



Politechnika Wrocławska

PROJEKT Z BAZ DANYCH

MENEDŻER OGRODU ZOOLOGICZNEGO

Autorzy:

Krzysztof Dydak, 242075
Damian Hrycalik, 241340
Szymon Pakuła, 241313

Prowadzący:

Dr inż. Roman PTAK, W4/K9

Termin zajęć:

Wtorek, 13:15

ROK AKADEMICKI 2019-2020

Spis treści

1	Wstęp	4
1.1	Cel projektu	4
1.2	Zakres projektu	4
2	Analiza wymagań	4
2.1	Opis działania i schemat logiczny systemu	4
2.2	Opis zasobów ludzkich	5
2.3	Wymagania funkcjonalne	6
2.4	Wymagania niefunkcjonalne	6
3	Projekt systemu	7
3.1	Projekt bazy danych	7
3.1.1	Model logiczny	7
3.1.2	Model fizyczny i konceptualny	7
3.1.3	Inne elementy schematu - mechanizmy przetwarzania danych . . .	8
3.1.4	Zabezpieczenia na poziomie bazy danych	8
3.1.5	Widoki	9
3.2	Projekt aplikacji użytkownika	9
3.2.1	Architektura aplikacji i diagramy projektowe	10
3.2.2	Projekt wybranych funkcji systemu	10
3.2.3	Metoda połączenia do bazy danych	11
3.2.4	Zabezpieczenia na poziomie aplikacji	11
4	Implementacja systemu baz danych	12
4.1	Tworzenie tabel i definiowanie ograniczeń	12
4.2	Implementacja mechanizmów przetwarzania danych	14
4.2.1	Procedury	14
4.2.2	Widoki	15
4.2.3	Wyzwalacze	16
4.2.4	Indeksy	16
4.3	Implementacja uprawnień i innych zabezpieczeń	17
4.3.1	Testowanie uprawnień	18
4.4	Testowanie bazy danych na przykładowych danych	21
4.4.1	Wstęp	21
4.5	Sposoby generowania danych	21
4.5.1	Testy wydajnościowe	22
4.5.2	Omówienie danych	25
5	Implementacja i testy aplikacji	25
5.1	Instalacja i konfigurowanie systemu	25
5.2	Instrukcja użytkownika aplikacji	26
5.3	Testowanie opracowanych funkcji systemu	31

5.4	Omówienie wybranych rozwiązań programistycznych	35
5.4.1	Implementacja interfejsu dostępu do bazy danych	35
5.4.2	Implementacja wybranych funkcjonalności systemu	37
5.4.3	Implementacja mechanizmów bezpieczeństwa	39
6	Podsumowanie i wnioski	41
6.1	Podsumowanie	41
6.1.1	Wnioski	41
	Literatura	43
	Spis rysunków	43
	Spis tablic	43

1. Wstęp

1.1. Cel projektu

Zaprojektowanie i implementacja bazy danych oraz aplikacji internetowej z prostym interfejsem użytkownika przeznaczonych do obsługi ogrodu zoologicznego.

1.2. Zakres projektu

W naszym zakresie jest podsystem odpowiedzialny za zarządzanie zwierzętami.

2. Analiza wymagań

2.1. Opis działania i schemat logiczny systemu

Ogród zoologiczny, dla którego określonych pracowników przeznaczona jest aplikacja, to placówka edukacyjna oraz rekreacyjna. Jest urządzonym oraz zagospodarowanym terenem wraz z infrastrukturą techniczną i budynkami funkcjonalnie z nim związanymi, przystosowany do hodowli i aklimatyzacji zwierząt pochodzących z różnych rejonów geograficznych, udostępniony dla zwiedzających.

Nasz system usprawnia przechowywanie danych na temat zwierząt w ośrodku oraz ułatwia organizację i zarządzanie nimi.

