

1. Projekt systemu

1.1 Analiza wymagań

Stworzona na potrzeby projektu aplikacja webowa została zintegrowana z bazą danych, aplikacja zostanie następnie skonteneryzowana, a jej kod umieszczony w repozytorium github. Aplikacja webowa jest prostym systemem „bibliotecznym” który pozwala na wybranie ulubionych książek w dostępnej bazie i zarządzanie nimi. Użytkownicy mają również możliwość stworzenia konta. Aplikacja w samym zamyśle ma prezentować jedynie podstawowe funkcje oferowane przez strony internetowe, jednak główny nacisk został nałożony na konteneryzację omawianego projektu.

W interakcje z aplikacją webową możemy wejść z trzech perspektyw, różnych użytkowników:

- Niezalogowany użytkownik – jest to podstawowy rodzaj interakcji z aplikacją webową, pierwsze wejście na stronę zawsze będzie wiązało się z tym rodzajem interakcji z nią. Ten rodzaj użytkownika będzie posiadał jedynie dwie możliwości, zalogowania się oraz w przypadku jeśli nie posiada jeszcze konta w aplikacji, zarejestrowania się.
- Zalogowany użytkownik – jest to rodzaj konta które może stworzyć każdy kto ma dostęp do aplikacji webowej. Po zalogowaniu się użytkownik będzie miał możliwość przeglądania książek, które są dostępne w aplikacji, a następnie wybrać swoje ulubione książki. Kiedy użytkownik już doda książki do „ulubionych” będzie w stanie nimi zarządzać w kolejnej zakładce aplikacji webowej „Favourites”. W tej zakładce będzie mógł przeglądać dodane książki oraz usuwać je z „ulubionych”.
- Admin – ten rodzaj konta jest ograniczony pod względem tworzenia kolejnych jego odpowiedników. Jedynie już stworzony administrator będzie w stanie dodać kolejne konto admina. Admin będzie posiadał dostęp do wszystkich funkcjonalności które posiada zwykły zalogowany użytkownik oraz do specjalnego panelu o nazwie „CRUD”. Panel ten pozwoli mu na dodawanie nowych wpisów do istniejących tabel w bazach danych, aktualizacje, odczytywanie oraz usuwanie ich. Jest to konto najbardziej uprzywilejowane.

1.1.1. Cechy funkcjonalne systemu

W celu spełnienia wcześniej opracowanych wymagań została opracowana lista dostępnych funkcjonalności systemu:

- Rejestracja użytkownika,
- Logowanie,
- Przeglądanie dostępnych książek z bazy danych po zalogowaniu się,
- Dodawanie książek do swoich „ulubionych” poprzez przycisk dostępny dla zalogowanych użytkowników,
- Usuwanie książek z „ulubionych” poprzez przycisk dostępny dla zalogowanych użytkowników,
- Dodawanie wpisów do dowolnej tabeli bazy danych poprzez konto Admin,
- Usuwanie wpisów z dowolnej tabeli bazy danych poprzez konto Admin,
- Aktualizowanie wpisów dowolnej tabeli bazy danych poprzez konto Admin,
- Odczytywanie wpisów dowolnej tabeli bazy danych poprzez konto Admin

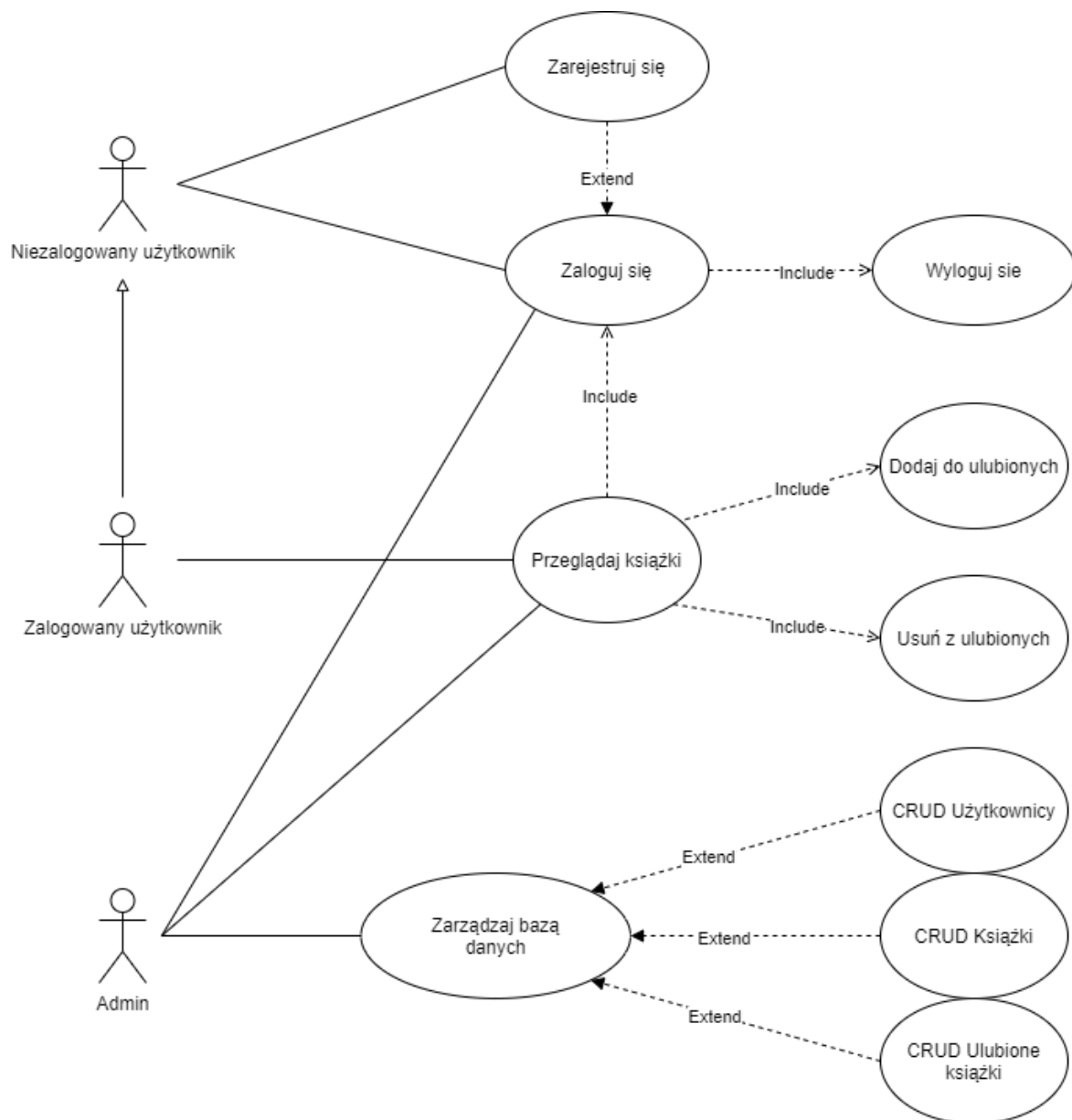
1.1.2. Cechy niefunkcjonalne systemu:

Opracowane zostały następujące cechy niefunkcjonalne systemu:

- Intuicyjny interfejs, który jest prosty w obsłudze.
- Podział na podstrony umożliwiające łatwą nawigację,
- Rozróżnianie zwykłego użytkownika od administratora,

1.1.3 Diagram przypadków użycia

W celu lepszego zobrazowania zamodelowanych funkcjonalności systemu stworzony został również diagram przypadków użycia. Dzięki jego wykorzystaniu możemy w prostszy sposób zorientować się w jaki sposób poszczególne rodzaje użytkowników mogą wejść w interakcje z aplikacją webową.

