Skład grupy

- Nikodem Adamczyk, nikodemadm@student.agh.edu.pl
- Stanisław Barycki, barycki@student.agh.edu.pl

Temat projektu

Aplikacja bazodanowa obsługująca wynajem krótkoterminowy w jednym budynku mieszkalnym. Głównym jej założeniem jest ułatwienie i usprawnienie zarządzania budynkiem, zapewniając efektywność i łatwość obsługi zarówno dla administratorów, jak i dla klientów.

Informację o wykorzystywanym SZBD i technologii realizacji projektu

- MySql
- Node.js i Express.js
- React Native

Link do repozytorium

https://github.com/Jeremylaby/MINIPROJEKT_BARYCKI_ADAMCZYK

Aktorzy i ich Uprawnienia

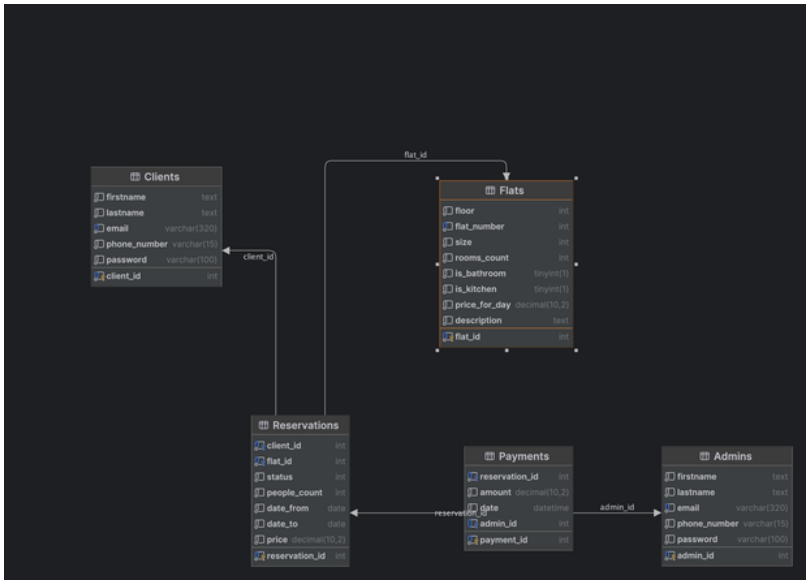
Aktorzy:

1. **Admin** - Administrator systemu, ma dostęp do zarządzania danymi klientów, rezerwacji oraz płatności.
2. **Klient** - Użytkownik systemu, który może dokonywać rezerwacji oraz przeglądać swoje płatności.

Uprawnienia:

- **User**
  - Wyświetlanie Szczegółów mieszkania
  - Wyświetlanie dostępnych mieszkań w danym przedziale czasowym
  - Wyświetlanie mieszkań po tym ile mają pokoi i czy kuchnia łazienka
  - Rezerwowanie mieszkania
  - Dokonywanie płatności za rezerwacje
  - Anulowanie rezerwacji
- **Admin**
  - Wyświetlanie anulowanych rezerwacji
  - Widok raportu finansowego z danego miesiąca
  - Wyświetlanie wszystkich niezwróconych rezerwacji
  - Wyświetlanie mieszkań których rezerwacja kończy się w danym dniu (do sprzątania)
  - Wyświetlanie zwróconych rezerwacji po id Admina
  - Wyświetlanie rezerwacji zwróconych przez system

Schemat bazy danych



Struktura Tabel

Admins

Kolumna	Typ	Opcje	Opis
admin_id	int	auto_increment, primary key	Unikalny identyfikator administratora
firstname	text	not null	Imię administratora
lastname	text	not null	Nazwisko administratora

Kolumna	Typ	Opcje	Opis
email	varchar(320)	not null, unique	Adres email administratora
phone_number	varchar(15)	not null	Numer telefonu administratora
password	varchar(100)	not null	Hasło administratora

```
create table Admins
(
    admin_id     int auto_increment
        primary key,
    firstname    text         not null,
    lastname     text         not null,
    email        varchar(320) not null,
    phone_number varchar(15)  not null,
    password     varchar(100) not null,
    constraint Admins__email
        unique (email),
    constraint Admins_ak_1
        unique (email)
);
```

Flats

Kolumna	Typ	Opcje	Opis
flat_id	int	auto_increment, primary key	Unikalny identyfikator mieszkania
floor	int	not null	Piętro, na którym znajduje się mieszkanie
flat_number	int	not null, unique	Numer mieszkania
size	int	not null	Wielkość mieszkania w metrach kwadratowych
rooms_count	int	not null	Liczba pokoi
is_bathroom	tinyint(1)	not null	Czy mieszkanie ma łazienkę (1 - tak, 0 - nie)
is_kitchen	tinyint(1)	not null	Czy mieszkanie ma kuchnię (1 - tak, 0 - nie)
price_for_day	decimal(10, 2)	not null	Cena za dzień wynajmu
description	text	not null	Opis mieszkania

```
create table Flats
(
    flat_id       int auto_increment
        primary key,
    floor         int            not null,
    flat_number   int            not null,
    size          int            not null,
    rooms_count   int            not null,
    is_bathroom   tinyint(1)     not null,
    is_kitchen    tinyint(1)     not null,
    price_for_day decimal(10, 2) not null,
    description   text           not null,
    constraint Flats_ak_1
        unique (flat_number)
);
```

Clients

Kolumna	Typ	Opcje	Opis
client_id	int	auto_increment, primary key	Unikalny identyfikator klienta
firstname	text	not null	Imię klienta
lastname	text	not null	Nazwisko klienta
email	varchar(320)	not null, unique	Adres email klienta
phone_number	varchar(15)	not null	Numer telefonu klienta
password	varchar(100)	not null	Hasło klienta

```
create table Clients
(
    client_id    int auto_increment
        primary key,
    firstname    text         not null,
    lastname     text         not null,
    email        varchar(320) not null,
    phone_number varchar(15)  not null,
    password     varchar(100) not null,
    constraint Clients__email
        unique (email),
    constraint Clients_ak_1
        unique (email)
);
```

Payments

Kolumna	Typ	Opcje	Opis
payment_id	int	auto_increment, primary key	Unikalny identyfikator płatności

Kolumna	Typ	Opcje	Opis
reservation_id	int	not null, foreign key (Reservations.reservation_id)	Identyfikator rezerwacji związanej z płatnością
amount	decimal(10, 2)	not null	Kwota płatności
date	datetime	not null	Data dokonania płatności
admin_id	int	foreign key (Admins.admin_id)	Identyfikator administratora, który zatwierdził płatność (opcjonalne)

```
create table Payments
(
    payment_id     int auto_increment
        primary key,
    reservation_id int            not null,
    amount         decimal(10, 2) not null,
    date           datetime       not null,
    admin_id       int            null,
    constraint Payments_Admins
        foreign key (admin_id) references Admins (admin_id),
    constraint Payments_Reservations
        foreign key (reservation_id) references Reservations (reservation_id)
);

create index Payments_ak_1
    on Payments (reservation_id);
```

Reservations

Kolumna	Typ	Opcje	Opis
reservation_id	int	auto_increment, primary key	Unikalny identyfikator rezerwacji
client_id	int	not null, foreign key (Clients.client_id)	Identyfikator klienta
flat_id	int	not null, foreign key (Flats.flat_id)	Identyfikator mieszkania
status	int	not null	Status rezerwacji (np. 0 - anulowana, 1 - aktywna)
people_count	int	not null	Liczba osób
date_from	date	not null	Data rozpoczęcia rezerwacji
date_to	date	not null	Data zakończenia rezerwacji
price	decimal(10, 2)	not null	Całkowity koszt rezerwacji

```
create table Reservations
(
    reservation_id int auto_increment
        primary key,
    client_id      int            not null,
    flat_id        int            not null,
    status         int            not null,
    people_count   int            not null,
    date_from      date           not null,
    date_to        date           not null,
    price          decimal(10, 2) not null,
    constraint Reservations_Clients
        foreign key (client_id) references Clients (client_id),
    constraint Reservations_Flats
        foreign key (flat_id) references Flats (flat_id)
);
```

Widoki

ActiveReservations

Opis: Widok zawierający aktywne rezerwacje.

```
create definer = root@`%` view ActiveReservations as
select `MiniDb`.`Reservations`.`reservation_id` AS `reservation_id`,
       `MiniDb`.`Reservations`.`client_id`      AS `client_id`,
       `MiniDb`.`Reservations`.`flat_id`        AS `flat_id`,
       `MiniDb`.`Reservations`.`status`         AS `status`,
       `MiniDb`.`Reservations`.`people_count`   AS `people_count`,
       `MiniDb`.`Reservations`.`date_from`      AS `date_from`,
       `MiniDb`.`Reservations`.`date_to`        AS `date_to`,
       `MiniDb`.`Reservations`.`price`          AS `price`
from `MiniDb`.`Reservations`
where (((`MiniDb`.`Reservations`.`status` = 1) and (`MiniDb`.`Reservations`.`date_from` >= curdate())) or
       (`MiniDb`.`Reservations`.`date_to` >= curdate()));
```