Nasz system umożliwiać będzie organizację i zarządzanie zwierzętami w ogrodzie zoologicznym w oparciu o relacyjną bazę danych. Tabele opisywać będą m. in. zwierzęta, pracowników (np. opiekunów, weterynarzy, administratorów), wybiegi dla zwierząt, noty weterynaryjne. W obowiązkach pracownika będzie modyfikacja podstawowych danych, takich jak ilość zwierząt w danym wybiegu oraz edycja swojego profilu (tj. dane kontaktowe, adres korespondencyjny). Natomiast administratorzy będą odpowiedzialni m. in. za dodawanie i modyfikację pracowników, ustalanie grafiku pracownika. Dostęp do bazy danych będzie odbywał się za pośrednictwem aplikacji internetowej. Określone operacje mogą zostać podjęte w zależności od rodzaju konta użytkownika (pracownik, administrator).

Aplikacja będzie działać w oparciu o technologie webowe. Dostęp do niej będzie się odbywał przez zalogowanie się do konta na stronie internetowej. W naszym planie jest

skorzystanie z języka JavaScript i framework-u przeznaczonego tworzenia aplikacji webowych - Express.js. Dane będą przechowywane w relacyjnej bazie danych. Do tego celu wybraliśmy bazę danych PostgreSQL.

2.2. Opis zasobów ludzkich

Każde **zwierzę** będzie reprezentowane przez następujące dane:

- Nazwa gatunkowa,
- Imię,
- Numer identyfikacyjny,
- Miejsce urodzenia,
- Data urodzenia,
- Wybieg na którym dane zwierzę mieszka,
- Opiekunowie,
- Adnotacja do aktualnego stanu zwierzęcia.

Pracownik ogrodu może aktualizować informacje na temat zwierząt mieszkających w zoo. Każdemu z nich zostanie utworzone konto, któremu będzie przypisany unikalny numer identyfikacyjny. Profil użytkownika będzie zawierał dane kontaktowe, adres korespondencyjny, a także harmonogram prac i listę zwierząt, które są pod opieką pracownika.

Pracownik ogrodu będzie reprezentowany w przybliżeniu przez następujące dane:

- Imię
- Nazwisko
- Numer identyfikacyjny
- Dane kontaktowe
- Adres korespondencyjny
- Profesja (np. Opiekun, weterynarz)

2.3. Wymagania funkcjonalne

- Pracownik może modyfikować informacje na temat zwierząt.
- Zwierzęta można przypisywać do wybiegu.
- Opiekunów można przypisywać do zwierząt
- Administrator może przypisać dowolnego opiekuna do zwierząt oraz opiekun samego siebie.
- Administrator może tworzyć konta pracownika lub administratora.
- Pracownik może dodawać komentarze na temat stanu podopiecznych.
- Pracownik może aktualizować swoje dane.
- Gość może pobierać dane o rodzajach zwierząt w zoo, ich ilości oraz gatunkach.

