BD-MiniProject

Grzegorz Piśkorski: piskorski@student.agh.edu.pl

Antoni Wójcik: antoniwojcik@student.agh.edu.pl

Temat projektu:

Rezerwowanie noclegów w hotelach. Aplikacja będzie pozwalała na rezerwację pokoi w kilku wybranych hotelach.

Technologia:

MongoDB, Python Flask, Jinja2

Spis treści dokumentacji

Instrukcja uruchomienia aplikacji

1. Instalujemy odpowiednie requirements:

```
pip install -r requirements.txt
```

2. Do folderu server dodajemy plik o nazwie .env i zamieszczamy w nim login i hasło dostępu do bazy:

```
MONGODB_USERNAME = ...
MONGODB_PASSWORD = ...
```

Następnie możemy uruchomić całą aplikację z poziomu pliku app.py.

Główne funkcjonalności projektu

- możliwość zarezerwowania noclegu w jednym z dostępnych hotelów w bazie danych (wyświetlenie dostępnych pokoi w danym okresie czasu)
- możliwość zarządzania swoją rezerwacją (dodanie nowej, modyfikacja jednej z "posiadanych" rezerwacji, rezygnacja z rezerwacji)

Struktura bazy danych

Hotels

```
{
    "name": string,
    "street": string,
    "city": string,
    "zip_code": string
}
```

Rooms

```
{
    "hotel_id": ObjectId(),
    "room_type": string,
    "room_number": string,
    "price_per_night": number,
    "is_available": boolean,
    "bookings": [
      {
        "booking_id": ObjectId
        "customer_id": ObjectId,
        "date_from": date,
        "date_to": date
      }
    ]
}
```

Customers

```
{
    "name": string,
    "surname": string,
    "email": string,
    "bookings": [ {
        "booking_id": ObjectId <- generowanie automatyczne
        "room_id": ObjectId,
        "date_from": string,
        "date_to": string
    } ],
    "password": string
}
```

Booking_logs

```
{
  "booking_id": ObjectId
  "customer_id": ObjectId,
  "room_id": ObjectId,
  "date_from": date,
  "date_to": date
}
```

Metody i funkcje operujące na bazie danych

Część z nich nie jest wykorzystywana w aplikacji, ponieważ nie udało się zaimplementować niektórych funkcjonalności, jednak przydatne są przy zarządzaniu bazą danych

- Hotels
  - `get_all_hotels()` - zwraca listę z danymi o hotelach
  - `get_all_cities()` - zwraca listę miast, z których są hotele (przydatna w filtrach)
  - `add_hotel(name, street, city, zip_code, img)` - dodaje hotel do bazy
  - `remove_hotel(hotel_id)` - usuwa hotel z bazy wraz z jego pokojami (zał. nie ma żadnych rezerwacji na pokoje z danego hotelu)
- Rooms
  - `get_wrong_bookings(room_id, check_in, check_out, booking_id)` - zwraca listę rezerwacji nachądzących na podany okres czasu
  - `get_occupied_rooms(check_in, check_out)` - zwraca listę pokoi, które są zarezerwowane w podanym okresie czasu
  - `add_room(hotel_id, room_type, room_number, ppn, img, availability)` - dodaje pokój od bazy
  - `remove_room(room_id)` - usuwa pokój z bazy danych
  - `set_price_per_night(room_id, new_price)` - zmienia cenę za noc danego pokoju

    - `set_availability(room_id, availability)` - uaktualnia dostępność danego pokoju (chodzi o dostępność w razie np. remontu pokoju)
- Customers
    - `add_customer(name, surname, mail, passwd)` - dodaje nowego użytkownika do bazy danych
    - `get_all_user_bookings(user_id)` - zwraca listę wszystkich rezerwacji danego użytkownika
    - `get_user_email(email)` - zwraca użytkownika o podanym emailu - przydatna w uwierzytelnianiu użytkownika
    - `remove_customer(customer_id)` - usuwa użytkownika z bazdy danych

Metody i funkcje korzystające z więcej niż jednej kolekcji

- `can_be_booked(room_id, check_in, check_out, booking_id)` - funkcja pomocnicza, korzystająca z get_wrong_bookings - sprawdza czy można zarezerwować podany pokój na konkretny termin
- `push_bookings(booking_id, customer_id, room_id, check_in, check_out)` - funkcja pomocnicza - dodaje odpowiednie dane do odpowiedniego pokoju i użytkownika na temat rezerwacji
- `add_new_booking(customer_id, room_id, check_in, check_out)` - dodanie nowej rezerwacji - dodawana jest w kolekcji Customers i Rooms (o ile to możliwe)
- `change_booking(customer_id, room_id, booking_id, check_in, check_out)` - zmiana rezerwacji danego pokoju przez klienta, wprowadza zmiany w obu kolekcjach (o ile to możliwe)
- `filter_rooms(check_in, check_out, min_price, max_price, room_type, hotel_city)` - zwraca listę pokoi, spełniających podane kryteria (np. cena min i max, liczba osób w pokoju, pokoje wolne w danym terminie itp.)
- `remove_booking(booking_id, customer_id, room_id)` - usuwa danę rezerwację z obu kolekcji - Rooms i Customers
- `add_validators()` - dodaje do bazy danych walidatory, których schemat pokazany jest poniżej