AllCancelledReservations

Opis: Widok zawierający wszystkie anulowane rezerwacje.

```
create definer = root@`%` view AllCancelledReservations as
select `MiniDb`.`Reservations`.`reservation_id` AS `reservation_id`,
       `MiniDb`.`Reservations`.`client_id`      AS `client_id`,
       `MiniDb`.`Reservations`.`date_from`      AS `date_from`,
       `MiniDb`.`Reservations`.`date_to`        AS `date_to`,
       `MiniDb`.`Reservations`.`status`         AS `status`
from `MiniDb`.`Reservations`
where (`MiniDb`.`Reservations`.`status` = 0);
```

AllFlats

Opis: Widok zawierający wszystkie mieszkania.

```
create definer = root@`%` view AllFlats as
select `MiniDb`.`Flats`.`flat_id`       AS `flat_id`,
       `MiniDb`.`Flats`.`floor`         AS `floor`,
       `MiniDb`.`Flats`.`flat_number`   AS `flat_number`,
       `MiniDb`.`Flats`.`size`          AS `size`,
       `MiniDb`.`Flats`.`rooms_count`   AS `rooms_count`,
       `MiniDb`.`Flats`.`is_bathroom`   AS `is_bathroom`,
       `MiniDb`.`Flats`.`is_kitchen`    AS `is_kitchen`,
       `MiniDb`.`Flats`.`price_for_day` AS `price_for_day`,
       `MiniDb`.`Flats`.`description`   AS `description`
from `MiniDb`.`Flats`;
```

CurrentlyOccupiedRooms

Opis: Widok zawierający obecnie zajęte mieszkania.

```
create definer = root@`%` view CurrentlyOccupiedRooms as
select `f`.`flat_id`     AS `flat_id`,
       `f`.`flat_number` AS `flat_number`,
       `r`.`client_id`   AS `client_id`,
       `r`.`date_from`   AS `date_from`,
       `r`.`date_to`     AS `date_to`
from (`MiniDb`.`Flats` `f` join `MiniDb`.`Reservations` `r` on ((`f`.`flat_id` = `r`.`flat_id`)))
where ((`r`.`date_from` <= curdate()) and (`r`.`date_to` >= curdate()) and (`r`.`status` = 1));
```

ReservationsToRefund

Opis: Widok zawierający rezerwacje do zwrotu.

```
create definer = root@`%` view ReservationsToRefund as
select `r`.`reservation_id` AS `reservation_id`,
       `r`.`client_id`      AS `client_id`,
       `r`.`date_from`      AS `date_from`,
       `r`.`date_to`        AS `date_to`,
       (case
            when ((`r`.`status` = 0) and (coalesce(sum(`p`.`amount`), 0) <> 0)) then coalesce(sum(`p`.`amount`), 0)
            when (`r`.`status` = 1) then coalesce((sum(`p`.`amount`) - `r`.`price`), 0)
            else 0 end)     AS `to_return`,
       `r`.`status`         AS `status`
from (`MiniDb`.`Reservations` `r` left join `MiniDb`.`Payments` `p` on ((`r`.`reservation_id` = `p`.`reservation_id`)))
group by `r`.`reservation_id`, `r`.`client_id`, `r`.`date_from`, `r`.`date_to`, `r`.`status`
having (`to_return` > 0);
```

Funkcje

GetClientId

Opis: Pobiera identyfikator klienta na podstawie adresu email.

Parametry:

- `in_email` (varchar(320)): Adres email klienta.

Działanie: Pobiera identyfikator klienta lub zwraca błąd, jeśli klient nie został znaleziony.

```
create
    definer = root@`%` procedure GetClientId(IN in_email varchar(320))
BEGIN
    DECLARE clientID INT;
    SELECT client_id INTO clientID FROM Clients WHERE email=in_email;
    IF clientID is null THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Client not found';
    ELSE
        SELECT clientID AS client_id;
    END IF;
end;

grant execute on procedure GetClientId to client;
```

Procedury

AddReservation

Opis: Dodaje nową rezerwację.

Parametry:

- `resClientId` (int): Identyfikator klienta.
- `resFlatId` (int): Identyfikator mieszkania.
- `resDateFrom` (date): Data rozpoczęcia rezerwacji.
- `resDateTo` (date): Data zakończenia rezerwacji.
- `resPeopleCount` (int): Liczba osób.

Działanie: Sprawdza dostępność mieszkania, a następnie dodaje nową rezerwację do bazy danych.

```
create
    definer = root@`%` procedure AddReservation(IN resClientId int, IN resFlatId int, IN resDateFrom date, IN resDateTo date, IN resPeopleCount int)
BEGIN
    DECLARE roomCapacity INT;
    DECLARE roomPricePerDay DECIMAL(10,2);
    DECLARE nights INT;

    SELECT size, price_for_day INTO roomCapacity, roomPricePerDay FROM Flats WHERE flat_id = resFlatId;
    IF resPeopleCount > roomCapacity THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'The number of people exceeds the room capacity.';
    END IF;
    CALL isRoomAvailable(resFlatId,resDateFrom,resDateTo);
    SET nights = DATEDIFF(resDateTo, resDateFrom);
    INSERT INTO Reservations (client_id, flat_id, date_from, date_to, people_count, status, price)
    VALUES (resClientId, resFlatId, resDateFrom, resDateTo, resPeopleCount, 1, roomPricePerDay * nights);

END;

grant execute on procedure AddReservation to client;
```

CancelFutureReservation

Opis: Anuluje przyszłą rezerwację.

Parametry:

- `clientID` (int): Identyfikator klienta.
- `reservationID` (int): Identyfikator rezerwacji.

Działanie: Sprawdza, czy rezerwacja należy do klienta i czy jest w przyszłości, a następnie ją anuluje.

```
create
    definer = root@`%` procedure CancelFutureReservation(IN clientID int, IN reservationID int)
BEGIN
    DECLARE startDate DATE;

    SELECT date_from INTO startDate FROM Reservations WHERE reservation_id = reservationID;
    CALL CheckReservationOwner(clientID,reservationID);
    IF startDate > CURDATE() THEN
        UPDATE Reservations SET status = 0 WHERE reservation_id = reservationID;
        SELECT 'Reservation cancelled successfully.' AS message;
    ELSE
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot cancel past or current reservations.';
    END IF;
END;

grant execute on procedure CancelFutureReservation to client;
```

CheckReservationOwner

Opis: Sprawdza, czy rezerwacja należy do danego klienta.

Parametry:

- `in_clientId` (int): Identyfikator klienta.
- `in_reservationId` (int): Identyfikator rezerwacji.

Działanie: Sprawdza, czy rezerwacja należy do klienta.

```
create
    definer = root@`%` procedure CheckReservationOwner(IN in_clientId int, IN in_reservationId int)
BEGIN
    DECLARE actualClientId INT;
    SELECT client_id INTO actualClientId FROM Reservations WHERE reservation_id = in_reservationId;
    IF actualClientId != in_clientId THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'This reservation does not belong to the specified client.';
    END IF;

    SELECT 'Reservation ownership verified successfully.' AS message;
END;
```

FindAvailableFlats

Opis: Znajduje dostępne mieszkania w określonym okresie.

Parametry:

- `startDate` (date): Data rozpoczęcia.
- `endDate` (date): Data zakończenia.
- `numPeople` (int): Liczba osób.
- `withKitchen` (tinyint): Czy mieszkanie ma kuchnię (1 - tak, 0 - nie).

```
create
    definer = root@`%` procedure FindAvailableFlats(IN startDate date, IN endDate date, IN numPeople int, IN withKitchen tinyint(1))
BEGIN

    SELECT f.flat_id, f.flat_number, f.size, f.is_kitchen,f.description,f.price_for_day
    FROM Flats f
    WHERE f.size >= numPeople AND (
          (withKitchen = 1 AND f.is_kitchen = 1)
          OR
          (withKitchen = 0)
```

```
        )
          AND f.flat_id NOT IN (
              SELECT r.flat_id
              FROM ActiveReservations r
              WHERE (r.date_from < endDate AND r.date_to > startDate)
          );
END;

grant execute on procedure FindAvailableFlats to client;
```

GetRoomsToClean

Opis: Pobiera listę mieszkań do posprzątania na określony dzień.

Parametry:

- `target_date` (date): Docelowa data.

```
create
    definer = root@`%` procedure GetRoomsToClean(IN target_date date)
BEGIN
    SELECT
        f.flat_number,
        r.flat_id,
        r.date_to,
        r.reservation_id
    FROM
        Flats f
    JOIN
        Reservations r ON f.flat_id = r.flat_id
    WHERE

        r.date_to = target_date
        AND r.status = 1;
END;
```

GetRoomsToCleanToday

Opis: Pobiera listę mieszkań do posprzątania na dzisiejszy dzień.

```
create
    definer = root@`%` procedure GetRoomsToCleanToday()
BEGIN
    CALL GetRoomsToClean(curdate());
end;
```

MonthlyFinancialReport

Opis: Generuje miesięczny raport finansowy.