2.4. Wymagania niefunkcjonalne

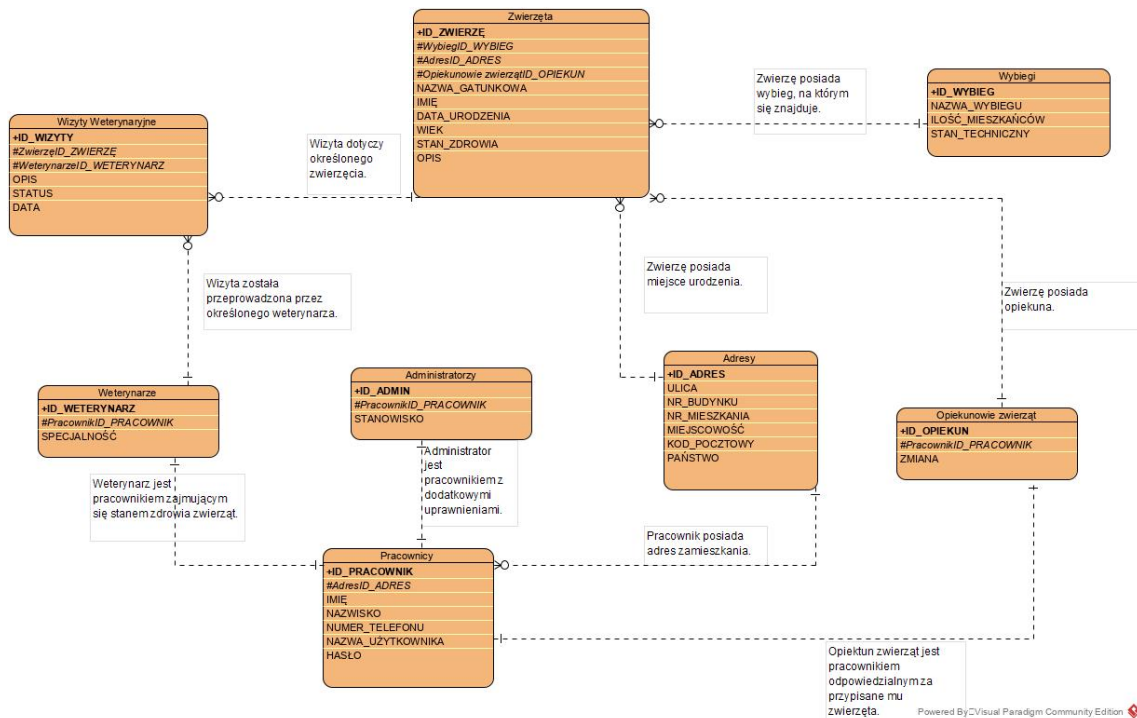
- Modyfikacja informacji o zwierzętach może odbywać się tylko przez pracownika lub moderatora.
- Korzystanie z systemu odbywa się przez aplikację internetową.
- Zwierze musi mieć co najmniej jednego opiekuna na jednej zmianie.
- System bazodanowy PostgreSQL.
- Język programowania JavaScript z frameworkiem Express.js.
- Analiza wielkości bazy danych: W naszej bazie danych przewiduje się że najwięcej instancji będzie w tabeli zwierząt. Z kolei liczebność tabeli pracownicy będzie kilkukrotnie mniejsza od zwierząt, lecz wprost proporcjonalna do niej. Wpływa na to zależność, że na jednego opiekuna będzie przypadać ok 10 osobników.

3. Projekt systemu

3.1. Projekt bazy danych

3.1.1. Model logiczny

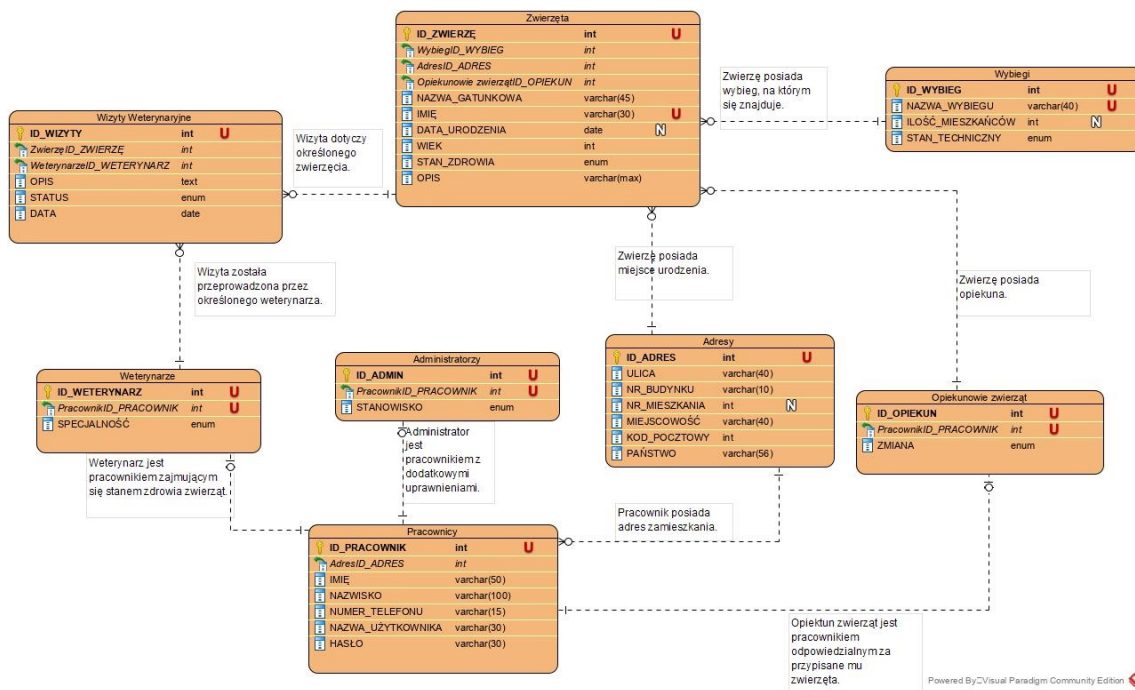
Model bazy danych w PostgreSQL.