Opis kodu najważniejszych funkcjonalności projektu

Rezerwacja pokoju, zmiana terminów już zarezerwowanego pokoju

Aby zrozumieć najważniejsze funkcje aplikacji, należy zacząć od funkcji pomocniczych. Funkcja `get_wrong_bookings` zwraca listę rezerwacji kolidujących z wybranym przez nas terminem dla danego pokoju. Domyślnie przyjmuje ona id pokoju oraz daty zameldowania i wymeldowania. Może jednak przyjąć dodatkowo id_bookingu - nie jest on wówczas brany pod uwagę przy analizie kolidujących terminów; ma to znaczenie podczas zmiany daty posiadanego już przez nas bookingu. Funkcja może nie przyjąć również id pokoju - zwróci listę wszystkich kolidujących z danym terminem bookingów.
Poniżej omówimy kod funkcji waz zapytaniem wysyłanym do MongoDB:

```
def get_wrong_bookings(room_id: ObjectId, check_in: datetime, check_out: datetime,
booking_id: ObjectId):
    # pipeline wywoływany funkcją aggregate (nazwany query) składa się z
następujących etapów:
    query = [
        {   # Etap 1 - wybierz pokoje o podanym _id, dostępne do
użytku(is_available)
            '$match': {
```

```
                '_id': {'$exists': 1},  # zapytanie we wstępnej wersji ustawione
jest, aby zwracało każde pokoje
                'is_available': True
            }
        },
        {   # Etap 2 - pozbycie się nieporzebnych pól
            '$project': {
                'bookings': 1
            }
        },
        {   # Etap 3 - rozpakowanie dla każdego pokoju tablicy bookingów złożonych
na niego - tworzy się dzięki temu kolekcja wszystkich rezerwacji spełniających
dotychczasowe warunki.
            '$unwind': '$bookings'
        }
    ]
    # Jeżeli szukami kolizji dla danego pokoju, w tym miejscu dokonujemy
modyfikacji naszego query
    if room_id is not None:
        query[0]['$match']['_id'] = room_id

    # Gdy chcemy wykluczyć rezerwacje o danym id, dodawany jest dodatkowy etap do
tworzonego pipeline
    if booking_id is not None:
        query.append({  # Etap dodatkowy - filtrowanie bookingów nie będących
podanym przez nas id
            "$match": {
                "bookings.booking_id": {'$nin': [ObjectId(booking_id)]}
            }
        })
    query.append({  # Etap 4 - wybranie bookingów kolidujących z naszym terminem
        '$match': {
            '$or': [
                {
                    '$and': [
                        {
                            'bookings.date_from': {
                                '$gte': check_in
                            }
                        }, {
                            'bookings.date_from': {
                                '$lt': check_out
                            }
                        }
                    ]
                }, {
                    '$and': [
                        {
                            'bookings.date_from': {
                                '$gte': check_in
                            }
                        }, {
                            'bookings.date_to': {
                                '$lte': check_out
```

```
                            }
                        }
                    ]
                }, {
                    '$and': [
                        {
                            'bookings.date_from': {
                                '$lte': check_in
                            }
                        }, {
                            'bookings.date_to': {
                                '$gte': check_out
                            }
                        }
                    ]
                }, {
                    '$and': [
                        {
                            'bookings.date_to': {
                                '$gt': check_in
                            }
                        }, {
                            'bookings.date_to': {
                                '$lte': check_out
                            }
                        }
                    ]
                }
            ]
        }
    })

    # funkcja zwraca listę kolidujących bookingów
    return list(mongo.rooms.aggregate(query))
```

Aby sprawdzić, czy pokój może zostać zarezerwowany, używamy funkcji `can_be_booked()`. Nie jest ona skomplikowana, bazuje na poznanej wyżej funkcji `get_wrong_bookings()`, sprawdzając, czy zwracana lista jest pusta:

```
def can_be_booked(room_id: ObjectId, check_in: datetime, check_out: datetime,
booking_id: ObjectId = None):
    if check_in >= check_out:
        print("[SERVER] Check in date must be less than check out date.")
        return False

    bookings = get_wrong_bookings(room_id, check_in, check_out, booking_id)

    return len(bookings) == 0
```