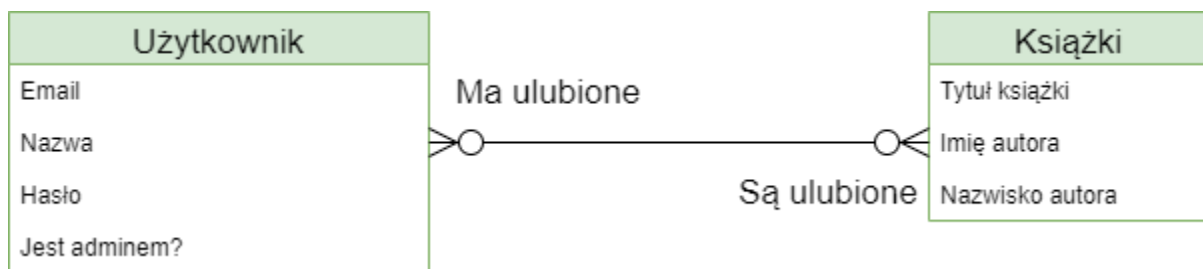


Rysunek 1 Diagram przypadków użycia

1.2 Projekt bazy danych

1.2.1 Model konceptualny

Po poznaniu funkcjonalności naszego systemu możemy przystąpić do modelowania bazy danych. Pierwszym modelem, który pozwala rozeznąć się w tym jakie tabele oraz atrybuty powinny znajdować się w bazie danych tworzonego systemu jest model konceptualny. Poniżej możemy zaobserwować model konceptualny stworzony na potrzeby tego systemu.



Rysunek 2 Model konceptualny bazy danych

Jak widać znajdują się tutaj kluczowe informacje pozwalające nam na rozpoczęcie procesu implementacji bazy danych. W przypadku tabeli użytkownik widzimy tutaj atrybuty takie jak:

- Email
- Nazwa
- Hasło
- Jest adminem?

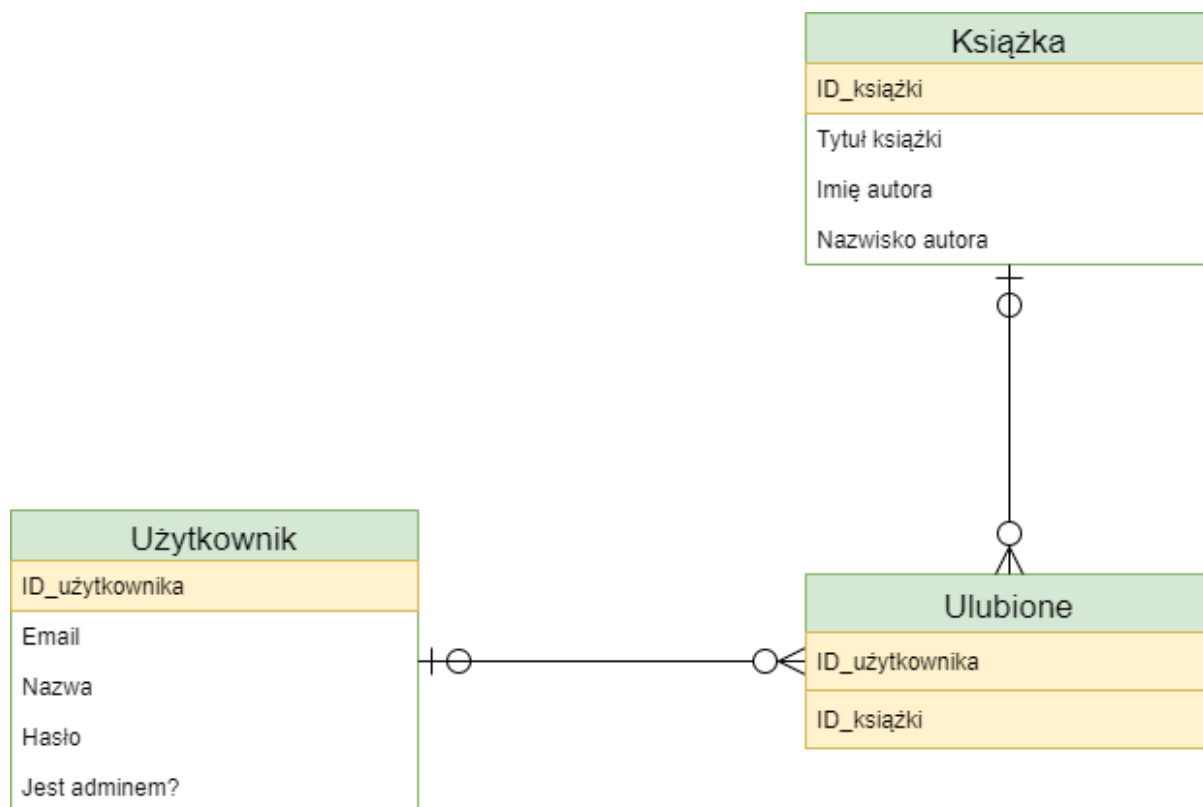
Zaś w przypadku tabeli książki widzimy podstawowe informacje takie jak:

- Tytuł książki
- Imię autora
- Nazwisko autora

Widoczna jest również relacja między obiema tabelami. Jeden użytkownik może mieć wiele ulubionych książek lub żadną, zaś jedna książka może być ulubioną książką wielu użytkowników lub żadnego. Mamy tutaj relacje wiele do wielu. Przy tworzeniu kolejnych modeli należy uwzględnić ten fakt i stworzyć tabelę przejściową.

1.2.2 Model logiczny bazy danych

Stworzenie modelu logicznego bazy danych pozwoliło nam na wybranie kluczy głównych poszczególnych tabeli oraz utworzenie tabeli przejściowej. Model logiczny możemy zobaczyć kolejnej stronie.