Rysunek 1: Model logiczny

3.1.2. Model fizyczny i koncepcyjny

Model bazy danych w PostgreSQL.



Rysunek 2: Model fizyczny i konceptualny

3.1.3. Inne elementy schematu - mechanizmy przetwarzania danych

Planowane indeksy w bazie danych:

- tabela *Adresy*, kolumna *id_Adres*,
- tabela *Adresy*, kolumny *państwo*, *miescowość*,
- tabela *Pracownik*, kolumna *id_pracownik*,
- tabela *Pracownik*, kolumna *nazwa użytkownika*,
- tabela *Wizyty Weterynaryjne*, kolumna *data*,

3.1.4. Zabezpieczenia na poziomie bazy danych

Głównym sposobem zabezpieczenia bazy danych będzie okresowe tworzenie kopii zapasowych, co pozwoli na zminimalizowanie strat spowodowanych np. zdropowaniem całej bazy danych.

3.1.5. Widoki

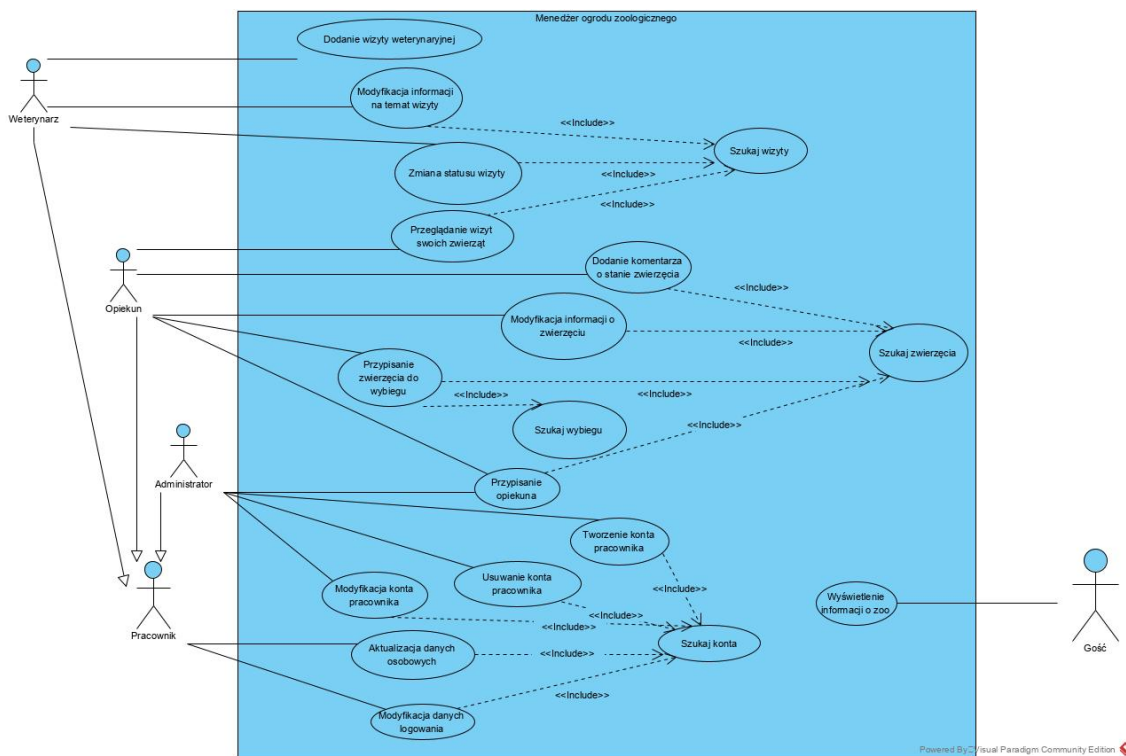
Baza będzie udostępniała następujące konceptualne widoki do aplikacji.