Posiadając te funkcje, jesteśmy w stanie tworzyć bookingi oraz zmieniać ich daty jeżeli użytkowanik sobie tego zażyczy.
Aby dodać booking, korzystamy z funkcji `add_new_booking()`:

```
def add_new_booking(customer_id: str, room_id: str, check_in: datetime, check_out:
datetime):
    # konwersja id klienta i pokoju na obiekt typu ObjectId
    try:
        customer_id = ObjectId(customer_id)
        room_id = ObjectId(room_id)
    except Exception as e:
        print("[SERVER]", e)
        return False

    # Następnie sprawdzamy, czy można stworzyć taki booking funkcją can_be_booked
bez argumentu customer_id
    if can_be_booked(room_id, check_in, check_out):
        # Jeżeli tak, tworzymy nowe Id dla bookingu
        booking_id = ObjectId()
        # Następnie wypychamy je do bazy danych (funkcja opisana poniżej)
        return push_bookings(booking_id, customer_id, room_id, check_in,
check_out)
    else:
        print("[SERVER] Term is colliding.")
        return False
```

Rezerwacja umieszczana jest w bazie przy pomocy funkcji `push_bookings`:

```
def push_bookings(booking_id: ObjectId, customer_id: ObjectId, room_id: ObjectId,
check_in: datetime,
                  check_out: datetime):
    # Utworzenie struktur słownikowych reprezentujących booking w kolekcji Rooms i
Customers
    booking_in_customers = {
        "booking_id": booking_id,
        "room_id": room_id,
        "date_from": check_in,
        "date_to": check_out
    }
    booking_in_rooms = {
        "booking_id": booking_id,
        "customer_id": customer_id,
        "date_from": check_in,
        "date_to": check_out
    }

    # Dodanie rezerwacji do kolekcji Rooms, dokładniej do tablicy bookings dla
danego pokoju
    room_update = mongo.rooms.update_one({"_id": room_id}, {"$push": {"bookings":
booking_in_rooms}})
```

```
    if room_update.matched_count <= 0:
        print("[SERVER] Failed to add booking to a room.")
        return False

    # Dodanie rezerwacji do kolekcji Customer, dokładniej do tablicy bookings dla
podanego klienta
    customer_update = mongo.customers.update_one({"_id": customer_id},
                                                  {"$push": {"bookings":
booking_in_customers}})
    if customer_update.matched_count <= 0:
        print("[SERVER] Failed to add booking to a customer.")
        return False
    print("[SERVER] Successfully booked a room.")
    return True
```

Użytkownik posiadający już booking ma możliwość zmiany jego daty. W naszej aplikacji odbywa się to w sposób następujący, przy pomocy funkcji `change_booking()`. W kodzie użyta jest znana już funkcja `can_be_booked()`, jednak tym razem posiada dodatkowy argument z id bookingu, aby nie brać go pod uwagę przy znajdowaniu kolizji (inaczej zawsze z nowym terminem kolidować będzie aktualny booking, którego daty chcemy zmienić):

```
def change_booking(customer_id: str, room_id: str, booking_id: str, check_in:
datetime, check_out: datetime):
    try:
        room_id = ObjectId(room_id)
        booking_id = ObjectId(booking_id)
        customer_id = ObjectId(customer_id)
    except Exception as e:
        print("[SERVER]", e)
        return False
    # Sprawdzanie, czy nowa data jest osiągalna pod względem dostępności pokoju
    if can_be_booked(room_id, check_in, check_out, booking_id):

        # update w kolekcji Customers
        customer_update = mongo.customers.update_one(
            {   # zastosujemy update dla wybranego klienta i konkretnego bookingu
                "_id": ObjectId(customer_id),
                "bookings.booking_id": booking_id
            },
            {   # zmiana dat
                "$set": {
                    "bookings.$.date_from": check_in,
                    "bookings.$.date_to": check_out
                }
            }
        )

        if customer_update.matched_count <= 0:
            print("[SERVER] Failed to add booking to a room.")
            return False
```

```
        # update w kolekcji Rooms
        room_update = mongo.rooms.update_one(
            {   # zastosujemy update dla wybranego pokoju i konkretnego bookingu
                "_id": room_id,
                "bookings.booking_id": booking_id
            },
            {   # zmiana dat
                "$set": {
                    "bookings.$.date_from": check_in,
                    "bookings.$.date_to": check_out
                }
            }
        )

        if room_update.matched_count <= 0:
            print("[SERVER] Failed to add booking to a room.")
            return False
        return True
    else:
        print("[SERVER] You cannot rebook this room.")
        return False
```