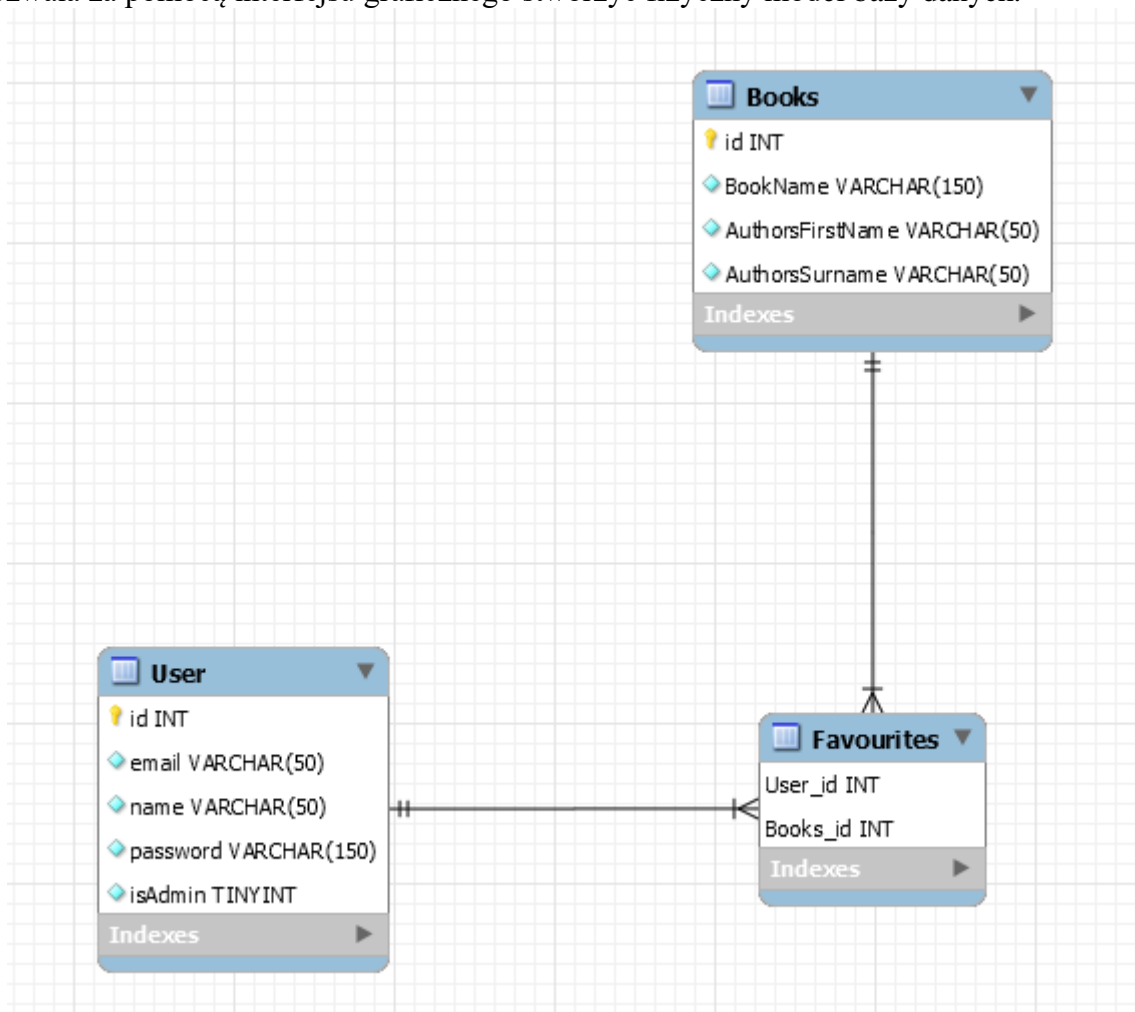


Rysunek 3 Model logiczny bazy danych

Jak widać powyżej kluczem głównym dla tabeli „Użytkownik” zostało jego ID i analogicznie taka sytuacja miała miejsce w przypadku tabeli „Książka”. Świeżo zamodelowana tabela przejściowa „Ulubione” posiadała klucz główny składający się z dwóch atrybutów ID użytkownika oraz ID książki. Jeden użytkownik nie może wiele razy dodać do „ulubionych” tej samej książki, tak więc takie rozwiązanie, gdzie klucz główny składa się z już obecnych w bazie danych informacji, pozwoliłoby na zaoszczędzenie pamięci dyskowej w przypadku bardziej komercyjnego rozwiązania. Tabela przejściowa pozwoliła na rozbicie jednej relacji wiele do wielu na dwie relacje jeden do wielu.

1.2.3 Model fizyczny bazy danych

Dzięki stworzeniu modelu logicznego jesteśmy w stanie w prostszy sposób fizycznie zamodelować bazę danych. MySQL udostępnia proste narzędzie MySQL Workbench, które pozwala za pomocą interfejsu graficznego stworzyć fizyczny model bazy danych.



Rysunek 4 Fizyczny model jednej bazy danych

Jak widać powyżej w przypadku tworzenia fizycznego modelu należy sprecyzować jaki rodzaj danych będziemy przechowywać w bazie danych. W przypadku kluczy głównych tabel które nazywane są ID jest to typ INT, który pozwala na używanie liczb całkowitych. W przypadku atrybutów, które mają przechowywać tekst wykorzystany został typ VARCHAR zaś w nawiasach zostało sprecyzowane jak długie mogą być przechowywane teksty. Atrybut `isAdmin` tabeli `User` wykorzystuje typ `TINYINT` który w bazach danych MySQL jest wykorzystywany jako zmienna Boolean (*True/False*).

Poniżej możemy zobaczyć jakie ograniczenia zostały wybrane dla poszczególnych atrybutów:

User:

- `id` INT – Primary key, Not Null, Unique, Auto Incremental
- `email` VARCHAR(50) – Not Null, Unique
- `name` VARCHAR(50) – Not Null
- `password` VARCHAR(50) – Not Null

- isAdmin TINYINT – Not Null

Books:

- id INT – Primary key, Not Null, Unique, Auto Incremental
- BookName VARCHAR(150) – Not Null
- AuthorsFirstName VARCHAR(50) – Not Null
- AuthorsSurname VARCHAR(50) – Not Null

Favourites:

- User_id INT – Primary key, Not Null
- Books_id INT – Primary key, Not Null

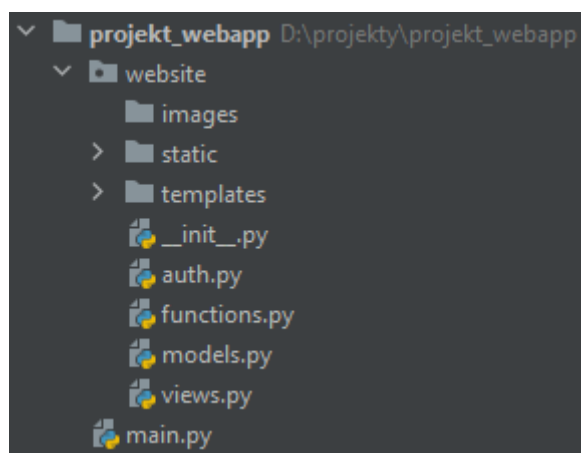
Ten model bazy danych został wykorzystany przy tworzeniu bazy danych połączonej z projektowaną aplikacją webową.

2. Implementacja projektu

W tej części dokumentacji zostanie omówiona implementacja tworzonego na potrzeby projektu systemu. Niemal każdą aplikację webową można podzielić na dwie główne części, stronę serwerową oraz stronę kliencką. Strona serwerowa, czyli tak zwany *backend*, odpowiada za logikę aplikacji oraz komunikację z bazą danych, zaś *frontend* jest częścią dzięki której klient może wejść w interakcję z aplikacją.

2.1 Backend aplikacji webowej

Stworzony kod aplikacji webowej jest podzielony między różnymi plikami oraz katalogami. Każda z nazw plików w spójny sposób opisuje za który element aplikacji odpowiada. Poniżej możemy zobaczyć jak wygląda struktura kodu.



Rysunek 5 Struktura *backendu*

Plik `main.py` odpowiada w głównej mierze za uruchomienie aplikacji. Kod został napisany w języku python przy wykorzystaniu framework'a Flask. Dzięki takiemu rozwiązaniu uruchamianie aplikacji na lokalnej maszynie było banalnie proste co można zobaczyć na rysunku 6.

```
app = create_app()

if __name__ == '__main__':
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain('D:\projekty\cert.pem', 'D:\projekty\key.pem')
    app.run(host='0.0.0.0', port='443', ssl_context=context)
```

Rysunek 6 Plik `main.py`

Plik `__init__.py` w folderze `website` miał za zadanie, tak jak nazwa wskazuje, inicjalizację działania strony przed jej uruchomieniem. W tym pliku konfigurowane były takie informacje jak klucz tajny wykorzystywany przez *framework* Flask do plików Cookies, połączenia z bazami danych oraz inicjalizacja modułu logowania i podstron. Poniżej możemy zaobserwować jak wyglądał ten plik.

```

def create_app():
    app.config['SECRET_KEY'] = 'aeh projekt'
    app.config['SQLALCHEMY_DATABASE_URI'] = "mysql://admin2:adminpass2@localhost/library3"
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
    app.config['SQLALCHEMY_BINDS'] = {
        'user': "mysql://user2:user2@localhost/library3",
        'login': "mysql://login2:loginpass2@localhost/library3"
    }

    db.init_app(app)

    from .views import views
    from .auth import auth

    app.register_blueprint(views, url_prefix='/')
    app.register_blueprint(auth, url_prefix='/')

    from .models import User, Books, Favourites, UserView, BooksView, MyAdminIndexView, FavouritesView, MainPageView

    admin = Admin(app, index_view=MyAdminIndexView())

    login_manager = LoginManager()
    login_manager.login_view = 'auth.login'
    login_manager.init_app(app)

    from .functions import hash_user_password

    admin.add_view(UserView(User, db.session))
    admin.add_view(BooksView(Books, db.session))
    admin.add_view(FavouritesView(Favourites, db.session))
    admin.add_view(MainPageView(name="Return to main page"))

    @login_manager.user_loader
    def load_user(id):
        return User.query.get(int(id))

    return app

```

Rysunek 7 Plik __init__.py

Zadaniem pliku models.py była implementacja mapowania obiektowo-relacyjnego, czyli ORM. Dzięki takiemu zabiegowi jesteśmy w stanie w aplikacji używać obiektów, które reprezentują poszczególne tabele z podłączonej bazy danych. Poniżej możemy zobaczyć wycinek pliku models.py.

```

class Books(db.Model):
    id = db.Column(db.Integer, primary_key=True, unique=True, autoincrement=True)
    BookName = db.Column(db.String(50), nullable=False)
    AuthorsFirstName = db.Column(db.String(50), nullable=False)
    AuthorsSurname = db.Column(db.String(50), nullable=False)
    users = db.relationship('Favourites', backref='Book')

    def __repr__(self):
        return '<Books %r>' % (self.BookName)

```

Rysunek 8 Plik models.py

Jak widać klasa Books z aplikacji webowej posiada takie same atrybuty jak jej odpowiednik z bazy danych MySQL. Widoczna jest również relacja z tabelą Favourites, analogicznie do tego co pokazywał nam model fizyczny z poprzedniego rozdziału.