- Widok `animal_places_impletion`, mówiący nam o aktualnym zapelnieniu każdego z wybiegów.
- Widok `animal_places_count`, mówiący nam o aktualnej ilości wybiegów na terenie zoo
- Widok `vets_count`, mówiący nam o ilości lekarzy na terenie zoo,
- Widok `visits_fulfilled_count`, mówiący nam i ilości przeprowadzonych wizyt.
- Widok `workers_count`, mówiący nam o ilości wszystkich pracowników na terenie zoo,
- Widok `caretakers_count`, mówiący nam o ilości opiekunów do zwierząt na terenie zoo,
- Widok `animals_count`, mówiący nam o ilości zwierząt na terenie zoo,
- Widok `animals_species_count`, mówiący nam o ilości gatunków zwierząt występujących na terenie zoo, ————
- Informacje o zwierzęciu o podanym ID. Będzie on zawierał dane z tabel Zwierzęta (nazwa gatunkowa, imię, data urodzenia, stan zdrowia), Pracownicy (imię, nazwisko), Adresy (państwo) - dla zwierzęcia.
- Informacje na temat zdrowia zwierzęcia. Wyświetla on wszystkie informacje o wizytach weterynaryjnych dla podanego ID zwierzęcia, opis wizyty, datę wizyty oraz gatunek i stan zdrowia z tabeli Zwierzęta.
- Informacje o pracowniku z tabeli Pracownicy (imię, nazwisko, dodatkowo - numer telefonu i wszystkie dane, dla podanego pracownika, z tabeli Adresy w wersji rozbudowanej) oraz podaje jego profesję na podstawie tabel Weterynarze, Administratorzy, Opiekunowie zwierząt.
- Widok użytkowy dla weterynarzy. Widok o nazwie `animal_health`, zawierający id zwierzęcia oraz stan zdrowotny,

3.2. Projekt aplikacji użytkownika

Link do mock-up'ów: [Projekt](#)

3.2.1. Architektura aplikacji i diagramy projektowe



Rysunek 3: Diagram przypadków użycia

3.2.2. Projekt wybranych funkcji systemu

Przypisywać zwierzę do wybiegu będzie mógł pracownik-opiekun, który będzie wyszukiwał w bazie danych zwierzę a następnie otrzyma listę dostępnych miejsc na poszczególnych wybiegach. Jeśli wybieg będzie odpowiedni dla wybranego zwierzęcia, to nastąpi zmiana ID_WYBIEGU w tabeli Zwierzę. W sytuacji braku miejsc na wybiegach należy niezwłocznie zgłosić się do administracji.

Każdy pracownik posiada funkcję **Aktualizacji danych osobowych**. W celu aktualizacji danych konta pracownik musi najpierw zalogować się na swoje konto, a następnie wprowadzić nowe informacje. Jeśli operacja się powiedzie, odpowiednie rekordy bazy danych zostaną aktualizowane. W razie niepowodzenia, zostanie zwrócony komunikat o błędzie.

W tabeli Zwierzę istnieje możliwość wpisania ID_PRACOWNIK, który jest opiekunem zwierząt. Z tego powodu przy wpisywaniu, musi być sprawdzane czy dany ID jest zawarty w tabeli Opiekun zwierząt. Na podobnej zasadzie, powinno być sprawdzane, czy ID_PRACOWNIK wpisywany w Wizyty Weterynaryjne jest w tabeli Weterynarze.

3.2.3. Metoda połączenia do bazy danych

Aplikacja będzie napisana przy użyciu frameworka Express.js. Aby umożliwić łączenie się z bazą danych wykorzystamy interfejs node-postgres, który jest zbiorem modułów Node.js do obsługi bazy danych PostgreSQL.

3.2.4. Zabezpieczenia na poziomie aplikacji

Aplikacja będzie zabezpieczona panelem logowania. To utrudni osobom nieuprawnionym korzystanie z funkcji aplikacji.

Konta użytkowników będą posiadały różne stopnie dostępu do bazy danych. Dzięki temu użytkownik będący opiekunem zwierząt nie będzie w stanie dodawać lub usuwać konta pracownika (jest to funkcja administratora). Na podobnej zasadzie administrator lub opiekun zwierząt nie będzie w stanie edytować informacji na temat wizyty weterynaryjnej.