Kolejną z najważniejszych funkcjonalności jest filtrowanie dostępnych pokoi. Aby zrozumieć kod, należy najpierw zapoznać się z funkcją `get_occupied_rooms()`. Korzysta ona ze znanej już nam funkcji `get_wrong_bookings()`, następnie zamienia listę kolidujących rezerwacji na listę id pokoi, których te rezerwacje dotyczą.

```
def get_occupied_rooms(check_in: datetime, check_out: datetime):

    booked_rooms = get_wrong_bookings(None, check_in, check_out, None)
    res: set = set()
    for i in booked_rooms:
        res.add(i['_id'])
    return list(res)
```

Filtrowanie realizujemy przy pomocy `filter_rooms()`. Idea jest następująca: funkcja posiada pipeline, który domyślnie wyświetli wszystkie pokoje w bazie danych. W miarę dodawania kolejnych filtrów, zmieniane są poszczególne etapy tego pipeline'a, aby na koniec wysłać go do bazy poleceniem aggregate:

```
def filter_rooms(check_in: datetime = datetime(2400, 1, 1), check_out: datetime =
datetime(2400, 1, 2), min_price: float = None, max_price: float = None,
                 room_type: int = None, hotel_city: str = None):
    # Ucięcie części godzinowej z daty
    if check_in is None:
        check_in_fixed = datetime.now().date()
        check_in_fixed = datetime.combine(check_in_fixed, datetime.min.time())
    else:
        check_in_fixed = check_in
```

```
    # Następnie tworzymy czarną listę Id pokoi, na które na pewno nie może zostać złożona rezerwacja
    black_list = get_occupied_rooms(check_in_fixed, check_out)

    # pipeline query:
    query = [
        {   # Etap 1 -> wybór pokoi dostępnych do wynajęcia, o cenie z odpowiedniego przedziału i odpowiednim typie pokoju
            # ich Id nie może zawierać się na czarnej liście
            '$match': {
                'is_available': True,
                '_id': {
                    '$nin': black_list
                },
                'price_per_night': {
                    '$gte': 0.0,
                    '$lt': 100000000.0
                },
                'room_type': {'$exists': 1}
            }
        }, {    # Etap 2 -> Dołączenie dla wynikowych pokoi informacji o hotelu do którego należą
            '$lookup': {
                'from': 'Hotels',
                'localField': 'hotel_id',
                'foreignField': '_id',
                'as': 'hotel_info'
            }
        }, {    # Etap 3 -> Rozpakowanie tych informacji, aby stały się obiektem
            '$unwind': '$hotel_info'
        }, {    # Wyświetlanie interesujących nas informacji
            '$project': {
                '_id': 0,
                'room_id': '$_id',
                'room_type': 1,
                'price_per_night': 1,
                'room_imgUrl': '$imgUrl',
                'hotel_name': '$hotel_info.name',
                'hotel_street': '$hotel_info.street',
                'hotel_city': '$hotel_info.city'
            }
        }, {    # Wyświetlanie pokoi z konkretncyh miast
            '$match': {
                'hotel_city': {
                    '$exists': 1
                }
            }
        }
    ]
    # Modyfikacje zapytania, w zależności od użytych filtrów:
    if min_price is not None:
        # ustawienie minimalnej ceny z 0 na min_price
        query[0]['$match']['price_per_night']['$gte'] = min_price
    if max_price is not None:
```

```
        # ustawienie maksymalnej ceny z inf. na max_price
        query[0]['$match']['price_per_night']['$lt'] = max_price
    if room_type is not None:
        # ustawienie typu pokoju z {'$exists': 1} (warunek istnienia pola,
wyświetli każdy dokument) na room_type
        query[0]['$match']['room_type'] = room_type
    if hotel_city is not None:
        # ustawienie miasta z {'$exists': 1} (warunek istnienia pola, wyświetli
każdy dokument) na hotel_city
        query[4]['$match']['hotel_city'] = hotel_city

    result = mongo.rooms.aggregate(query)
    return list(result)
```