Kolejnym plikiem który pozwolił na częściowy podział logiki aplikacji webowej jest plik auth.py. Zadaniem tego pliku było przechowywanie metod dotyczących uwierzytelniania użytkownika. Znajdowały się w nim takie funkcje jak zaloguj, wyloguj oraz zarejestruj użytkownika, wykorzystujące bibliotekę flask_login. Jak widać na poniższym wycinku tego pliku funkcje zawierały również informacje o adresie podstrony oraz obsługiwanych metodach HTTP.


```

auth = Blueprint('auth', __name__)

@auth.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')

        user = User_loginprv.query.filter_by(email=email).first()
        if user:
            if check_password_hash(user.password, password):
                try:
                    login_user(user, remember=True)
                    flash('Login successful', category='success')
                except:
                    flash("Cannot login to that account", category='error')
                    return redirect(url_for('views.home'))
            else:
                flash('Wrong password', category='error')
        else:
            flash('User does not exist', category='error')
    return render_template("login.html", user=current_user)

```

Rysunek 9 Plik auth.py

Plik views.py pełnił zbliżoną funkcję do poprzednio omawianego pliku, w tym przypadku jednak przechowywane były tutaj informacje o wszystkich innych podstronach które nie dotyczyły uwierzytelniania użytkowników. Poniżej możemy zaobserwować wycinek z omawianego pliku prezentujący jak po stronie serwera wygląda strona główna po zalogowaniu się.

```

views = Blueprint('views', __name__)

@views.route('/', methods=['GET', 'POST'])
@login_required
def home():
    if request.method == 'GET':
        books = Books_userprv.query.all()
        user_has_books = Favourites_userprv.query.filter_by(User_id=int(current_user.id)).all()
        userbooks = []
        for book in user_has_books:
            favbook = book.Books_id
            userbooks.append(int(favbook))
        return render_template("home.html", user=current_user, books=books, userbooks=userbooks)
    if request.method == 'POST':
        imie = request.form.get('imie')
        nazwisko = request.form.get('nazwisko')
        tytul = request.form.get('tytul')

        books = Books_userprv.query.filter(Books_userprv.AuthorsFirstName.like(f'{imie}%')).filter(Books_userprv.Authors
        user_has_books = Favourites_userprv.query.filter_by(User_id=int(current_user.id)).all()
        userbooks = []
        for book in user_has_books:
            favbook = book.Books_id
            userbooks.append(int(favbook))
        return render_template("home.html", user=current_user, books=books, userbooks=userbooks)

```

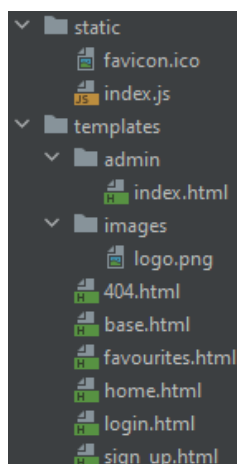
Rysunek 10 Plik views.py

Ostatnim plikiem z głównego folderu jest plik functions.py. Zadaniem tego pliku było przechowywanie wszelkich funkcji, które ze względu na swoją strukturę lub zastosowanie nie pasują to żadnej innej kategorii. Taką funkcją będzie na przykład funkcja hashująca hasła podczas umieszczania ich w bazie danych.

2.2 Frontend aplikacji webowej

2.2.1 Struktura i realizacja

Stworzony kod odpowiadający za *frontend* aplikacji rezyduje w dwóch podfolderach folderu website, nazwy tych dwóch folderów to odpowiednio static oraz templates. Folder templates jest kluczowy ze względu na to że w nim przechowywane są wszystkie pliki .html, zaś w folderze static plik .js oraz favicon.ico.



Rysunek 11 Struktura plików *frontendu*

Kluczowym plikiem tej struktury był plik base.html. Przy wykorzystaniu silnika szablonów Jinja2, który wspiera framework Flask, ten plik był nadpisywany. Dzięki temu każdy kolejny plik jak home.html czy login.html bazował na pliku base.html i podmieniał jedynie konkretne, wcześniej wyszczególnione elementy strony. Poniżej możemy zobaczyć wycinek tego jak prezentował się plik base.html.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link
      rel="stylesheet"
      href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
      integrity="sha384-VKoo8x4CGs03+Hhvxv8T/Q5PaXtkKtu6ug5TOeNV6gB1FeWPGFN9MuhOf23Q9Ifjh"
      crossorigin="anonymous"
    />
    <link
      rel="stylesheet"
      href="https://stackpath.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css"
      crossorigin="anonymous"
    />

    <title>{% block title %}Home{% endblock %}</title>
    <link rel="shortcut icon" href="{{ url_for('static', filename='favicon.ico') }}" />
  </head>
  <body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
      <button
        class="navbar-toggler"
        type="button"
        data-toggle="collapse"
        data-target="#navbar"
      >
        <span class="navbar-toggler-icon"></span>
      </button>
```

Rysunek 12 Plik base.html

Jak widać zarówno na rysunku 12, jak i rysunku 13, Jinja2 korzysta z bloków, które można nadpisywać w kolejnych plikach HTML nadpisując elementy danego bloku.

```
<title>{% block title %}Home{% endblock %}</title>
```

Rysunek 13 Blok silnika szablonów Jinja2

Poniżej możemy zobaczyć w jaki sposób nadpisywane były bloki w pliku HTML odpowiadającym za generowanie strony rejestracji.

```
{% extends "base.html" %}
{% block title %}Sign up{% endblock %}

{% block content %}
<form method="POST">
  <h3 align="center">Sign Up</h3>
  <div class="form-group">
    <label for="email">Email Address</label>
    <input
      type="email"
      class="form-control"
      id="email"
      name="email"
      placeholder="Enter email"
    />
  </div>
</form>
</div>
```

Rysunek 14 Nadpisywanie bloków Jinja2 w plikach HTML

Kolejną możliwością udostępnioną przez silnik szablonów Jinja2 było korzystanie ze składni Python'a w kodzie pliku HTML. Poniżej możemy zobaczyć przykład wykorzystania tej funkcjonalności.

```
{% for book in books %}
<tr>
  <td>{{book.BookName}}</td>
  <td>{{book.AuthorsFirstName}}</td>
  <td>{{book.AuthorsSurname}}</td>