Po stronie aplikacji będzie odbywała się także walidacja danych.

4. Implementacja systemu baz danych

4.1. Tworzenie tabel i definiowanie ograniczeń

```
CREATE TABLE Administrator(  
    admin_id INT GENERATED ALWAYS AS IDENTITY,  
    TYPE admin_type NOT NULL,  
    worker_id INTEGER NOT NULL,  
    FOREIGN KEY (worker_id) REFERENCES worker (worker_id)  
);
```

```
CREATE TYPE veterinary_specialty AS ENUM(  
    'Choroby koni',  
    'Choroby psów i kotów',  
    'Choroby drobiu oraz ptaków ozdobnych',  
    'Choroby zwierząt futerkowych',  
    'Choroby ryb',  
    'Choroby owadów użytkowych',  
    'Choroby zwierząt nieudomowionych',  
    'Chirurgia weterynaryjna',  
    'Weterynaryjna diagnostyka laboratoryjna');
```

```
CREATE TABLE vet(  
    vet_id INTEGER GENERATED ALWAYS AS IDENTITY,  
    vet_specialty veterinary_specialty NOT NULL,  
    worker_id INTEGER NOT NULL,  
    FOREIGN KEY (worker_id) REFERENCES worker (worker_id)  
);
```

```
CREATE TYPE place_condition AS ENUM(  
    'bardzo slaby',  
    'slaby',  
    'przecietny',  
    'dobry',  
    'bardzo dobry');
```

```
CREATE TABLE animal_place(  
    place_id INTEGER GENERATED ALWAYS AS IDENTITY,  
    animal_count INTEGER DEFAULT 0,  
    place place_condition NOT NULL  
);
```

```

CREATE TYPE health_condition AS ENUM(
    'worst',
    'bad',
    'unhealthy',
    'promising',
    'healthy');

CREATE TYPE shift AS ENUM(
    'morning',
    'evening',
    'night');

CREATE TABLE caretaker(
    caretaker_id INTEGER GENERATED ALWAYS AS IDENTITY,
    shift shift NOT NULL,
    worker_id INTEGER NOT NULL,

    FOREIGN KEY(worker_id) REFERENCES worker(worker_id)
);

CREATE TABLE animal(
    animal_id INTEGER GENERATED ALWAYS AS IDENTITY,
    species VARCHAR(4) NOT NULL,
    name VARCHAR(30) NOT NULL,
    birth_date DATE,
    health health_condition NOT NULL,
    place_id INTEGER NOT NULL,
    caretaker_id INTEGER NOT NULL,
    address_id INTEGER NOT NULL,
    FOREIGN KEY(caretaker_id) REFERENCES caretaker(caretaker_id),
    FOREIGN KEY(address_id) REFERENCES address(address_id),
    FOREIGN KEY(place_id) REFERENCES animal_place(place_id)
);

CREATE TYPE visit_state AS ENUM(
    'pending',
    'in progress',
    'finished');

CREATE TABLE vet_visits(
    visit_id INTEGER GENERATED ALWAYS AS IDENTITY,
    vet_id INTEGER NOT NULL,

```

```

description VARCHAR NOT NULL,
visit_state visit_state NOT NULL,
visit_date DATE NOT NULL,
animal_id INTEGER NOT NULL,
FOREIGN KEY(animal_id) REFERENCES animal(animal_id),
FOREIGN KEY(vet_id) REFERENCES vet(vet_id)
);

CREATE TABLE worker(
worker_id INTEGER GENERATED ALWAYS AS IDENTITY,
firstname VARCHAR(30) NOT NULL,
lastname VARCHAR(60) NOT NULL,
phoneNumber VARCHAR(12) NOT NULL,
username VARCHAR(30) UNIQUE NOT NULL,
worker_password VARCHAR(30) NOT NULL,
address_id INTEGER NOT NULL,
PRIMARY KEY(person_id),
FOREIGN KEY(address_id) REFERENCES address(id_address)
);

CREATE TABLE address(
id_address INTEGER PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
street VARCHAR(40) NOT NULL,
building_number INTEGER NOT NULL,
flat_number INTEGER NULL,
city VARCHAR(85) NOT NULL,
post_code VARCHAR(10) NOT NULL,
country VARCHAR(75) NOT NULL
);