Filtrowanie listy dostepnych pokoi

Trigger sprzątający nieaktualne rezerwacje z kolekcji Rooms

W Atlasie stworzyliśmy trigger, który usuwa przeszłe bookingi z kolekcji Rooms, w celu optymalizacji bazy danych (tablice te urosłyby szybko do ogromnych rozmiarów). Jego kod wraz z komentarzami opisującymi działanie:

```
exports = async function() {
  // Pobranie aktualnej daty i czasu
  const currentDate = new Date();

  // Pobranie kolekcji Rooms i Booking_Logs
  const collectionRooms =
context.services.get("HotelsCluster").db("HotelsDB").collection("Rooms");
  const collectionBookingLogs =
context.services.get("HotelsCluster").db("HotelsDB").collection("Booking_Logs");

  try {
    // Warunek wyszukiwania rezerwacji do usunięcia
    const filter = { "bookings.date_to": { $lt: currentDate } };

    // Projekcja dla wyszukiwania rezerwacji do usunięcia
    const projection = { bookings: { $elemMatch: { date_to: { $lt: currentDate } }
} };

    // Wyszukanie i zapisanie rezerwacji do usunięcia
    const bookingsToRemove = await collectionRooms.find(filter,
projection).toArray();

    if (bookingsToRemove.length > 0) {
      // Przygotowanie operacji zbiorczych do usunięcia rezerwacji
      const bulkOps = bookingsToRemove.map(booking => ({
        updateOne: {
          filter: { _id: booking._id },
          update: { $pull: { bookings: { date_to: { $lt: currentDate } } } }
        }
```

```
        }));

        // Usunięcie rezerwacji z kolekcji Rooms
        await collectionRooms.bulkWrite(bulkOps);

        // Przygotowanie danych rezerwacji do zapisania w kolekcji Booking_Logs
        const bookingLogs = bookingsToRemove.flatMap(booking =>
booking.bookings.map(bookingData => ({
          room_id: booking._id,
          booking_id: bookingData.booking_id,
          customer_id: bookingData.customer_id,
          date_from: bookingData.date_from,
          date_to: bookingData.date_to
        })));

        // Zapisanie rezerwacji w kolekcji Booking_Logs
        await collectionBookingLogs.insertMany(bookingLogs);

        // Wyświetlenie informacji o liczbie przeniesionych rezerwacji
        console.log(`Moved ${bookingsToRemove.length} bookings to BookingLogs
collection.`);
      } else {
        // Wyświetlenie informacji o braku rezerwacji do przeniesienia
        console.log("No bookings to move.");
      }
    } catch (err) {
      // Obsługa błędu
      console.error(err);
    }
};
```

Schema validators dla naszego schematu

Hotels

```
{
    "$jsonSchema": {
        "bsonType": "object",
        "required": ["name", "street", "city", "zip_code", "imgUrl"],
        "properties": {
            "name": {
                "bsonType": "string"
            },
            "street": {
                "bsonType": "string"
            },
            "city": {
                "bsonType": "string"
            },
            "zip_code": {
```

```
                "bsonType": "string",
                "description": "string consisting of 5 digit without any
separators"
            },
            "imgUrl": {
                "bsonType": "string"
            }
        }
    }
}
```

Rooms

```
{
    "$jsonSchema": {
        "bsonType": "object",
        "required": ["hotel_id", "room_type", "room_number", "price_per_night",
"is_available", "imgUrl"],
        "properties": {
            "hotel_id": {
                "bsonType": "objectId"
            },
            "room_type": {
                "bsonType": "int"
            },
            "room_number": {
                "bsonType": "int"
            },
            "price_per_night": {
                "bsonType": "double",
                "minimum": 0.0,
                "exclusiveMinimum": True
            },
            "is_available": {
                "bsonType": "bool"
            },
            "imgUrl": {
                "bsonType": "string"
            },
            "bookings": {
                "bsonType": "array",
                "items": {
                    "bsonType": "object",
                    "properties": {
                        "booking_id": {
                            "bsonType": "objectId"
                        },
                        "customer_id": {
                            "bsonType": "objectId"
                        },
                        "date_from": {
```

```
                            "bsonType": "date"
                        },
                        "date_to": {
                            "bsonType": "date"
                        }
                    }
                }
            }
        }
    }
}
```

Customers

```
{
    "$jsonSchema": {
        "bsonType": "object",
        "required": ["name", "surname", "email", "password", "bookings"],
        "properties": {
            "name": {
                "bsonType": "string"
            },
            "surname": {
                "bsonType": "string"
            },
            "email": {
                "bsonType": "string"
            },
            "password": {
                "bsonType": "string"
            },
            "bookings": {
                "bsonType": ["array"],
                "items": {
                    "bsonType": "object",
                    "properties": {
                        "booking_id": {
                            "bsonType": "objectId"
                        },
                        "room_id": {
                            "bsonType": "objectId",
                        },
                        "date_from": {
                            "bsonType": "date"
                        },
                        "date_to": {
                            "bsonType": "date"
                        }
                    }
                }
            }
```

```
        }
    }
}
```

Booking_Logs

```
{
    "$jsonSchema": {
        "bsonType": "object",
        "required": ["booking_id", "customer_id", "room_id", "date_from",
"date_to"],
        "properties": {
            "booking_id": {
                "bsonType": "objectId"
            },
            "customer_id": {
                "bsonType": "objectId"
            },
            "room_id": {
                "bsonType": "objectId"
            },
            "date_from": {
                "bsonType": "date"
            },
            "date_to": {
                "bsonType": "date"
            }
        }
    }
}
```

Widoki

- start_page - strona startowa, którą widzimy po wejściu do aplikacji
- login - widok logowania
- signup - widok tworzenia konta
- rooms_list - lista pokoi (wraz z filtrami), kiedy nie jesteśmy zalogowani
- my_bookings - widok dla zalogowanego użytkownika - wyświetla wszystkie nasze (przyszłe i przeszłe) rezerwacje oraz daje możliwość modyfikacji rezerwacji
- reserve_rooms - widok dla zalogowanego użytkownika - taki sam jak widok rooms_list, ale z opcją rezerwacji pokoju

Autentykacja i autoryzacja użytkownika

Do autoryzacji i autentykacji użytkownika korzystamy z modułu Flask_Login, który bardzo ułatwia sprawę, przy rzeczach typu: sprawdzanie, który użytkownik jest zalogowany, szybki dostęp do jego danych itp.