Parametry:

- `reportYear` (int): Rok raportu.
- `reportMonth` (int): Miesiąc raportu.

```
create
    definer = root@`%` procedure MonthlyFinancialReport(IN reportYear int, IN reportMonth int)
BEGIN
    IF EXISTS (
        SELECT 1
        FROM ReservationsToRefund r
        WHERE YEAR(r.date_to) = reportYear AND MONTH(r.date_to) = reportMonth
    ) THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot generate report: pending refunds exist for this month.';
    ELSE
        SELECT
            r.reservation_id,
            r.client_id,
            r.date_from,
            r.date_to,
            r.status,
            r.price,
            COALESCE((SELECT SUM(p.amount) FROM Payments p WHERE p.reservation_id = r.reservation_id), 0) AS balance
        FROM
            Reservations r
        WHERE
            YEAR(r.date_to) = reportYear AND
            MONTH(r.date_to) = reportMonth;
    END IF;
END;
```

MonthlyFinancialReportPreviousMonth

Opis: Generuje miesięczny raport finansowy za poprzedni miesiąc.

```
create
    definer = root@`%` procedure MonthlyFinancialReportPreviousMonth()
BEGIN
    DECLARE reportYear INT;
    DECLARE reportMonth INT;

    SET reportYear = YEAR(CURRENT_DATE - INTERVAL 1 MONTH);
```

```
    SET reportMonth = MONTH(CURRENT_DATE - INTERVAL 1 MONTH);
    CALL MonthlyFinancialReport(reportYear,reportMonth);
END;
```

RefundReservation

Opis: Zwrot rezerwacji.

Parametry:

- `adminId` (int): Identyfikator administratora.
- `reservationId` (int): Identyfikator rezerwacji.
- `amountToRefund` (decimal(10, 2)): Kwota do zwrotu.

```
create
    definer = root@`%` procedure RefundReservation(IN adminId int, IN reservationId int, IN amountToRefund decimal(10, 2))
BEGIN
    DECLARE toReturn DECIMAL(10,2);
    SELECT to_return INTO toReturn
    FROM ReservationsToRefund
    WHERE reservation_id = reservationId;
    IF toReturn IS NULL THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Brak możliwości zwrotu: rezerwacja nie jest anulowana lub została już zwrócona.';
    ELSE
        IF amountToRefund > toReturn THEN
            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Próba zwrotu większej kwoty niż pierwotnie zapłacono.';
        ELSE
            INSERT INTO Payments (reservation_id, amount, date, admin_id)
            VALUES (reservationId, -amountToRefund, NOW(), adminId);
        END IF;
    END IF;
END;
```

ShowCancelledReservations

Opis: Wyświetla anulowane rezerwacje w określonym miesiącu i roku.

Parametry:

- `input_year` (int): Rok.
- `input_month` (int): Miesiąc.

```
create
    definer = root@`%` procedure ShowCancelledReservations(IN input_year int, IN input_month int)
BEGIN
    SELECT * FROM AllCancelledReservations
    WHERE
        YEAR(date_from) = input_year AND
        MONTH(date_from) = input_month;
END;
```

ShowPaymentHistory

Opis: Wyświetla historię płatności klienta.

Parametry:

- `clientID` (int): Identyfikator klienta.

Działanie: Wyświetla historię płatności dla danego klienta, włączając szczegóły rezerwacji.

```
create
    definer = root@`%` procedure ShowPaymentHistory(IN clientID int)
BEGIN
    SELECT
        p.payment_id,
        p.reservation_id,
        p.amount,
        p.date,
        r.client_id,
        r.price AS reservation_cost,
        CASE
            WHEN r.status = 0 THEN -1 * IFNULL((SELECT SUM(amount) FROM Payments WHERE reservation_id = p.reservation_id), 0)
            ELSE IFNULL((SELECT SUM(amount) FROM Payments WHERE reservation_id = p.reservation_id), 0)
        END AS balance,
        r.date_from,
        r.date_to,
        r.status
    FROM
        Payments p
    JOIN
        Reservations r ON p.reservation_id = r.reservation_id
    WHERE
        r.client_id = clientID;
END;

grant execute on procedure ShowPaymentHistory to client;
```

ShowReturnedPaymentsByAdmin

Opis: Wyświetla płatności zwrócone przez danego administratora.

Parametry:

- `adminId` (int): Identyfikator administratora.

Działanie: Wyświetla listę płatności zwróconych przez danego administratora.

```
create
    definer = root@`%` procedure ShowReturnedPaymentsByAdmin(IN adminId int)
BEGIN
    SELECT p.reservation_id,p.amount,p.amount,p.date FROM Payments p WHERE p.admin_id=adminId ORDER BY p.date;
end;
```

UpdateReservationDateFrom

Opis: Aktualizuje datę rozpoczęcia rezerwacji.

Parametry:

- `reservationId` (int): Identyfikator rezerwacji.
- `clientId` (int): Identyfikator klienta.
- `newDateFrom` (date): Nowa data rozpoczęcia rezerwacji.

Działanie: Aktualizuje datę rozpoczęcia rezerwacji.

```
create
    definer = root@`%` procedure UpdateReservationDateFrom(IN reservationId int, IN clientId int, IN newDateFrom date)
BEGIN
    DECLARE dateTo DATE;
    SELECT date_to INTO dateTo FROM Reservations WHERE reservation_id=reservationId;
    CALL UpdateReservationDate(reservationId,clientId, newDateFrom,DateTo);
end;

grant execute on procedure UpdateReservationDateFrom to client;
```

UpdateReservationDateTo

Opis: Aktualizuje datę zakończenia rezerwacji.

Parametry:

- `reservationId` (int): Identyfikator rezerwacji.
- `clientId` (int): Identyfikator klienta.
- `newDateTo` (date): Nowa data zakończenia rezerwacji.

Działanie: Aktualizuje datę zakończenia rezerwacji.

```
create
    definer = root@`%` procedure UpdateReservationDateTo(IN reservationId int, IN clientId int, IN newDateTo date)
BEGIN
    DECLARE dateFrom DATE;
    SELECT date_from INTO dateFrom FROM Reservations WHERE reservation_id=reservationId;
    CALL UpdateReservationDate(reservationId,clientId, dateFrom,newDateTo);
end;

grant execute on procedure UpdateReservationDateTo to client;
```

addAdmin

Opis: Dodaje nowego administratora.

Parametry:

- `in_firstname` (text): Imię administratora.
- `in_lastname` (text): Nazwisko administratora.
- `in_email` (varchar(320)): Adres email administratora.
- `in_phone_number` (varchar(15)): Numer telefonu administratora.
- `in_password` (varchar(100)): Hasło administratora.

Działanie: Dodaje nowego administratora do bazy danych.

```
create
    definer = root@`%` procedure addAdmin(IN in_firstname text, IN in_lastname text, IN in_email varchar(320),
IN in_phone_number varchar(15), IN in_password varchar(100))
BEGIN
    INSERT INTO Admins (firstname, lastname, email, phone_number, password)
    VALUES (in_firstname, in_lastname, in_email, in_phone_number, in_password);
END;
```

addClient

Opis: Dodaje nowego klienta.

Parametry:

- `in_firstname` (text): Imię klienta.
- `in_lastname` (text): Nazwisko klienta.
- `in_email` (varchar(320)): Adres email klienta.

- `in_phone_number` (varchar(15)): Numer telefonu klienta.
- `in_password` (varchar(100)): Hasło klienta.

Działanie: Dodaje nowego klienta do bazy danych.

```
create
    definer = root@`%` procedure addClient(IN in_firstname text, IN in_lastname text, IN in_email varchar(320), IN in_phone_number varchar(15), IN in_password varchar(100))
BEGIN
    INSERT INTO Clients (firstname, lastname, email, phone_number, password)
    VALUES (in_firstname, in_lastname, in_email, in_phone_number, in_password);
END;

grant execute on procedure addClient to client;

grant execute on procedure addClient to guest;
```

isRoomAvailable

Opis: Sprawdza dostępność mieszkania na określony okres.

Parametry:

- `flatId` (int): Identyfikator mieszkania.
- `dateFrom` (date): Data rozpoczęcia.
- `dateTo` (date): Data zakończenia.

Działanie: Sprawdza, czy mieszkanie jest dostępne w danym okresie.

```
create
    definer = root@`%` procedure isRoomAvailable(IN flatId int, IN dateFrom date, IN dateTo date)
BEGIN
    IF EXISTS (
        SELECT 1 FROM Reservations
        WHERE flat_id = flatId AND  (date_to > dateFrom AND date_from < dateTo) AND status=1
    ) THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'The room is not available for the selected dates.';
    END IF;
END;
```

loginAdmin

Opis: Logowanie administratora.