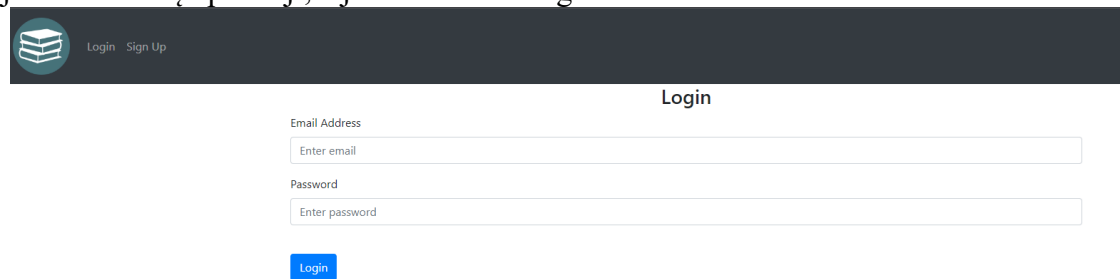
  <td>
    {% if book.id in userbooks %}
    <span>&#9989; Favourite Book</span>
    {% else %}
    <form method="POST" action="/manage_favourites">
      <button type="submit" class="btn btn-primary" name="add" value={{book.id}}> Add to favourites </button>
    </form>
    {% endif %}
  </td>
</tr>
{% endfor %}
```

Rysunek 15 Korzystanie ze składni Python'a w plikach HTML

Jak widać powyżej dzięki temu mogliśmy korzystać z takich elementów jak pętle oraz obiekty, co nie jest typowe dla czystego HTML'a. Takie rozwiązanie pozwoliło na stworzenie strony reagującej dynamicznie na zmiany i nasze potrzeby.

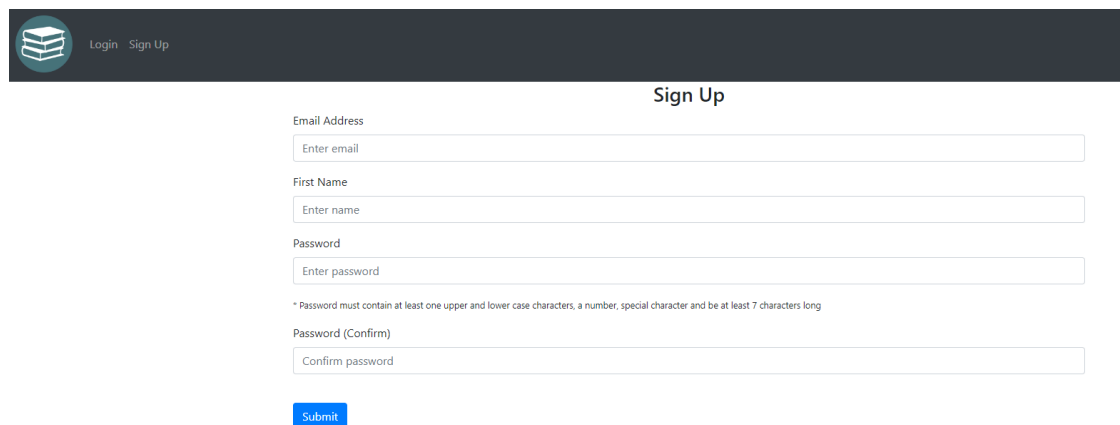
2.2.2. Widoki aplikacji

W tym podrozdziale zostanie zilustrowane jak prezentują się stworzone widoki aplikacji, które są przeznaczone dla użytkownika do interakcji z aplikacją webową. Poniżej możemy zobaczyć pierwszy widok, który niezalogowany użytkownik ujrzy na oczy w momencie wejścia na stronę aplikacji, a jest nim ekran logowania.



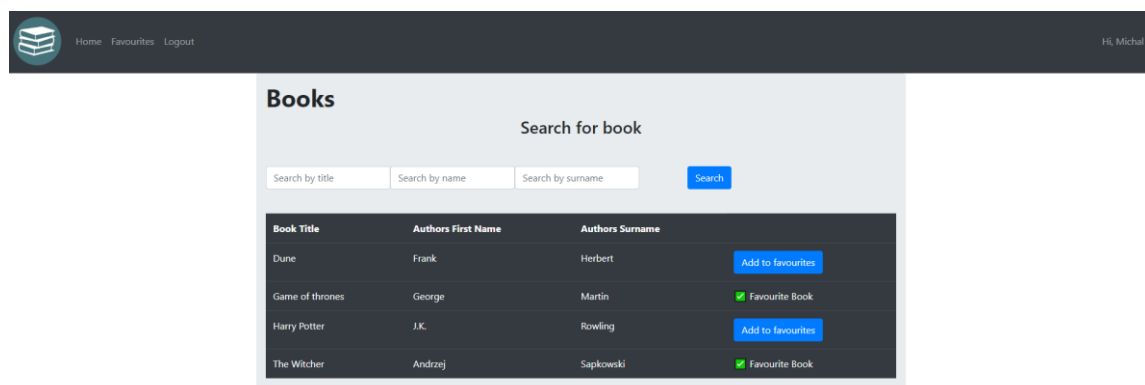
Rysunek 16 Ekran logowania

Jak widać po wejściu na stronę poza zakładką logowania mamy również dostęp do podstrony odpowiadającej za rejestrację użytkownika.



Rysunek 17 Ekran rejestracji

Po zalogowaniu się ujrzymy stronę główną aplikacji, udostępniającą moduł wyszukiwania książek z bazy oraz dodawanie ich do ulubionych.

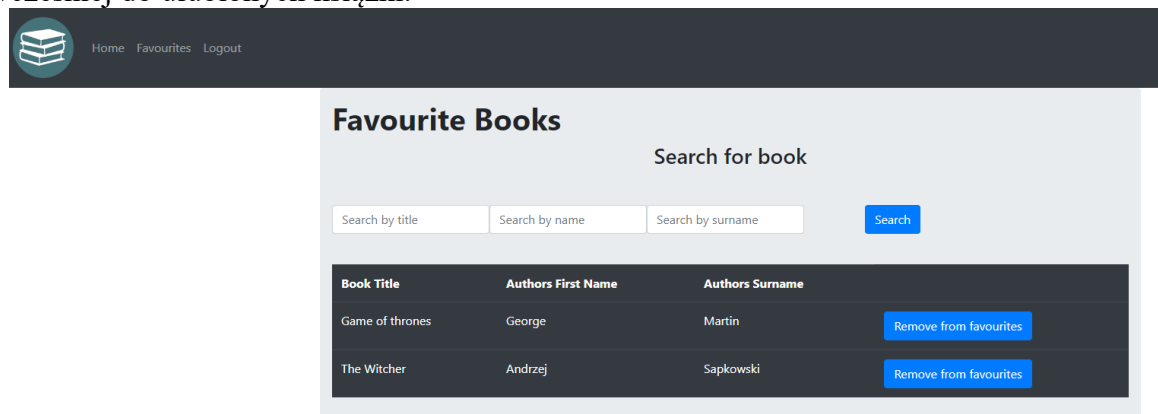


Book Title	Authors First Name	Authors Surname	
Dune	Frank	Herbert	Add to favourites
Game of thrones	George	Martin	✓ Favourite Book
Harry Potter	J.K.	Rowling	Add to favourites
The Witcher	Andrzej	Sapkowski	✓ Favourite Book

Rysunek 18 Strona główna aplikacji

Jak widać zalogowany jest zwykły użytkownik o nazwie „Michał”, ma on dostęp jedynie do podstawowych funkcjonalności aplikacji.

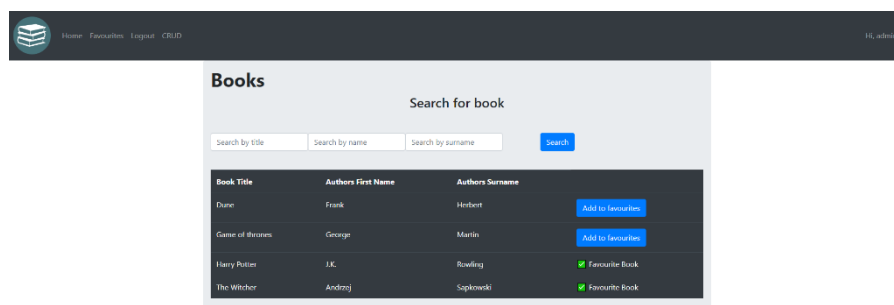
Po przejściu na podstronę „Favourites” naszym oczom ukażą się wszystkie dodane wcześniej do ulubionych książki.



Rysunek 19 Strona Favourites

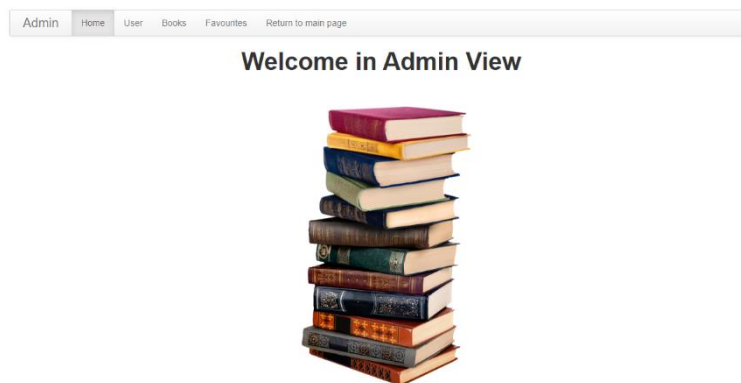
Jak widać powyżej z tej podstrony możemy usuwać książki z ulubionych. Ostatni przycisk dostępny na stronie „Logout” powoduje wylogowanie użytkownika i powrót do strony logowania.

W momencie, kiedy zalogowany jest administrator poza funkcjonalnościami użytkownika ma on dostęp do panelu admina o nazwie „CRUD”.



Rysunek 20 Strona główna po zalogowaniu się jako administrator

Po kliknięciu w nowy przycisk zostaniemy przekierowani do panelu administratora, przeznaczonego do zarządzania bazą danych podłączoną do aplikacji webowej.



Rysunek 21 Panel administratora – strona główna

Korzystając z tego widoku możemy dodawać, odczytywać, modyfikować i usuwać wszelkie wpisy z bazy danych.

Admin	Home	User	Books	Favourites	Return to main page
-------	------	------	-------	------------	---------------------

List (2)	Create	With selected ▾
----------	--------	-----------------

	Email	Name	Password	
<input type="checkbox"/>	admin@admin.pl	admin	sha256\$ysmqXxH4jxs70gr\$d085cc7eca7f1b4fb49787685b9644620dabc52b9ceb1326892e51904069563c	⊙
<input type="checkbox"/>		Michał	sha256\$wnmni7XMLowp69j8Sc8e48f19d860f98e88e97f5d379098785a39b3a2bce5cec07c31ea5c32a5b03e	●

Rysunek 22 Panel administratora – zarządzanie użytkownikami

Admin	Home	User	Books	Favourites	Return to main page
-------	------	------	-------	------------	---------------------

List (4)	Create	With selected ▾
----------	--------	-----------------

	Bookname	Authorsfirstname	Authorssurname
<input type="checkbox"/>	Dune	Frank	Herbert
<input type="checkbox"/>	Game of thrones	George	Martin
<input type="checkbox"/>	Harry Potter	J.K.	Rowling
<input type="checkbox"/>	The Witcher	Andrzej	Sapkowski

Rysunek 23 Panel administratora – zarządzanie książkami

Admin	Home	User	Books	Favourites	Return to main page
-------	------	------	-------	------------	---------------------

List (4)	Create	With selected ▾
----------	--------	-----------------

	Book	User
<input type="checkbox"/>	<Books 'Game of thrones'>	<User 'morlun111@gmail.com'>
<input type="checkbox"/>	<Books 'Harry Potter'>	<User 'admin@admin.pl'>
<input type="checkbox"/>	<Books 'The Witcher'>	<User 'admin@admin.pl'>
<input type="checkbox"/>	<Books 'The Witcher'>	<User 'morlun111@gmail.com'>

Rysunek 24 Panel administratora – zarządzanie ulubionymi

Dostęp do takiego panelu dla osób nieuprawnionych mógłby być katastrofalny w skutkach i doprowadzić do odcięcia dostępu od niej uprawnionym osobom. Mogłyby zostać wprowadzone również nieautoryzowane zmiany powodujące wyciek danych i spowodować kłopoty na firmę odpowiedzialną za nią.

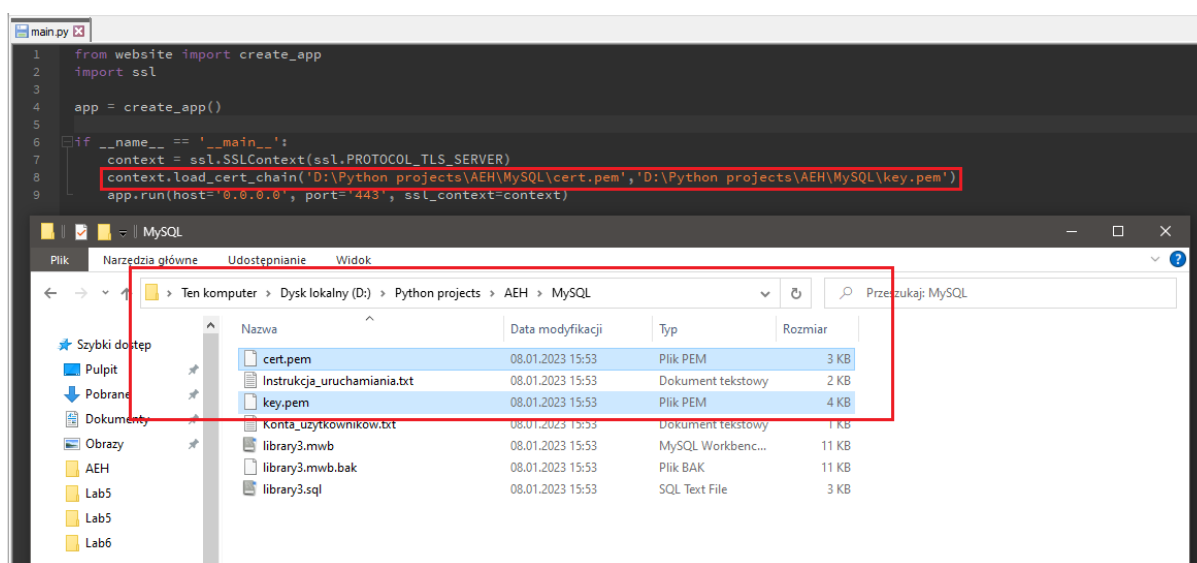
3. Docker

3.1 Wymagania

Do poprawnego działania kontenera będzie potrzebne zainstalowanie aplikacji Docker Desktop, którą można pobrać z oficjalnej strony producenta. Aplikacja ta pozwala na łatwe budowanie oraz zarządzanie kontenerami.

<https://www.docker.com/products/docker-desktop/>

Po pobraniu i zainstalowaniu aplikacji, jako następny krok należy sprawdzić, czy ścieżka do plików cert.pem, oraz key.pem została poprawnie podana w pliku main.py



Rysunek 25. Poprawna ścieżka do plików cert.pam i

3.2 Uruchomienie

Korzystając z Wiersza poleceń lub PowerShell'a uruchomionych najlepiej z uprawnieniami administratora, należy przejść do folderu zawierającego pliki main.py, Dockerfile, docker-compose.yml, oraz requirements.txt. Aby uruchomić plik docker compose a tym samym zbudować kontenery aplikacji i bazy danych, należy wywołać polecenie `docker compose up`, lub `docker compose up -d`, które uruchomi plik w trybie detached, czyli kontenery zostaną uruchomione w tle.