Poniżej widzimy funkcję odpowiedzialną za rejestrowanie nowego użytkownika. Po odebraniu requesta typu `POST` odczytywane są wszystkie dane przesłane w formularzu i wykonywana jest proste sprawdzanie danych.

```
@auth.route('/sign-up', methods=['GET', 'POST'])
def sign_up():
    if request.method == 'POST':
        # Odczyt danych z formularza
        name = request.form.get('firstName')
        surname = request.form.get('surname')
        email = request.form.get('email')
        password1 = request.form.get('password1')
        password2 = request.form.get('password2')

        # Sprawdzenie jakości danych i walidacja podanych haseł
        email_pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
        result = re.match(email_pattern, email)
        if not result:
            flash('Wrong email format!', category='error')
        elif len(name) == 0:
            flash('You must enter your name!', category='error')
        elif len(surname) == 0:
            flash('You must enter your surname!', category='error')
        elif len(password1) < 5:
            flash('Your password is too short!', category='error')
        elif password1 != password2:
            flash('Passwords do not match!', category='error')
        else:
            # Dodanie użytkownika do bazy daych i zalogowanie go przy pomocy
flask_login
            if add_customer(name, surname, email,
generate_password_hash(password1, method='sha256')):
                flash('Account created successfully!', category='success')
                user = get_user_email(email)
                user = LoggedUser(str(user['_id']), name, surname, email,
password1)
                login_user(user, remember=True)
                return redirect(url_for('views.home'))
            else:
                flash('Creating account failed, this email is already taken!',
category='error')

    return render_template("signup.html", user=current_user)
```

Następnie mamy funkcję odpowiedzialną za logowanie się użytkownika. Podobnie jak w przypadku rejestracji, w przypadku requesta typu POST czytane są odpowiednie dane i dokonywane jest proste sprawdzenie, czy dany użytkownik jest już w naszej bazie danych i zalogowanie użytkownika z użyciem `login_user()`.

```
@auth.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # Odczyt danych z przesłanego formularza
        email = request.form.get('email')
        password = request.form.get('password')

        # Sprawdzenie, czy taki użytkownik istnieje w naszej bazie danych i
sprawdzenie czy hasło się zgadza
        user = get_user_email(email)
        if user is None:
            flash('There is no user with this email address.', category='error')
        elif check_password_hash(user['password'], password):
            user = LoggedUser(str(user['_id']), user['name'], user['surname'],
user['email'], user['password'], user['bookings'])
            # Zalogowanie użytkownika przy pomocy flask_login
            login_user(user, remember=True)
            flash("Logged in!", category='success')
            return redirect(url_for('views.home'))
        else:
            flash('Incorrect password.', category='error')
    return render_template("login.html", user=current_user)
```

Ostatnią częścią związaną z tym punktem jest wylogowyanie użytkownika. Ponownie jak w poprzednich funkcja z wielką pomocą przychodzi nam `flask_login` - w tym przypadku funkcja `logout_user()`.

```
@auth.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for('auth.login'))
```

Generowanie widoków

Generowanie strony startowej - na stronie startowej naszej aplikacji widnieje lista hoteli, z którymi "współpracujemy", a także oczywiście odpowiedni navbar i footer.
Funkcja generująca stronę:

```
@views.route('/')
def home():
    hotels = get_all_hotels()
    return render_template("start_page.html", user=current_user, hotels=hotels)
```

Fragment kodu umożliwiający nam łatwe wygenerowanie kart hoteli korzystającego z Jinja2:

```
<h1>We work with following hotels:</h1>
<div class="hotels-wrapper" id="hotels">
    <-- Fragment Jinji2 -->
    {% for hotel in hotels %}
    <div class="hotel-card">
        <img src={{ hotel['imgUrl'] }}>
        <h3 class="hotel-name">Name: {{ hotel['name'] }}</h3>
        <h3>Street: {{ hotel['street'] }}</h3>
        <h3>City: {{ hotel['city'] }}</h3>
    </div>
    {% endfor %}
</div>
```

Generowanie listy pokoi w momencie gdy użytkownik jest zalogowany - w tym widoku poza listą pokojów, które można zarezerwować i formularzem rezerwacji mamy dostęp do filtrów - możemy filtrować po następujących kategoriach:

- max/min price
- checkin/checkout date
- number of people in one room
- city

W tym przypadku, musieliśmy rozróżnić dwa typy `POST` - jeden odpowiedzialny za filtry, a drugi za rezerwację pokoju. Rozróżniamy je na podstawie informacji przychodzących wraz z requestem.

```
@views.route('/reserve_rooms', methods=['GET', 'POST'])
@login_required
def reserve_list():
    cities = get_all_cities()
    rooms = filter_rooms()
    if request.method == 'POST' and request.form.get('checkin') is not None:
        date_format = "%Y-%m-%d"
        check_in = request.form.get('checkin')
        check_in = datetime.strptime(check_in, date_format)

        check_out = request.form.get('checkout')
        check_out = datetime.strptime(check_out, date_format)

        curr_date = datetime.now().date()
        curr_date = datetime.combine(curr_date, datetime.min.time())
        if check_in < curr_date:
            flash('Check in date must be greater than or equals current date.',
category='error')
        elif check_out < check_in:
            flash('Check in date must be less or equal than check out date.',
category='error')
        else:
            room_id = request.form.get('room_id')
```

```
            customer_id = request.form.get('customer_id')

            if add_new_booking(customer_id, room_id, check_in, check_out):
                flash('Room booked successfully!', category='success')
            else:
                flash('Room is already booked in this period of time.',
category='error')
    elif request.method == 'POST':
        # Odczytanie danych z formularza dotyczącego filtrów
        min_price = request.form.get('min_price')
        max_price = request.form.get('max_price')
        check_in = request.form.get('checkin-filter')
        check_out = request.form.get('checkout-filter')
        people = request.form.get('people')
        city = request.form.get('city')

        # Obsługa danych - zamiana na odpowiedni typ danych
        date_format = "%Y-%m-%d"
        check_in = datetime.strptime(check_in, date_format) if check_in != '' else
None
        check_out = datetime.strptime(check_out, date_format) if check_out != ''
else None
        min_price = float(min_price) if min_price != '' else None
        max_price = float(max_price) if max_price != '' else None
        people = int(people) if people != '' else None

        if city == 'select':
            city = None

        curr_date = datetime.now().date()
        curr_date = datetime.combine(curr_date, datetime.min.time())

        # Proste sprawdzanie "jakości" dat
        if (check_in is not None and check_out is not None) and check_out <
check_in:
            flash('Check in date must be less or equal than check out date.',
category='error')
        elif check_in is not None and check_in < curr_date:
            flash('Check in date must be greater or equal to current date.',
category='error')
        else:
            # Wywołanie funkcji filtrującej pokoje
            rooms = filter_rooms(check_in, check_out, min_price, max_price,
people, city)

    return render_template("reserve_rooms.html", user=current_user, rooms=rooms,
cities=cities)
```

Tak jak w poprzednim przypadku Jinja2 bardzo ułatwia nam generowanie listy przefiltorwanych pokoi:

```
<div class="rooms-wrapper">
    {% for room in rooms %}
    <div class="room-card">
        <img src={{ room['room_imgUrl'] }}>
        <h3>Hotel: {{ room['hotel_name'] }}</h3>
        <h3>City: {{ room['hotel_city'] }}</h3>
        <h3>Street: {{ room['hotel_street'] }}</h3>
        <h3>People in room: {{ room['room_type'] }}</h3>
        <h3>Price per night: {{ room['price_per_night'] }} zł</h3>
        <form method="POST" class="date-form">
            <div class="form-group">
                <label for="checkin">Check in date:</label>
                <input type="date" id="checkin" name="checkin" required>
            </div>
            <div class="form-group">
                <label for="checkout">Check out date:</label>
                <input type="date" id="checkout" name="checkout" required>
            </div>
            <input type="hidden" name="room_id" value={{ room['room_id'] }}>
            <input type="hidden" name="customer_id" value={{ current_user._id }}>
            <div class="btn-wrap">
                <button type="submit" class="reserve-btn">Book</button>
            </div>
        </form>
    </div>
    {% endfor %}
</div>
```