Parametry:

- `in_email` (varchar(320)): Adres email administratora.
- `in_password` (varchar(100)): Hasło administratora.

Działanie: Sprawdza poprawność danych logowania administratora.

```
create
    definer = root@`%` procedure loginAdmin(IN in_email varchar(320), IN in_password varchar(100))
BEGIN
    DECLARE adminID INT;
    DECLARE correctPassword varchar(100);
    SELECT admin_id, password INTO adminID, correctPassword
    FROM Admins
    WHERE email = in_email;
    IF correctPassword IS NULL THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'No such user found.';
    ELSEIF correctPassword=in_password THEN
        SELECT adminID AS admin_id;
    ELSE
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Incorrect password.';
    END IF;
END;

grant execute on procedure loginAdmin to guest;
```

loginClient

Opis: Logowanie klienta.

Parametry:

- `in_email` (varchar(320)): Adres email klienta.
- `in_password` (varchar(100)): Hasło klienta.

Działanie: Sprawdza poprawność danych logowania klienta.

```
create
    definer = root@`%` procedure loginClient(IN in_email varchar(320), IN in_password varchar(100))
BEGIN
    DECLARE clientID INT;
    DECLARE correctPassword varchar(100);
    SELECT client_id, password INTO clientID, correctPassword
    FROM Clients
    WHERE email = in_email;
    IF correctPassword IS NULL THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'No such user found.';
```

```
    ELSEIF correctPassword=in_password THEN
        SELECT clientID AS client_id;
    ELSE
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Incorrect password.';
    END IF;
END;

grant execute on procedure loginClient to client;

grant execute on procedure loginClient to guest;
```

Wyzwalacze

AfterInsertReservation

Opis: Wyzwalacz po dodaniu nowej rezerwacji.

Działanie: Po dodaniu nowej rezerwacji, automatycznie dodaje odpowiednią płatność.

```
create definer = root@`%` trigger AfterInsertReservation
    after insert
    on Reservations
    for each row
BEGIN
    INSERT INTO Payments (reservation_id, amount, date, admin_id)
    VALUES (NEW.reservation_id, NEW.price, NOW(), NULL);
END;
```

AfterUpdateReservation

Opis: Wyzwalacz po zaktualizowaniu rezerwacji.

Działanie: Po zaktualizowaniu rezerwacji, jeśli cena rezerwacji wzrosła, dodaje odpowiednią płatność.

```
create definer = root@`%` trigger AfterUpdateReservation
    after update
    on Reservations
    for each row
BEGIN
    IF NEW.price > OLD.price THEN
        INSERT INTO Payments (reservation_id, amount, date, admin_id)
        VALUES (NEW.reservation_id, NEW.price - OLD.price, NOW(), NULL);
    END IF;
END;
```

BeforeInsertReservation

Opis: Wyzwalacz przed dodaniem nowej rezerwacji.

Działanie: Sprawdza poprawność dat przed dodaniem nowej rezerwacji.

```
create definer = root@`%` trigger BeforeInsertReservation
    before insert
    on Reservations
    for each row
BEGIN
#     IF NEW.date_from<curdate() THEN
#         SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'You cannot set date_from to a past date' ;
#     END IF;
    IF NEW.date_from >= NEW.date_to THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'date_from must be less than date_to.';
    END IF;
END;
```

BeforeUpdateReservation

Opis: Wyzwalacz przed zaktualizowaniem rezerwacji.

Działanie: Sprawdza poprawność dat przed zaktualizowaniem rezerwacji.

```
create definer = root@`%` trigger BeforeUpdateReservation
    before update
    on Reservations
    for each row
BEGIN
#     IF NEW.date_from<curdate() THEN
#         SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'You cannot update date_from to a past date' ;
#     END IF;
    IF NEW.date_from >= NEW.date_to THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'date_from must be less than date_to.';
    END IF;
END;
```

Plik `app.js`

Plik `app.js` konfiguruje serwer, middleware oraz trasy aplikacji.

Importy

- `express`: Framework do tworzenia serwerów.

- `body-parser`: Middleware do parsowania ciała żądania.
- `mysql`: Biblioteka do obsługi MySQL z obietnicami.
- `adminRoute`: Trasy dla administratora.
- `clientRoute`: Trasy dla klienta.

Konfiguracja aplikacji

- Tworzy instancję aplikacji Express.
- Ustawia port serwera na 3000.

Middleware

- Konfiguruje middleware do parsowania ciała żądania w formacie JSON.

Konfiguracja bazy danych

- Ustawienia połączenia z bazą danych dla gościa (`dbGuest`).

Trasy

- `app.use('/admin', adminRoute)`: Trasy dla administratora.
- `app.use('/client', clientRoute)`: Trasy dla klienta.

`POST /loginClient`

Logowanie klienta.

- Pobiera email i hasło z żądania.
- Łączy się z bazą danych i wykonuje procedurę logowania klienta.
- Sprawdza wynik i zwraca odpowiednią odpowiedź.

`POST /loginAdmin`

Logowanie administratora.

- Pobiera email i hasło z żądania.
- Łączy się z bazą danych i wykonuje procedurę logowania administratora.
- Sprawdza wynik i zwraca odpowiednią odpowiedź.

`POST /signup`

Rejestracja nowego klienta.

- Pobiera dane nowego klienta z żądania.
- Łączy się z bazą danych i wykonuje procedurę dodawania nowego klienta.
- Zwraca odpowiedź potwierdzającą dodanie nowego klienta.

`POST /findAvailableFlats`

Znajdowanie dostępnych mieszkań.

- Pobiera kryteria wyszukiwania z żądania.
- Łączy się z bazą danych i wykonuje procedurę wyszukiwania dostępnych mieszkań.
- Zwraca listę dostępnych mieszkań.

`GET /all-flats`

Pobieranie listy wszystkich mieszkań.

- Łączy się z bazą danych i wykonuje zapytanie pobierające wszystkie mieszkania.
- Zwraca listę mieszkań.

Uruchomienie serwera

- Uruchamia serwer i nasłuchuje na porcie 3000.

```
import express from 'express';
import bodyParser from 'body-parser';
import mysql from 'mysql2/promise';
import adminRoute from './routes/admin.js';
import clientRoute from './routes/client.js';

import router from "./routes/client.js";

const app = express();
const port = 3000;

app.use(bodyParser.json());
const dbGuest = {
    host: '127.0.0.1',
    port: 3333,
    user: 'guest',
    password: 'guest',
    database: 'MiniDb',
    timezone: '00:00'
};
```

```
app.use('/admin', adminRoute);
app.use('/client', clientRoute);
app.post('/loginClient', async (req, res) => {
    const {email, password} = req.body;
    try {
        const connection = await mysql.createConnection(dbGuest);
        const [results] = await connection.execute("CALL loginClient(?, ?)", [email, password]);
        if (results[0].length > 0) {
            res.json({message: 'Login successful', clientID: results[0][0].client_id});
            console.log("Cllient logged clientID: " + results[0][0].client_id)
        } else {
            res.status(404).send({message: 'User not found or password incorrect'});
        }
        await connection.end();
    } catch (error) {
        if (error.sqlState === '45000') {
            console.error(error.message)
            res.status(401).send({message: error.message});
        } else {
            console.error('Database error:', error);
            res.status(500).send({message: 'Server error'});
        }
    }
});
app.post('/loginAdmin', async (req, res) => {
    const { email, password } = req.body;
    try {
        const connection = await mysql.createConnection(dbGuest);
        const [results] = await connection.execute("CALL loginAdmin(?, ?)", [email, password]);
        if (results[0].length > 0) {
            res.json({ message: 'Login successful', adminID: results[0][0].admin_id });
            console.log("Admin logged adminID: " + results[0][0].admin_id)
        } else {
            res.status(404).send({ message: 'User not found or password incorrect' });
        }
        await connection.end();
    } catch (error) {
        if (error.sqlState === '45000') {
            console.error(error.message)
            res.status(401).send({ message: error.message });
        } else {
            console.error('Database error:', error);
            res.status(500).send({ message: 'Server error' });
        }
    }
});
app.post("/signup", async (req, res) => {
    const {firstname, lastname, email, phoneNumber, password} = req.body;
    try {
        const connection = await mysql.createConnection(dbGuest);
        await connection.execute(
            "CALL addClient(?, ?, ?, ?, ?)",
            [firstname, lastname, email, phoneNumber, password]
        );
        res.status(201).send({ message: 'Client added successfully' });
        console.log('Client added successfully : '+ "{" + firstname +", " +lastname+", " +email+ ", " +phoneNumber+" }");
        await connection.end();
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Failed to add client' });
    }

})
app.post("/findAvailableFlats", async (req,res)=>{
    const {dateFrom,dateTo,numPeople,withKitchen}=req.body;
    try {
        const connection = await mysql.createConnection(dbGuest);
        const [results] = await connection.execute("CALL FindAvailableFlats(?, ?, ?, ?)", [dateFrom,dateTo,numPeople,withKitchen]);
        res.json({message: 'Available flats', flats: results[0]});
        await connection.end();
    } catch (error) {
        if (error.sqlState === '45000') {
            console.error(error.message)
            res.status(401).send({message: error.message});
        } else {
            console.error('Database error:', error);
            res.status(500).send({message: 'Server error'});
        }
    }
})
app.get('/all-flats', async (req, res) => {
    try {
        const connection = await mysql.createConnection(dbGuest);
        const [rows] = await connection.query('SELECT * FROM AllFlats');
        await connection.end();
        res.json(rows);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});

// Start server
app.listen(port, () => {
    console.log(`Server running on http://localhost:${port}`);
});
```

Plik `package.json`

Plik `package.json` zawiera meta-informacje o projekcie oraz listę zależności. Jest to kluczowy plik dla zarządzania projektem Node.js.