```
Administrator: Windows PowerShell
PS D:\Python projects\AEH\aeH-pap-project-master> dir

Directory: D:\Python projects\AEH\aeH-pap-project-master

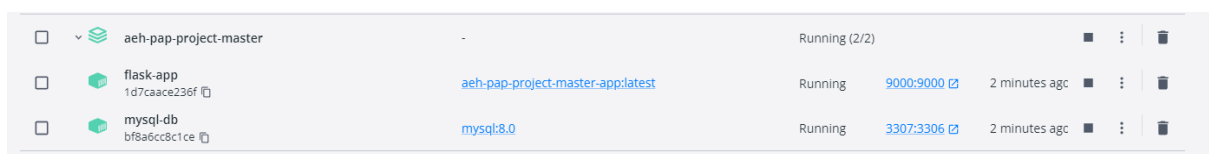
Mode                LastWriteTime         Length Name
----                -
d-----          08.01.2023    15:59             db
d-----          08.01.2023    15:59          website
d-----          08.01.2023    16:01          __pycache__
-a-----          08.01.2023    15:53          911 docker-compose.yml
-a-----          08.01.2023    15:53          323 Dockerfile
-a-----          08.01.2023    15:59          320 main.py
-a-----          08.01.2023    15:53          119 requirements.txt

PS D:\Python projects\AEH\aeH-pap-project-master> docker compose up -d

- Network aeH-pap-project-master_backend      Created                                0.8s
- Volume "aeH-pap-project-master_dbstore"     Created                                0.0s
- Volume "aeH-pap-project-master_flask-app-cache" Created                                0.0s
- Container mysql-db                           Started                               1.2s
- Container flask-app                          Started                               2.3s
PS D:\Python projects\AEH\aeH-pap-project-master>
```

Rysunek 26. Uruchomienie docker compose

W tym momencie aplikacja powinna być w pełni dostępna w przeglądarce internetowej pod adresem <http://localhost:9000/>. Ponadto status naszych kontenerów możemy sprawdzić w Docker Desktop.



Rysunek 27. Widok z Docker Desktop

3.3 Dockerfile

Plik Dockerfile jest plikiem, który buduje obraz dockerowy. Zaletą stworzenia takiego pliku jest to, że nie trzeba za każdym razem ręcznie wpisywać komend z wiersza poleceń, tylko mamy je wszystkie w jednym pliku, co przyczynia się do poprawy automatyzacji i szybkości. Zawartość pliku Dockerfile.