Generowanie rezerwacji użtykownika - umożliwia on zarządzanie naszymi rezerwacjami - rezygnację lub zmianę daty danej rezerwacji.

```
@views.route('/bookings', methods=['GET', 'POST'])
@login_required
def my_bookings():
    # Kod obsługujący zmianę daty rezerwacji
    if request.method == 'POST' and request.form.get('new_checkin') is not None:
        booking_id = request.form.get('booking_id')
        customer_id = current_user._id
        room_id = request.form.get('room_id')

        date_format = "%Y-%m-%d"
        new_checkin = request.form.get('new_checkin')
        new_checkin = datetime.strptime(new_checkin, date_format)

        new_checkout = request.form.get('new_checkout')
        new_checkout = datetime.strptime(new_checkout, date_format)

        curr_date = datetime.now().date()
        curr_date = datetime.combine(curr_date, datetime.min.time())
```

```
        if new_checkin < curr_date:
            flash('Check in date must be greater than or equals current date.',
category='error')
        elif new_checkout < new_checkin:
            flash('Check in date must be less or equal than check out date.',
category='error')
        else:
            if change_booking(customer_id, room_id, booking_id, new_checkin,
new_checkout):
                flash('Room booked successfully!', category='success')
            else:
                flash('Room is already booked in this period of time.',
category='error')

    user_bookings = get_all_user_bookings(current_user._id)   # str
    return render_template("my_bookings.html", user=current_user,
bookings=user_bookings)
```

Z rezygnacją z rezerwacji przychodzi nam z pomocą JavaScript:

```
function removeBooking(booking_id, room_id) {
    fetch('/remove-booking', {
        method: 'POST',
        body: JSON.stringify({booking_id: booking_id, room_id: room_id})
    }).then((_res) => {
        window.location.href = '/bookings'
    });
}
```

A z generowaniem listy ponownie pomagam nam Jinja2:

```
{% for booking in bookings %}
<div class="room-card">
    <img src={{ booking['room_imgUrl'] }}>
    <h3>Hotel: {{ booking['hotel_name'] }}</h3>
    <h3>Address: {{ booking['hotel_city'] }}, {{ booking['hotel_address'] }} </h3>
    <h3>Room type: {{ booking['room_type'] }} people</h3>
    <h3>Price per night: {{ booking['price_per_night'] }}zł</h3>
    <h3>Check in date: {{ booking['date_from'] }}</h3>
    <h3>Check out date: {{ booking['date_to'] }}</h3>

    {% if booking['can_be_edited'] %}
        <div class="btn-wrap">
            <button class="reserve-btn" onClick="fillAndShowTheForm('{{
booking['hotel_name'] }}', '{{ booking['date_from'] }}', '{{ booking['date_to']
}}', '{{ booking['booking_id'] }}', '{{ current_user._id }}', '{{
booking['room_id'] }}')">
                Change booking
            </button>
```

```
            <form method="POST">
                <button type="submit" class="remove-booking"
onClick="removeBooking('{{ booking['booking_id'] }}', '{{ booking['room_id']
}}')">Resign</button>
            </form>
        </div>
    {% endif %}
</div>
{% endfor %}
```

Z zachowaniem spójności we wszystkich widokach pomaga nam system dziedziczenia w Jinja2 - każdy widok dziedziczy po widoku bazowym w ten sam sposób:

```
{% extends "start_base.html" %}
{% block content %}
    ...
{% endblock %}
```

A sam widok bazowy wygląda w następujący sposób:

```
<nav class="navbar">
<!-- Generowanie zawartości menu na podstawie tego czy użytkownik jest zalogowany
-->
    {% if user.is_authenticated %}
    <div class="nav-elem"><a id="home" href="/">Home</a></div>
    <div class="nav-elem"><a id="rooms" href="/reserve_rooms">Rooms selection</a>
</div>
    <div class="nav-elem"><a id="bookings" href="/bookings">My bookings</a></div>
    <div class="nav-elem"><a id="logout" href="/logout">Logout</a></div>
    {% else %}
    <div class="nav-elem"><a id="home" href="/">Home</a></div>
    <div class="nav-elem"><a id="rooms" href="/rooms">Rooms selection</a></div>
    <div class="nav-elem"><a id="login" href="/login">Login</a></div>
    <div class="nav-elem"><a id="signUp" href="/sign-up">Sign-Up</a></div>
    {% endif %}
</nav>

<!-- Wypisywanie ewentualnych błądów lub różnych informacji po wykonaniu jakiejś
akcji na stronie np. zalogowaniu się -->
{% with messages = get_flashed_messages(with_categories=true) %}
    {% if messages %}
        {% for category, message in messages %}
            {% if category == 'error' %}
                <div class="alert error" role="alert">
                    {{ message }}
                </div>
            {% else %}
                <div class="alert success" role="alert">
```

```
                    {{ message }}
                </div>
            {% endif %}
        {% endfor %}
    {% endif %}
{% endwith %}

<!-- Miejsce, w którym generować się będzie nasze potomne widoki -->
{% block content %} {% endblock %}
<footer>
    <ul class="footer-items">
        <li class="footer-item title">OurName</li>
        <li class="footer-item"><a href="mailto:email@gmail.com"><i class="fa-
solid fa-envelope"></i>email@gmail.com</a></li>
        <li class="footer-item"><a href="tel:555-0179"><i class="fa-solid fa-
mobile-button"></i>Phone-Number: 123-456-789</a></li>
    </ul>
</footer>
```