Kluczowe sekcje

- `name`: Nazwa projektu.
- `version`: Wersja projektu.
- `private`: Określa, że projekt jest prywatny.
- `scripts`: Skrypty do zarządzania cyklem życia aplikacji, takie jak `start`.
  - `start`: Skrypt uruchamiający serwer `node ./bin/www`.
- `dependencies`: Lista zależności produkcyjnych, które są niezbędne do działania aplikacji.
  - `body-parser`: Middleware do parsowania ciała żądania.
  - `cookie-parser`: Middleware do parsowania ciasteczek.
  - `debug`: Narzędzie do debugowania.
  - `express`: Framework do tworzenia serwerów.
  - `http-errors`: Narzędzie do obsługi błędów HTTP.
  - `morgan`: Middleware do logowania żądań HTTP.
  - `mysql`: Biblioteka do obsługi MySQL.
  - `mysql2`: Biblioteka do obsługi MySQL z obietnicami.
  - `pug`: Silnik szablonów HTML.

```
{
  "name": "server",
  "version": "0.0.0",
  "private": true,
  "type": "module",
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "^1.20.2",
    "cookie-parser": "~1.4.4",
    "debug": "~2.6.9",
    "express": "~4.16.1",
    "http-errors": "~1.6.3",
    "morgan": "~1.9.1",
    "mysql": "^2.18.1",
    "mysql2": "^3.10.0",
    "pug": "2.0.0-beta11"
  }
}
```

Opis plików w katalogu `routes`

Plik `admin.js`

Plik `admin.js` zawiera trasy związane z operacjami administracyjnymi. Każda trasa odpowiada za konkretną operację, jak zarządzanie rezerwacjami, klientami czy raportami finansowymi.

- `GET /dashboard`: Wyświetla dashboard administratora.
- `POST /signup`: Rejestracja nowego administratora.
- `GET /rooms-to-clean/today`: Pobiera listę pokoi do sprzątania na dzisiaj.
- `GET /rooms-to-clean/:date`: Pobiera listę pokoi do sprzątania na podany dzień.
- `GET /reservations`: Pobiera listę wszystkich rezerwacji.
- `GET /reservations/client/:clientId`: Pobiera rezerwacje danego klienta.
- `GET /reservations/:reservationId`: Pobiera szczegóły rezerwacji o podanym ID.
- `GET /client`: Pobiera listę wszystkich klientów.
- `GET /client/:clientId`: Pobiera szczegóły klienta o podanym ID.
- `GET /currently-occupied-rooms`: Pobiera listę aktualnie zajętych pokoi.
- `GET /cancelled-reservations`: Pobiera listę wszystkich anulowanych rezerwacji.
- `GET /cancelled-reservations/:year/:month`: Pobiera listę anulowanych rezerwacji w danym miesiącu i roku.
- `GET /reservations-to-refund`: Pobiera listę rezerwacji do zwrotu.
- `GET /report/:year/:month`: Generuje miesięczny raport finansowy za podany miesiąc i rok.
- `GET /report-previous-month`: Generuje miesięczny raport finansowy za poprzedni miesiąc.
- `POST /refund-reservation`: Przetwarza zwrot za rezerwację.
- `GET /returned-payments/:adminId`: Pobiera historię zwrotów dokonanych przez danego administratora.
- `GET /payment-history/:clientId`: Pobiera historię płatności danego klienta.

```
import { Router } from 'express';
import mysql from "mysql2/promise";
const router = Router();
const dbAdmin = {
    host: '127.0.0.1',
    port: 3333,
    user: 'root',
    password: 'root',
    database: 'MiniDb',
    timezone: '00:00'
};
router.get('/dashboard', (req, res) => {
    res.send('Admin Dashboard');
});


router.post("/signup", async (req, res) => {
    const {firstname, lastname, email, phoneNumber, password} = req.body;
    try {
        const connection = await mysql.createConnection(dbAdmin);
        await connection.execute(
            "CALL addClient(?, ?, ?, ?, ?)",
            [firstname, lastname, email, phoneNumber, password]
        );
        res.status(201).send({ message: 'Admin added successfully' });
        console.log('Admin added successfully : '+ "{" + firstname +", " +lastname+", " +email+ ", " +phoneNumber+" }");
        await connection.end();
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Failed to add Admin' });
```

```
    }

})
router.get('/rooms-to-clean/today', async (req, res) => {
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [results] = await connection.execute("CALL GetRoomsToCleanToday()");
        res.json(results[0]);
        console.log("Rooms to clean today: "+results[0])
        await connection.end();

    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.get('/rooms-to-clean/:date', async (req, res) => {
    const  date  = req.params.date;
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [results] = await connection.execute("CALL GetRoomsToClean(?)", [date]);
        res.json(results[0]);
        console.log("Rooms to clean "+date+": "+results[0])
        await connection.end();
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});

router.get('/reservations', async (req, res) => {
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [reservations] = await connection.query('SELECT * FROM Reservations');
        await connection.end();
        res.json(reservations);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.get('/reservations/client/:clientId', async (req, res) => {
    const clientId = req.params.clientId;
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [reservations] = await connection.query('SELECT * FROM Reservations WHERE client_id = ?', [clientId]);
        await connection.end();
        res.json(reservations);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.get('/reservations/:reservationId', async (req, res) => {
    const reservationId = req.params.reservationId;
    console.log(reservationId)
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [result] = await connection.execute(
            "SELECT * FROM Reservations WHERE reservation_id=?",
            [reservationId]
        );
        if (result.length > 0) {
            res.json(result[0]);
        } else {
            res.status(404).send({ message: 'Reservation not found' });
        }
        await connection.end();
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});



router.get('/client', async (req, res) => {
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [client] = await connection.query('SELECT * FROM Clients');
        await connection.end();
        res.json(client);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.get('/client/:clientId', async (req, res) => {
    const clientId = req.params.clientId;
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [client] = await connection.query('SELECT * FROM Clients WHERE client_id = ?', [clientId]);
        await connection.end();
        res.json(client);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.get('/currently-occupied-rooms', async (req, res) => {
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [rooms] = await connection.query("SELECT * FROM CurrentlyOccupiedRooms");
        await connection.end();
```

```
        res.json(rooms);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.get('/cancelled-reservations', async (req, res) => {
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [result] = await connection.execute("SELECT * FROM AllCancelledReservations");
        await connection.end();
        res.json(result);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.get('/cancelled-reservations/:year/:month', async (req, res) => {
    const { year, month } = req.params;
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [results] = await connection.execute(
            "CALL ShowCancelledReservations(?, ?)",
            [year, month]
        );
        await connection.end();
        res.json(results);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.get('/reservations-to-refund', async (req, res) => {
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [refundReservations] = await connection.execute("SELECT * FROM ReservationsToRefund");
        await connection.end();
        res.json(refundReservations);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.get('/report/:year/:month', async (req, res) => {
    const { year, month } = req.params;
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [results] = await connection.execute("CALL MonthlyFinancialReport(?, ?)", [parseInt(year), parseInt(month)]);
        await connection.end();
        res.json(results[0]);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.get('/report-previous-month', async (req, res) => {
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [results] = await connection.execute("CALL MonthlyFinancialReportPreviousMonth()");
        await connection.end();
        res.json(results[0]);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.post('/refund-reservation', async (req, res) => {
    const { adminId, reservationId, amountToRefund } = req.body;
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [results] = await connection.execute(
            "CALL RefundReservation(?, ?, ?)",
            [adminId, reservationId, parseFloat(amountToRefund)]
        );
        await connection.end();
        res.json({ message: 'Refund processed successfully', details: results });
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error', error: error.message });
    }
});
router.get('/returned-payments/:adminId', async (req, res) => {
    const adminId = parseInt(req.params.adminId);
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [results] = await connection.execute(
            "CALL ShowReturnedPaymentsByAdmin(?)",
            [adminSetAdminId]
        );
        await connection.end();
        res.json(results[0]);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.get('/payment-history/:clientId', async (req, res) => {
    const clientId = parseInt(req.params.clientId);
    try {
        const connection = await mysql.createConnection(dbAdmin);
        const [results] = await connection.execute(
            "CALL ShowPaymentHistory(?)",
            [clientId]
        );
```

```
        await connection.end();
        res.json(results[0]);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});



export default router;
```

Plik `client.js`

Plik `client.js` zawiera trasy związane z operacjami wykonywanymi przez klienta, takimi jak przeglądanie profilu, historii płatności oraz zarządzanie rezerwacjami.