```
1. FROM python:3.10-slim
2.
3. COPY requirements.txt requirements.txt
4.
5. RUN apt-get update
6. RUN apt-get install -y gcc
7. RUN apt-get install -y default-libmysqlclient-dev
8. RUN pip install -r requirements.txt
9.
10.     COPY . \app
11.     WORKDIR \app
12.     ENV FLASK_APP=main.py
13.
14.     EXPOSE 9000
15.     CMD ["flask", "run", "--host=0.0.0.0", "--port=9000"]
```


W pierwszej linii pliku definiowany jest język, z którego obraz będzie inicjowany, w tym przypadku python 3.10 w wersji slim. Następnie komenda COPY kopiuje plik z requirements.txt do wnętrza kontenera i zapisuje go pod taką samą nazwą. Plik ten zawiera niezbędne do działania aplikacji biblioteki, głównie związane z Flask’iem i MySQL’em. Zawartość tego pliku to:

```
flask
flask_sqlalchemy==2.5.1
flask_login
flask_admin
werkzeug
python-gettext
mysql-connector-python
mysqlclient
```

W dalszej części pliku Dockerfile wywoływane są 3 komendy RUN, których celem jest instalacja dodatkowych pakietów w kontenerze. Takie rozwiązanie zostało przyjęte, aby zapewnić pełną skuteczność podczas instalacji wymienionych wyżej bibliotek. Czwarta, czyli ostatnia komenda RUN instaluje biblioteki z pliku requirements.txt.

Dziesiąta linia pliku kopiuje aplikację do folderu “app” a następna definiuje go jako folder roboczy aplikacji. W linii dwunastej tworzona jest zmienna środowiskowa FLASK_APP i przypisana zostaje do niej wartość main.py.

Komenda EXPOSE informuje, że aplikacja będzie wystawiona na porcie 9000. Ostatnia linia pliku Dockerfile definiuje komendę, która zostanie wywołana przy starcie kontenera.

3.4 Docker compose

Docker compose pozwala uruchamiać kilka kontenerów dockerowych na raz. W naszym przypadku kontenerami tymi są kontener aplikacji Flask i kontener bazy danych MySQL. Zawartość pliku docker-compose.yml.

```
1. version: '3.11'
2.
3. networks:
4.   backend:
5.
6. volumes:
7.   flask-app-cache:
8.   dbstore:
9.
10.   services:
11.     db:
12.       container_name: mysql-db
13.       image: "mysql:8.0"
14.       restart: "no"
15.       ports:
16.         - "3307:3306"
17.       environment:
18.         MYSQL_ROOT_PASSWORD: root
19.         MYSQL_DATABASE: library3
20.         MYSQL_USER: user
21.         MYSQL_PASSWORD: password
22.       volumes:
23.         - "./db/init.sql:/docker-entrypoint-initdb.d/init.sql"
24.         - "dbstore:/var/lib/mysql"
```

```

25.         networks:
26.             backend:
27.     app:
28.         container_name: flask-app
29.         restart: "no"
30.         build:
31.             context: ./
32.             dockerfile: ./Dockerfile
33.         ports:
34.             - "9000:9000"
35.         links:
36.             - db
37.         volumes:
38.             - ".:/app"
39.             - "flask-app-cache:/root/.cache/pip"
40.         environment:
41.             PYTHONPATH: /app
42.
43.         networks:
44.             backend:
45.         depends_on:
46.             - db

```

UNI_DATABASE_URI:

mysql://user:password@db:3306/library3

W pierwszej linii definiowana jest wersja docker compose. Następnie definiowana jest sieć “backend”, w której będą się znajdować oba kontenery. Bez tego elementu komunikacja między aplikacją a bazą danych będzie niemożliwa. Kolejne linie są odpowiedzialne za zdefiniowanie wolumenów, które będą składowały dane. Jest to niezbędny element dla bazy danych a w przypadku kontenera aplikacji, wolumen pozwala na zapamiętanie niektórych danych, np. zainstalowanych bibliotek. Następnie definiujemy pod “services” zbiór parametrów kontenera z bazą danych pod “db” oraz aplikacji pod “app”.

Kolejne linie pliku odpowiadają właściwościom kontenera bazy danych tj. “container_name” przypisujące nazwę kontenera, “image” definiujące wybrany obraz, następnie opcja “restart” zapobiegająca automatycznemu restartowi kontenera. Pod “ports” definiujemy port wystawiony w kontenerze oraz port docelowy. Poniżej “environment” znajdują się zdefiniowane zmienne środowiskowe, konieczne do poprawnego działania bazy danych oraz wspomniane wolumeny.

Dla kontenera aplikacji parametry definiowane są podobnie. Do inicjacji budowy kontenera wskazujemy na dockerfile pod “build”. Aplikacja jest zależna od bazy danych, dlatego należało zdefiniować w docker compose zależność pomiędzy kontenerami korzystając z “links” odpowiadającego za połączenie sieciowe kontenerów oraz “depends_on” przez co kontener z bazą danych zostanie stworzony przed kontenerem aplikacji.

3.5 Baza Danych

Plik init.sql tworzy bazę danych oraz definiuje początkowy stan tabel. W pierwszych liniach pliku tworzona jest baza danych o nazwie “library3”. Następnie tworzona jest pierwsza tablica “user” przechowująca wartości tj.: identyfikator, adres email, nazwa oraz wartość “isAdmin” pozwalająca sprawdzić w późniejszym etapie czy użytkownik jest administratorem. Kolejna tabela “books” opisuje wartości dla książek oraz ostatnia tworzona tabela to “favourites”, która przechowuje ulubione pozycje książkowe dodane przez użytkownika.

W kolejnej części pliku od linii czterdziestej wykonywane są instrukcje “CREATE USER” tworzące pierwszych użytkowników oraz nadawane są im uprawnienia za pomocą “GRANT”.

Ostatnia część pliku od linii pięćdziesiątej odpowiada za wstępne wypełnienie bazy przez instrukcje “INSERT”.

Zawartość pliku init.sql:

```
1. - MySQL Workbench Forward Engineering
2. CREATE SCHEMA IF NOT EXISTS `library3` DEFAULT CHARACTER SET utf8 ;
3. USE `library3` ;

4. CREATE TABLE IF NOT EXISTS `library3`.`user` (
5.   `id` INT NOT NULL AUTO_INCREMENT,
6.   `email` VARCHAR(50) NOT NULL,
7.   `name` VARCHAR(50) NOT NULL,
8.   `password` VARCHAR(150) NOT NULL,
9.   `isAdmin` TINYINT NOT NULL,
10.  PRIMARY KEY (`id`),
11.  UNIQUE INDEX `UserID_UNIQUE` (`id` ASC) VISIBLE,
12.  UNIQUE INDEX `email_UNIQUE` (`email` ASC) VISIBLE)
13. ENGINE = InnoDB;

14. CREATE TABLE IF NOT EXISTS `library3`.`books` (
15.   `id` INT NOT NULL AUTO_INCREMENT,
16.   `BookName` VARCHAR(150) NOT NULL,
17.   `AuthorsFirstName` VARCHAR(50) NOT NULL,
18.   `AuthorsSurname` VARCHAR(50) NOT NULL,
19.   PRIMARY KEY (`id`),
20.   UNIQUE INDEX `BookID_UNIQUE` (`id` ASC) VISIBLE)
21. ENGINE = InnoDB;

22. CREATE TABLE IF NOT EXISTS `library3`.`favourites` (
23.   `User_id` INT NOT NULL,
24.   `Books_id` INT NOT NULL,
25.   PRIMARY KEY (`User_id`, `Books_id`),
26.   INDEX `fk_User_has_Books_Books1_idx` (`Books_id` ASC) VISIBLE,
27.   INDEX `fk_User_has_Books_User_idx` (`User_id` ASC) VISIBLE,
28.   CONSTRAINT `fk_User_has_Books_User`
29.     FOREIGN KEY (`User_id`)
30.     REFERENCES `library3`.`user` (`id`)
31.     ON DELETE NO ACTION
32.     ON UPDATE NO ACTION,
33.   CONSTRAINT `fk_User_has_Books_Books1`
34.     FOREIGN KEY (`Books_id`)
35.     REFERENCES `library3`.`books` (`id`)
36.     ON DELETE NO ACTION
37.     ON UPDATE NO ACTION)
38. ENGINE = InnoDB;

39. -- TWORZENIE KONT--
40. CREATE USER 'user2'@'localhost' IDENTIFIED BY 'user2';
41. GRANT SELECT ON library3.books TO 'user2'@'localhost';
42. GRANT SELECT,INSERT,DELETE ON library3.favourites TO
'user2'@'localhost';
43. CREATE USER 'admin2'@'localhost' IDENTIFIED BY 'adminpass2';
44. GRANT SELECT,INSERT,DELETE,UPDATE ON library3.books TO
'admin2'@'localhost';
45. GRANT SELECT,INSERT,DELETE,UPDATE ON library3.favourites TO
'admin2'@'localhost';
46. GRANT SELECT,INSERT,DELETE,UPDATE ON library3.user TO
'admin2'@'localhost';
47. CREATE USER 'login2'@'localhost' IDENTIFIED BY 'loginpass2';
```

