
    private String password;

    @Column(nullable = false)
    private String email;

    @Column(nullable = false)
    private String salt;

    @Column(nullable = false)
    private Boolean isConfirmed;

    @OneToOne
    private Address address;
    private String name;
    private String surname;
    private String facebook;
}
```

Każdy klasa implementująca obiekty typu encji musi zaczynać się od adnotacji "@Entity". Adnotacja @Table daje dostęp do kontroli tego, w jaki sposób zostanie stworzona tabela reprezentująca daną klasę. Na przykładzie widać w jaki sposób można nadać nazwę tabeli (domyślna wartość jest równa nazwie klasy). Cztery kolejne adnotacje pochodzą z narzędzia łombok"pozwalającego na automatyczne wygenerowanie standardowych metod (takich jak np gettery i settery). Przejdźmy do ciała klasy - każde pole reprezentuje kolejną kolumnę, a adnotacje nad polami pozwalają sterować tym w jaki sposób dane pole zostanie odwzorowane w bazie danych. Adnotacją "@Idóznacza się klucz główny, @GeneratedValue mówi o tym, że wartość powinna być wygenerowana automatycznie, @Column pozwala na sterowanie parametrami pól, @OneToOne wskazuje na typ relacji pomiędzy dwoma obiektami (adnotacja musi znajdować się nad zdefiniowanym przez nas obiektem). Istnieje wiele innych, ciekawych możliwości pozwalających budować modele obiektów, jednak wymienienie ich wszystkich niej jest celem tej dokumentacji. Należy jednak wspomnieć o jeszcze jednym

ważnym elemencie jakim jest adnotacja @Embededkey. Pozwala ona na tworzenie kluczy złożonych i stosowana jest analogicznie do @Id z tym że nad specjalnie stworzonym przez nas polu typu id. Poniżej przedstawiono przykład.

```
@Entity
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Friend implements Serializable{

    @EmbeddedId
    private FriendId friendId;

    private Boolean friend1Allow;
    private Boolean friend2Allow;

    private Boolean friendshipConfirmed;

}

@Embeddable
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class FriendId implements Serializable{

    @ManyToOne(optional = false)
    @OnDelete(action = OnDeleteAction.CASCADE)
    private User friend1;

    @ManyToOne(optional = false)
    @OnDelete(action = OnDeleteAction.CASCADE)
    private User friend2;
}
```

Rozdział 5

Instalacja i konfiguracja systemu

Instalację powinna przeprowadzać osoba posiadająca podstawową wiedzę z zakresu systemów baz danych, serwerów HTTP i aplikacji webowych.

- Pobieranie niezbędnych plików

  1. Baza danych

     Aby aplikacja działała poprawnie należy przede wszystkim zainstalować bazę danych - *Postgresql-9.6*

     https://www.postgresql.org/download/

     Do obsługi bazy danych można użyć dowolnego narzędzia, które to umożliwia, jednak zaleca się korzystanie z programu *pgAdmin*

     https://www.pgadmin.org/download/

  2. Aplikacja webowa

     Aby aplikacja webowa wyglądała tak jak wyglądać powinna należy pobrać niezbędne zależności. Posłuży nam do tego manager pakietów *Npm*. Można go pobrać stąd:

     https://www.npmjs.com/

     Po pobraniu *Npm'a* należy otworzyć konsolę systemu i wprowadzić następującą komendę:

     npm install -g bower

     Następnie należy wejść do głównego katalogu plików źródłowych (tam gdzie znajduje się plik *bower.json* i wykonać komendę *bower install*

- Konfiguracja środowiska

  Wszystkie parametry którymi będziemy sterować znajdują się w pliku */src/main/resources/application.properties*, jednak domyślne ustawienia powinny być wystarczające. Pierwsze trzy linijki określają adres bazy da-

nych, użytkownika bazy danych i jego hasło. Linijka *spring.jpa.hibernate.ddl-auto=create-drop* oznacza, że baza danych będzie automatycznie czyszczona ilekroć ktoś zrestartuje aplikację (jeżeli takie zachowanie jest niepożądane polecamy zmienić wartość na *update*).

Trzy kolejne parametry odnoszą się do adresu serwera. Klucz *server.address* i *registration.address* powinny posiadać tę samą wartość. Tutaj należy zaznaczyć, że jeżeli wartość *serwer.address* nie będzie ustawiona no localhost to usługi Google nie będą działały (istnieje też spore ryzyko, że nieużywane przestaną działać samoistnie), gdyż taka wartość została ustalona przy konfigurowaniu połączenia z chmurą (po stronie Google).

Dwa kolejne bloki nie powinny być konfigurowane - pierwszy z nich odpowiada za konfigurację serwisu pocztowego, a drugi za narzędzie pozwalające modyfikować kod HTML z poziomu języka JAVA.

- Budowanie i uruchomienie środowiska.

  Aby zbudować środowisko należy mieć zainstalowane narzędzie *Maven*. W głównym katalogu wywołujemy polecenie *mvn clean package*, a następnie uruchamiamy środowisko komendą *java -jar <targer>*, gdzie jako <target> podajemy nazwę zbudowanego pliku *.jar*. Jeżeli zachodzi potrzeba wypełnienia bazy przykładowymi danymi (tutaj trzeba zauważyć, że wyszukiwanie w bibliotekach nie będzie działać, jeżeli nie zostaną uzupełnione tabele *Institution* i *Department*) należy uruchomić skrypt uzupełniający bazę (*script.sql*), który można znaleźć w głównym katalogu aplikacji. Następnie, jeżeli chcemy mieć możliwość zalogowania się na jednego z istniejących użytkowników (hasło użytkownika jest takie same jak jego login), należy zakodować ich hasła. W tym celu powstał adres, na który należy wysłać następujące zapytanie typu POST - *localhost:8080/api/v1//users/hash*.

Rozdział 6

Spis tabel i obrazów

Spis rysunków

Spis tablic