- `GET /profile`: Wyświetla profil klienta.
- `POST /payment-history`: Pobiera historię płatności na podstawie adresu email klienta.
- `POST /cancel-reservation`: Anuluje rezerwację na podstawie adresu email klienta i ID rezerwacji.
- `POST /update-reservation-date`: Aktualizuje daty rezerwacji na podstawie adresu email klienta, ID rezerwacji oraz nowych dat.
- `POST /update-reservation-date-from`: Aktualizuje datę rozpoczęcia rezerwacji.
- `POST /update-reservation-date-to`: Aktualizuje datę zakończenia rezerwacji.
- `POST /add-reservation`: Dodaje nową rezerwację na podstawie danych klienta i szczegółów rezerwacji.

```
import {Router} from 'express';
import mysql from 'mysql2/promise';

const router = Router();
const dbClient = {
    host: '127.0.0.1',
    port: 3333,
    user: 'client',
    password: 'client',
    database: 'MiniDb',
    timezone: '00:00'
};
router.get('/profile', (req, res) => {
    res.send('Client Profile');
});




router.post('/payment-history', async (req, res) => {
    const { email } = req.body;
    try {
        const connection = await mysql.createConnection(dbClient);
        const [client] = await connection.execute(
            "CALL GetClientId(?)",
            [email]
        );
        if (client.length === 0) {
            res.status(404).send({ message: 'Client not found' });
            return;
        }
        const clientId = client[0][0].client_id;

        const [results] = await connection.execute(
            "CALL ShowPaymentHistory(?)",
            [clientId]
        );
        await connection.end();
        res.json(results[0]);
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});

router.post('/cancel-reservation', async (req, res) => {
    const { email, reservationId } = req.body;
    try {
        const connection = await mysql.createConnection(dbClient);
        const [client] = await connection.execute(
            "CALL GetClientId(?)",
            [email]
        );
        if (client.length === 0) {
            res.status(404).send({ message: 'Client not found' });
            return;
        }
        const clientId = client[0][0].client_id;

        const [results] = await connection.execute(
            "CALL CancelFutureReservation(?, ?)",
            [clientId, reservationId]
        );
        await connection.end();
        console.log('Reservation cancelled successfully')
        res.send( {message: 'Reservation cancelled successfully' });
    } catch (error) {
        console.error('Database error:', error);
        if (error.sqlState === '45000') {
            res.status(400).send({ message: error.message });
        } else {
            res.status(500).send({ message: 'Internal server error' });
        }
    }
});
```

```
router.post('/update-reservation-date', async (req, res) => {
    const { reservationId, email, newDateFrom, newDateTo } = req.body;
    try {
        const connection = await mysql.createConnection(dbClient);
        const [client] = await connection.execute(
            "CALL GetClientId(?)",
            [email]
        );
        if (client.length === 0) {
            res.status(404).send({ message: 'Client not found' });
            console.log('Client not found')
            return;
        }
        const clientId = client[0][0].client_id;
        await connection.execute(
            "CALL UpdateReservationDate(?, ?, ?, ?)",
            [reservationId, clientId, newDateFrom, newDateTo]
        );
        await connection.end();
        res.send({ message: 'Reservation dates updated successfully' });
        console.log('Reservation dates updated successfully')
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});


router.post('/update-reservation-date-from', async (req, res) => {
    const { reservationId, email, newDateFrom } = req.body;
    try {
        const connection = await mysql.createConnection(dbClient);
        const [client] = await connection.execute(
            "CALL GetClientId(?)",
            [email]
        );
        if (client.length === 0) {
            res.status(404).send({ message: 'Client not found' });
            return;
        }
        const clientId = client[0][0].client_id;
        await connection.execute(
            "CALL UpdateReservationDateFrom(?, ?, ?)",
            [reservationId, clientId, newDateFrom]
        );
        await connection.end();
        res.send({ message: 'Reservation start date updated successfully' });
        console.log('Reservation start date updated successfully' )
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});


router.post('/update-reservation-date-to', async (req, res) => {
    const { reservationId, email, newDateTo } = req.body;
    try {
        const connection = await mysql.createConnection(dbClient);
        const [client] = await connection.execute(
            "CALL GetClientId(?)",
            [email]
        );
        if (client.length === 0) {
            res.status(404).send({ message: 'Client not found' });
            return;
        }
        const clientId = client[0][0].client_id;
        await connection.execute(
            "CALL UpdateReservationDateTo(?, ?, ?)",
            [reservationId, clientId, newDateTo]
        );
        await connection.end();
        res.send({ message: 'Reservation end date updated successfully' });
        console.log('Reservation end date updated successfully')
    } catch (error) {
        console.error('Database error:', error);
        res.status(500).send({ message: 'Internal server error' });
    }
});
router.post('/add-reservation', async (req, res) => {
    const { email, flatId, dateFrom, dateTo, numPeople } = req.body;
    try {
        const connection = await mysql.createConnection(dbClient);
        const [client] = await connection.execute(
            "CALL GetClientId(?)",
            [email]
        );
        if (client.length === 0) {
            res.status(404).send({ message: 'Client not found' });
            return;
        }
        const clientId = client[0][0].client_id;

        const [results] = await connection.execute(
            "CALL AddReservation(?, ?, ?, ?, ?)",
            [clientId, flatId, dateFrom, dateTo,  numPeople]
        );
        await connection.end();
        res.send({ message: 'Reservation added successfully'});
    } catch (error) {
        console.error('Database error:', error);
        if (error.sqlState === '45000') {
            res.status(400).send({ message: error.message });
```

```
        } else {
            res.status(500).send({ message: 'Internal server error' });
        }
    }
});
export default router;
```

Plik `index.js`

Plik `index.js` definiuje trasę dla strony głównej aplikacji.

- `GET /` : Renderuje stronę główną aplikacji, wyświetlając tytuł "Express".

```
żvar express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

Plik `users.js`

Plik `users.js` zawiera trasy związane z operacjami na użytkownikach, takimi jak wyświetlanie listy użytkowników.

- `GET /` : Wyświetla listę użytkowników.

```
var express = require('express');
var router = express.Router();

/* GET users listing. */
router.get('/', function(req, res, next) {
  res.send('respond with a resource');
});

module.exports = router;
```

Opis plików w katalogu `test`

Plik `adminClean.js`

Plik `adminClean.js` zawiera testy jednostkowe dla funkcji związanych z zarządzaniem czystością pokoi. Testuje funkcje odpowiedzialne za pobieranie listy pokoi do sprzątania na dany dzień.

- `fetchRoomsToCleanByDate(date)` : Testuje funkcję pobierającą listę pokoi do sprzątania na podany dzień.
- `fetchRoomsToCleanToday()` : Testuje funkcję pobierającą listę pokoi do sprzątania na dzisiaj.

```
async function fetchRoomsToCleanByDate(date) {
    try {
        const response = await fetch(`http://localhost:3000/admin/rooms-to-clean/${date}`, {
            method: 'GET',
            headers: {
                'Content-Type': 'application/json'
            }
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const data = await response.json();
        console.log('Rooms to be cleaned:', data);
    } catch (error) {
        console.error('Error fetching rooms to clean:', error);
    }
}
async function fetchRoomsToCleanToday() {
    try {
        const response = await fetch('http://localhost:3000/admin/rooms-to-clean/today', {
            method: 'GET',
            headers: {
                'Content-Type': 'application/json'
            }
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const data = await response.json();
        console.log('Rooms to be cleaned today:', data);
    } catch (error) {
        console.error('Error fetching rooms to clean today:', error);
    }
}

fetchRoomsToCleanToday();
fetchRoomsToCleanByDate('2024-06-09');
```

Plik `adminRaports.js`

Plik `adminRaports.js` zawiera testy jednostkowe dla funkcji związanych z generowaniem raportów finansowych. Testuje funkcje odpowiedzialne za generowanie miesięcznych raportów finansowych.

- `fetchMonthlyFinancialReport(year, month)` : Testuje funkcję pobierającą miesięczny raport finansowy za podany miesiąc i rok.
- `fetchMonthlyFinancialReportPreviousMonth()` : Testuje funkcję pobierającą miesięczny raport finansowy za poprzedni miesiąc.