```
48. GRANT SELECT,INSERT ON library3.user TO 'login2'@'localhost';

49. -- WYPEŁNIENIE --
50. INSERT INTO user (email,name,password, isAdmin) VALUES
("admin@admin.pl","admin","sha256$ysmqXxHAjxs70gr$d085cc7eca7f1b4fb49787
685b9644620dabc52b9ceb1326892e51904069563c",TRUE);
51. INSERT INTO books (BookName,AuthorsFirstName,AuthorsSurname) VALUES
("Call of Cthulhu","Howard","Lovecraft");
52. INSERT INTO books (BookName,AuthorsFirstName,AuthorsSurname) VALUES
("Dune","Frank","Herbert");
53. INSERT INTO books (BookName,AuthorsFirstName,AuthorsSurname) VALUES
("Game of thrones","George","Martin");
54. INSERT INTO books (BookName,AuthorsFirstName,AuthorsSurname) VALUES
("Harry Potter","J.K.","Rowling");
55. INSERT INTO books (BookName,AuthorsFirstName,AuthorsSurname) VALUES
("The Witcher","Andrzej","Sapkowski");
56. INSERT INTO favourites(User_id,Books_id) VALUES (1,1);
```

4. Wykorzystanie Github Actions do automatycznego wdrożenia obrazu Dockera na platformie Docker Hub.

4.1 Konfiguracja procesu integracji repozytorium kodu z repozytorium obrazów.

Docker Hub jest zdalnym, darmowym i globalnym repozytorium obrazów wykorzystujących technologie Docker'a służącym do ich przechowywania i zarządzania nimi. Jest szeroko wykorzystywane przeważnie w procesach CI/CD dzięki czemu najnowsza wersja oprogramowania automatycznie jest przesyłana do globalnego katalogu – takie rozwiązanie zapewnia możliwość aktualizacji istniejących obrazów używając wyłącznie natywnych komend dostarczanych przez lokalny silnik Dockera.

Github Actions jest natywnym rozwiązaniem, które umożliwia konfigurację procesów CI/CD z poziomu repozytorium kodu zawartego na platformie Github za pomocą plików YAML. Umożliwia to budowanie, testowanie oraz wdrażanie kodu w trybie ciągłym dzięki czemu wszelkie aktualizacje oprogramowania są dostarczane automatycznie po spełnieniu zadeklarowanych warunków.

W projekcie wykorzystane zostały akcje wspierane przez Github Actions w celu autonomicznego budowania oraz przysyłania gotowego obrazu z Dockerfile'a (opisanego w podpunkcie 3.3) na platformie Docker Hub.

Poniżej została przedstawiona używana konfiguracja (zgodnie z zaleceniami dokumentacji Dockera):

```
name: Adding Dockerfile to DockerHub repo.
```

```
on:
```

```
  push:
```

```
    branches:
```

```
      - "master"
```

```
jobs:
```

```
  build:
```

```
    runs-on: ubuntu-latest
```

```
    steps:
```

```
      -
```

```
        name: Checkout
```

```
        uses: actions/checkout@v3
```

```
      -
```

```
        name: Login to Docker Hub
```

```
        uses: docker/login-action@v2
```

```
        with:
```

```
          username: ${ secrets.DOCKERHUB_USERNAME }
```

```
          password: ${ secrets.DOCKERHUB_TOKEN }
```

```
      -
```

```
        name: Set up Docker Buildx
```

```
        uses: docker/setup-buildx-action@v2
```

```
      -
```

```
        name: Build and push
```

```
        uses: docker/build-push-action@v3
```

```
        with:
```

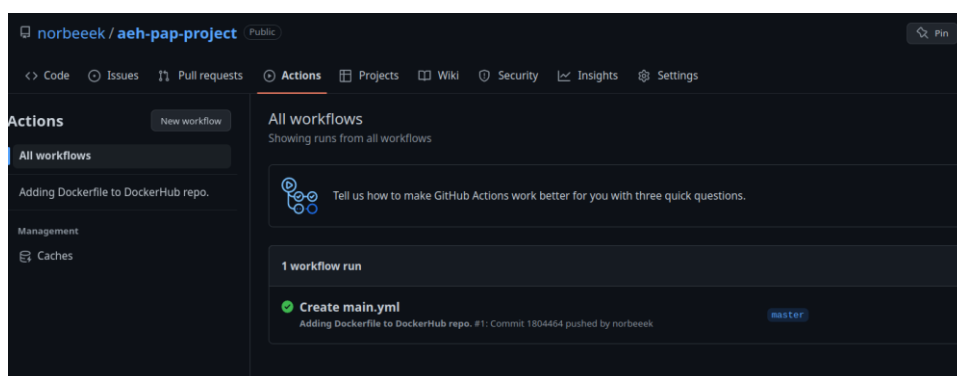
```
context: ./its_working!/projekt/aeH-pap-project-master/  
  
file: ./its_working!/projekt/aeH-pap-project-master/Dockerfile  
  
push: true  
  
tags: ${ secrets.DOCKERHUB_USERNAME }/aeH_pap_project:latest
```

Opis najwazniejszych funkcjonalnosci z podanej powyzej funkcjonalnosci:

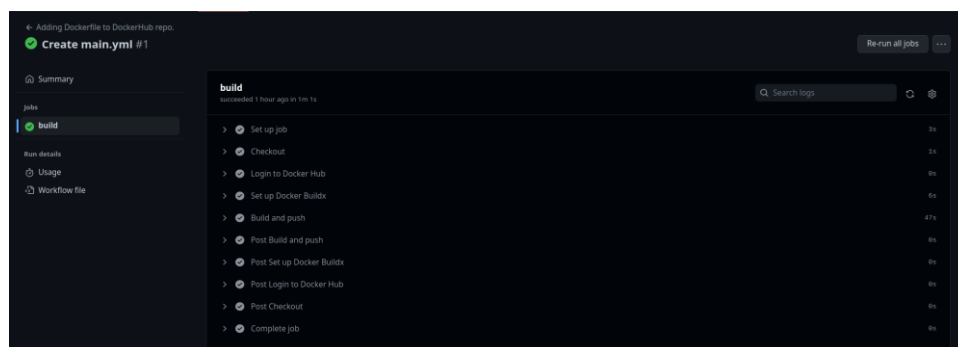
- *name* – wybrana nazwa dla procesu
- *on -> push -> branches -> “master”* - konfiguracja ta sprawia, ze dane zadania skonfigurowane ponizej zostana wykonane w przypadku jakiegokolwiek zmiany w galezi “master”
- *jobs* – podzbior *steps* definiuje akcje wykonywane w ramach platformy Github
Actions:
 - ‘*Login to Docker Hub*’ - logowanie do platformy Docker Hub uzywajac sekretnych zmiennych srodowiskowych skonfigurowanym w ustawieniach repozytorium, sa to unikalne wartosci przypisane do konta, *DOCKERHUB_USERNAME* zawiera nazwe uzytkownika natomiast *DOCKERHUB_TOKEN* przetrzymuje wartosc tokenu dostepu do platformy.
 - ‘*Set up Docker Buildx*’ - konfiguracja srodowiska, ktore posluzy do budowy obrazu.
 - ‘*Build and push*’ - kluczowa akcja podczas ktorej obraz jest budowany (bazujac na wartosciach *context* (*folder zawierajacy kluczowe pliki*) oraz *file* (*lokalizacja Dockerfile*’a)) - jesli ten krok zakonczy sie sukcesem, zbudowany obraz jest nastepnie wysylany do zdalnego repozytorium nazwany zgodnie z wartoscia zawarta w polu *tags*.

Powyzsza konfiguracja zostala zapisana w standardowej lokalizacji `{repository_root}/.github/workflows/main.yml`.

Ponizej znajduje sie widok poprawnie wykonanego zbudowania oraz wdrozenia obrazu na platforme Docker Hub.



Rysunek 28. Glowny widok zakladki Github Actions w systemie Github wraz z historia wykonania skonfigurowanych procesow.



Rysunek 29. Szczegółowy widok skonfigurowanego procesu CI/CD z uwzględnieniem poszczególnym krokow zgodnych z plikiem konfiguracyjnym.