```
async function fetchMonthlyFinancialReport(year, month) {
    try {
        const response = await fetch(`http://localhost:3000/admin/report/${year}/${month}`);
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        const result = await response.json();
        console.log('Monthly financial report:', result);
    } catch (error) {
        console.error('Error fetching monthly financial report:', error);
    }
}

async function fetchMonthlyFinancialReportPreviousMonth() {
    try {
        const response = await fetch('http://localhost:3000/admin/report-previous-month');
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        const result = await response.json();
        console.log('Monthly financial report for previous month:', result);
    } catch (error) {
        console.error('Error fetching monthly financial report for previous month:', error);
    }
}

fetchMonthlyFinancialReportPreviousMonth()
fetchMonthlyFinancialReport(2024, 5 );
```

Plik `adminRefund.js`

Plik `adminRefund.js` zawiera testy jednostkowe dla funkcji związanych z przetwarzaniem zwrotów rezerwacji. Testuje funkcje odpowiedzialne za sprawdzanie możliwości zwrotu oraz realizację zwrotu.

- `testRefundReservation(adminId, reservationId, amountToRefund)` : Testuje funkcję przetwarzającą zwrot za rezerwację.
- `fetchReturnedPaymentsByAdmin(adminId)` : Testuje funkcję pobierającą historię zwrotów dokonanych przez danego administratora.

```
async function testRefundReservation(adminId, reservationId, amountToRefund) {
    const url = 'http://localhost:3000/admin/refund-reservation';
    const body = {
        adminId,
        reservationId,
        amountToRefund
    };

    try {
        const response = await fetch(url, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(body)
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const data = await response.json();
        console.log('Refund successful:', data);
    } catch (error) {
        console.error('Error testing refund reservation:', error);
    }
}
async function fetchReturnedPaymentsByAdmin(adminId) {
    const url = `http://localhost:3000/returned-payments/${adminId}`;
    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        const payments = await response.json();
        console.log('Returned payments by admin:', payments);
    } catch (error) {
        console.error('Error fetching returned payments by admin:', error);
    }
}

fetchReturnedPaymentsByAdmin(1);

testRefundReservation(2 , 20, 300.00);
```

Plik `adminViews.js`

Plik `adminViews.js` zawiera testy jednostkowe dla funkcji związanych z przeglądaniem różnych widoków administracyjnych. Testuje funkcje odpowiedzialne za pobieranie listy anulowanych rezerwacji, rezerwacji do zwrotu oraz aktualnie zajętych pokoi.

- `fetchCancelledReservations()` : Testuje funkcję pobierającą listę wszystkich anulowanych rezerwacji.
- `fetchRefundReservations()` : Testuje funkcję pobierającą listę rezerwacji do zwrotu.
- `fetchCurrentlyOccupiedRooms()` : Testuje funkcję pobierającą listę aktualnie zajętych pokoi.
- `fetchCancelledReservationsByDate(year, month)` : Testuje funkcję pobierającą listę anulowanych rezerwacji w danym miesiącu i roku.

```
async function fetchCancelledReservations() {
    try {
        const response = await fetch('http://localhost:3000/admin/cancelled-reservations');
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        const cancelledReservations = await response.json();
        console.log('Cancelled reservations:', cancelledReservations);
    } catch (error) {
        console.error('Error fetching cancelled reservations:', error);
    }
}
async function fetchRefundReservations() {
    try {
        const response = await fetch('http://localhost:3000/admin/refund-reservations');
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        const refundReservations = await response.json();
        console.log('Reservations to refund:', refundReservations);
    } catch (error) {
        console.error('Error fetching reservations to refund:', error);
    }
}
async function fetchCurrentlyOccupiedRooms() {
    try {
        const response = await fetch('http://localhost:3000/admin/currently-occupied-rooms');
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        const rooms = await response.json();
        console.log('Currently occupied rooms:', rooms);
    } catch (error) {
        console.error('Error fetching data:', error);
    }
}
async function fetchCancelledReservationsByDate(year, month) {
    const url = `http://localhost:3000/admin/cancelled-reservations/${year}/${month}`;
    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        const reservations = await response.json();
        console.log('Cancelled reservations for given period:', reservations);
    } catch (error) {
        console.error('Error fetching cancelled reservations:', error);
    }
}

fetchCancelledReservationsByDate(2024, 4);

// fetchCurrentlyOccupiedRooms();
//
// fetchRefundReservations();

fetchCancelledReservations();
```

Plik `clientReservations.js`

Plik `clientReservations.js` zawiera testy jednostkowe dla funkcji związanych z zarządzaniem rezerwacjami klientów. Testuje funkcje odpowiedzialne za anulowanie oraz dodawanie nowych rezerwacji.

- `cancelReservation(email, reservationId)` : Testuje funkcję anulującą rezerwację na podstawie adresu email klienta i ID rezerwacji.
- `addReservation(email, flatId, dateFrom, dateTo, numPeople)` : Testuje funkcję dodającą nową rezerwację na podstawie danych klienta i szczegółów rezerwacji.

```
async function cancelReservation(email, reservationId) {
    const url = 'http://localhost:3000/client/cancel-reservation';  // Zmień URL jeśli potrzebujesz
    const body = {
        email,
        reservationId
    };

    try {
        const response = await fetch(url, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(body)
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const result = await response.json();
        console.log('Cancel reservation response:', result);
    } catch (error) {
        console.error('Error testing cancel reservation:', error);
    }
}
async function addReservation(email, flatId, dateFrom, dateTo, numPeople) {
    const url = 'http://localhost:3000/client/add-reservation';
    const body = {
        email,
        flatId,
        dateFrom,
        dateTo,
        numPeople
```

```
    };

    try {
        const response = await fetch(url, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(body)
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const result = await response.json();
        console.log('Add reservation response:', result);
    } catch (error) {
        console.error('Error testing add reservation:', error);
    }
}
//addReservation('tomasz.zielinski@example.com', 12, '2024-06-01', '2024-06-07', 2);

// cancelReservation('zofia.stepien@example.com', 16);
```

Plik `login.js`

Plik `login.js` zawiera testy jednostkowe dla funkcji związanych z logowaniem klientów i administratorów. Testuje funkcje odpowiedzialne za proces logowania.

- `loginClient(email, password)` : Testuje funkcję logującą klienta.
- `loginAdmin(email, password)` : Testuje funkcję logującą administratora.

```
async function loginClient(email,password) {
    const url = 'http://localhost:3000/loginClient';
    const data = {
        email: email,
        password: password
    };

    try {
        const response = await fetch(url, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const result = await response.json();
        console.log('Login successful:', result);
    } catch (error) {
        console.error('Error during login:', error.message);
    }
}
async function loginAdmin(email,password) {
    const url = 'http://localhost:3000/loginAdmin';
    const data = {
        email: email,
        password: password
    };

    try {
        const response = await fetch(url, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const result = await response.json();
        console.log('Login successful:', result);
    } catch (error) {
        console.error('Error during login:', error.message);
    }
}

loginClient("jan.kowalski@example.com","haslo3");
loginAdmin("admin1@example.com","haslo1")
```

Plik `paymentHistory.js`

Plik `paymentHistory.js` zawiera testy jednostkowe dla funkcji związanych z przeglądaniem historii płatności. Testuje funkcje odpowiedzialne za pobieranie historii płatności dla danego klienta.

- `fetchPaymentHistory(clientId)` : Testuje funkcję pobierającą historię płatności danego klienta.
- `fetchClientPaymentHistory(email)` : Testuje funkcję pobierającą historię płatności na podstawie adresu email klienta.

```
async function fetchPaymentHistory(clientId) {
    try {
        const response = await fetch(`http://localhost:3000/admin/payment-history/${clientId}`);
```

```
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        const history = await response.json();
        console.log('Payment history:', history);
    } catch (error) {
        console.error('Error fetching payment history:', error);
    }
}
async function fetchClientPaymentHistory(email) {
    try {
        const response = await fetch('http://localhost:3000/client/payment-history', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({ email })
        });
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        const history = await response.json();
        console.log('Client payment history:', history);
    } catch (error) {
        console.error('Error fetching client payment history:', error);
    }
}


fetchClientPaymentHistory('jan.kowalski@example.com');

fetchPaymentHistory(1);
```

Plik `showFlats.js`

Plik `showFlats.js` zawiera testy jednostkowe dla funkcji związanych z przeglądaniem mieszkań. Testuje funkcje odpowiedzialne za znajdowanie dostępnych mieszkań oraz pobieranie listy wszystkich mieszkań.

- `findAvailableFlats(dateFrom, dateTo, numPeople, withKitchen)` : Testuje funkcję znajdującą dostępne mieszkania na podstawie podanych kryteriów.
- `allFlats()` : Testuje funkcję pobierającą listę wszystkich mieszkań.

```
async function findAvailableFlats(dateFrom,dateTo,numPeople,withKitchen) {
    const url = 'http://localhost:3000/findAvailableFlats';
    const data = {
        dateFrom:dateFrom,
        dateTo: dateTo,
        numPeople:numPeople,
        withKitchen:withKitchen
    };

    try {
        const response = await fetch(url, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const result = await response.json();
        console.log('Success:', result);
    } catch (error) {
        console.error('Error during finding flats:', error);
    }
}
async function allFlats() {
    const url = 'http://localhost:3000/all-flats';
    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        const flats = await response.json();
        console.log('All flats:', flats);
    } catch (error) {
        console.error('Error fetching all flats:', error);
    }
}

allFlats();

findAvailableFlats("2024-06-13","2024-06-20",0,0);
```

Plik `signup.js`

Plik `signup.js` zawiera testy jednostkowe dla funkcji związanych z rejestracją klientów i administratorów. Testuje funkcje odpowiedzialne za proces rejestracji.

- `addClient(firstname, lastname, email, phoneNumber, password)` : Testuje funkcję rejestrującą nowego klienta.
- `addAdmin(firstname, lastname, email, phoneNumber, password)` : Testuje funkcję rejestrującą nowego administratora.

```
async function addClient(firstname,lastname,email,phoneNumber,password) {
    const url = 'http://localhost:3000/signup';
    const data = {
        firstname: firstname,
        lastname: lastname,
```

```
        email: email,
        phoneNumber: phoneNumber,
        password: password
    };

    try {
        const response = await fetch(url, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const result = await response.json();
        console.log('Success:', result);
    } catch (error) {
        console.error('Error during client addition:', error);
    }
}
async function addAdmin(firstname,lastname,email,phoneNumber,password) {
    const url = 'http://localhost:3000/admin/signup';
    const data = {
        firstname: firstname,
        lastname: lastname,
        email: email,
        phoneNumber: phoneNumber,
        password: password
    };

    try {
        const response = await fetch(url, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const result = await response.json();
        console.log('Success:', result);
    } catch (error) {
        console.error('Error during admin addition:', error);
    }
}

addClient("client2","client2","client2@email.com","123456789","haslo1234");
```

Plik `updateReservation.js`

Plik `updateReservation.js` zawiera testy jednostkowe dla funkcji związanych z aktualizowaniem rezerwacji. Testuje funkcje odpowiedzialne za aktualizację dat rezerwacji.

- `updateReservationDate(reservationId, email, newDateFrom, newDateTo)` : Testuje funkcję aktualizującą daty rezerwacji na podstawie adresu email klienta, ID rezerwacji oraz nowych dat.
- `updateReservationDateFrom(reservationId, email, newDateFrom)` : Testuje funkcję aktualizującą datę rozpoczęcia rezerwacji.
- `updateReservationDateTo(reservationId, email, newDateTo)` : Testuje funkcję aktualizującą datę zakończenia rezerwacji.

```
async function updateReservationDate(reservationId, email, newDateFrom, newDateTo) {
    const url = 'http://localhost:3000/client/update-reservation-date';
    const body = { reservationId, email, newDateFrom, newDateTo };

    try {
        const response = await fetch(url, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(body)
        });

        const result = await response.json();
        if (!response.ok) {
            console.error('Error:', result);
        } else {
            console.log('Success:', result);
        }
    } catch (error) {
        console.error('Network or other error:', error);
    }
}



async function updateReservationDateFrom(reservationId, email, newDateFrom) {
    const url = 'http://localhost:3000/client/update-reservation-date-from';
    const body = { reservationId, email, newDateFrom };

    try {
        const response = await fetch(url, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(body)
        });
```

```
        const result = await response.json();
        if (!response.ok) {
            console.error('Error:', result);
        } else {
            console.log('Success:', result);
        }
    } catch (error) {
        console.error('Network or other error:', error);
    }
}
async function updateReservationDateTo(reservationId, email, newDateTo) {
    const url = 'http://localhost:3000/client/update-reservation-date-to';
    const body = { reservationId, email, newDateTo };

    try {
        const response = await fetch(url, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(body)
        });

        const result = await response.json();
        if (!response.ok) {
            console.error('Error:', result);
        } else {
            console.log('Success:', result);
        }
    } catch (error) {
        console.error('Network or other error:', error);
    }
}


//updateReservationDate(8, "katarzyna.wojcik@example.com", '2024-06-07', '2024-06-10');
// updateReservationDateFrom(8, "katarzyna.wojcik@example.com", '2024-06-09');
updateReservationDateTo(1, "jan.kowalski@example.com", '2024-06-05');
```

Opis plików w katalogu `bin`

Plik `www`

Plik `www` zawiera skrypt uruchamiający serwer HTTP dla aplikacji. Skrypt ten konfiguruje port, na którym serwer nasłuchuje, oraz uruchamia aplikację.

- `normalizePort(val)` : Normalizuje port do liczby, ciągu znaków lub wartości false.
- `onError(error)` : Obsługuje błędy serwera HTTP.
- `onListening()` : Obsługuje zdarzenie `listening` serwera HTTP.

Sekcje skryptu:

- **Pobieranie portu z konfiguracji środowiska i zapisywanie go w Express**.
- **Tworzenie serwera HTTP przy użyciu aplikacji Express**.
- **Nasłuchiwanie na podanym porcie i obsługa zdarzeń `error` oraz `listening`**.

```
#!/usr/bin/env node

/**
 * Module dependencies.
 */

var app = require('../app');
var debug = require('debug')('server:server');
var http = require('http');

/**
 * Get port from environment and store in Express.
 */

var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

/**
 * Create HTTP server.
 */

var server = http.createServer(app);

/**
 * Listen on provided port, on all network interfaces.
 */

server.listen(port);
server.on('error', onError);
server.on('listening', onListening);

/**
 * Normalize a port into a number, string, or false.
 */

function normalizePort(val) {
  var port = parseInt(val, 10);

  if (isNaN(port)) {
    // named pipe
    return val;
  }

  if (port >= 0) {
    // port number
    return port;
```

```
  }

  return false;
}

/**
 * Event listener for HTTP server "error" event.
 */

function onError(error) {
  if (error.syscall !== 'listen') {
    throw error;
  }

  var bind = typeof port === 'string'
    ? 'Pipe ' + port
    : 'Port ' + port;

  // handle specific listen errors with friendly messages
  switch (error.code) {
    case 'EACCES':
      console.error(bind + ' requires elevated privileges');
      process.exit(1);
      break;
    case 'EADDRINUSE':
      console.error(bind + ' is already in use');
      process.exit(1);
      break;
    default:
      throw error;
  }
}

/**
 * Event listener for HTTP server "listening" event.
 */

function onListening() {
  var addr = server.address();
  var bind = typeof addr === 'string'
    ? 'pipe ' + addr
    : 'port ' + addr.port;
  debug('Listening on ' + bind);
}
```

Opis plików w katalogu `public/stylesheets`

Plik `style.css`

Plik `style.css` zawiera główne style CSS dla aplikacji. Definiuje ogólny wygląd aplikacji, takie jak padding, fonty oraz kolory linków.

```
body {
  padding: 50px;
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
}

a {
  color: #00B7FF;
}
```

Opis plików w katalogu `views`

Plik `error.pug`

Plik `error.pug` jest szablonem dla strony błędów. Wyświetla komunikaty błędów oraz dodatkowe informacje diagnostyczne.

- `h1= message`: Wyświetla główny komunikat błędu.
- `h2= error.status`: Wyświetla kod statusu błędu.
- `pre #{error.stack}`: Wyświetla stack trace błędu.

```
extends layout

block content
  h1= message
  h2= error.status
  pre #{error.stack}
```

Plik `index.pug`

Plik `index.pug` jest szablonem dla strony głównej aplikacji.

- `h1= title`: Wyświetla tytuł strony głównej.
- `p Welcome to #{title}`: Wyświetla powitalny komunikat z tytułem strony.

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

Plik `layout.pug`

Plik `layout.pug` jest podstawowym szablonem layoutu dla aplikacji. Definiuje ogólną strukturę dokumentu HTML, w tym nagłówek, sekcję ciała oraz miejsce na dynamiczne treści stron.

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

Jak uruchomić aplikację?

1. Uruchomienie kontenera MySQL

- Otwórz terminal.
- Uruchomienie kontenera MySQL:

```
docker run --name=mysql_server --env=MYSQL_ROOT_PASSWORD=root --env=MYSQL_DATABASE=MiniDb --network=bridge -p 3333:3306 -d mysql:latest
```

2. Skopiowanie pliku SQL do kontenera

- Skopiuj plik generateMiniDb.sql do kontenera:

```
docker cp /ścieżka/do/pliku/generateMiniDb.sql mysql_server:/generateMiniDb.sql
```

3. Wejście do kontenera MySQL i załadowanie danych

- Wejdź do kontenera MySQL:

4. Uruchomienie aplikacji Node.js

- Otwórz nowy terminal.
- Przejdź do katalogu z Twoją aplikacją Node.js:
- Przejdź do folderu, w którym znajduje się Twój plik app.js.
- Uruchom aplikację Node.js:

```
node app.js
```

Uwaga
Zakładamy, że aplikacja Docker jest zainstalowana oraz wszystkie zależności