```

4.2. Implementacja mechanizmów przetwarzania danych

4.2.1. Procedury

Procedura określenia typu konta.

```

CREATE PROCEDURE get_account_type(idw INTEGER)
LANGUAGE sql
AS $$
SELECT w.worker_id, ad.admin_id, ca.caretaker_id, v.vet_id FROM workers AS w
LEFT JOIN administrators AS ad on w.worker_id = ad.worker_id

```

```

LEFT JOIN caretakers AS ca ON w.worker_id = ca.worker_id
LEFT JOIN vets AS v ON w.worker_id = v.worker_id
WHERE w.worker_id = idw;
$$;

```

4.2.2. Widoki

Widok informacji o pracowniku.

```

CREATE VIEW worker_details AS
SELECT w.worker_id,
       w.firstname,
       w.lastname,
       w.phonenumber,
       w.username,
       ad.city,
       ad.street,
       ad.building_number,
       ad.flat_number,
       ad.country,
       am.position,
       c.shift,
       v.vet_specialty
FROM workers AS w
LEFT JOIN addresses AS ad ON w.address_id = ad.address_id
LEFT JOIN administrators AS am ON w.worker_id = am.worker_id
LEFT JOIN caretakers AS c ON w.worker_id = c.worker_id
LEFT JOIN vets AS v ON w.worker_id = v.worker_id;

```

Widok informacji zwierzęciu.

```

CREATE VIEW animal_info AS
SELECT a.animal_id,
       a.species,
       a.name,
       a.birth_date,
       a.age,
       a.health,
       c.caretaker_id,
       w.firstname,

```

```

w.lastname,
ad.country
FROM animals AS a
  LEFT JOIN caretakers AS c ON a.caretaker_id = c.caretaker_id
  LEFT JOIN workers AS w ON c.worker_id = w.worker_id
  LEFT JOIN addresses AS ad ON ad.address_id = a.address_id;

```

Widok informacji o stanie zdrowia zwierzęcia.

```

\label{animalHealthView}
CREATE VIEW public.animal_health AS
  SELECT animals.animal_id,
         animals.health AS health_condition
  FROM public.animals;

```

4.2.3. Wyzwalacze

```

CREATE TRIGGER animal_insert_trigger AFTER INSERT ON animals
FOR EACH ROW EXECUTE PROCEDURE calculate_animal_age();

CREATE OR REPLACE FUNCTION calculate_animal_age() RETURNS TRIGGER AS $age$
BEGIN
  UPDATE animals
  SET age = date_part('year',age(NEW.birth_date))
  WHERE animal_id = NEW.animal_id;
  RETURN NEW;
END;
$age$ LANGUAGE plpgsql;

```

4.2.4. Indeksy

```

CREATE INDEX id_address_index ON addresses(address_id)
CREATE INDEX country_city_index ON addresses(country, city)
CREATE INDEX worker_id_index ON workers(worker_id)
CREATE INDEX worker_usernames_index ON workers(username)
CREATE INDEX vet_visits_date_index ON vet_visits(visit_date);

```


4.3. Implementacja uprawnień i innych zabezpieczeń

Role:

```
CREATE ROLE administrators WITH LOGIN PASSWORD 'role_admin' ;
CREATE ROLE workers WITH LOGIN PASSWORD 'role_workers' ;
CREATE ROLE vets WITH LOGIN PASSWORD 'role_vets';
CREATE ROLE caretakers WITH LOGIN PASSWORD 'role_guests' ;
CREATE ROLE guests WITH LOGIN PASSWORD 'role_guests' ;
```

Użytkownicy:

```
CREATE USER vet_user PASSWORD 'user_vet' ROLE vets;
CREATE USER admin_user PASSWORD 'user_admin' ROLE administrators;
CREATE USER caretaker_user PASSWORD 'user_caretaker' ROLE caretakers;
```

Uprawnienia:

- wszyscy

```
CONNECT
USAGE ON SCHEMA
LOGIN
PASSWORD
```

- goście

```
GRANT SELECT
ON TABLE animals, animal_places
TO guest;
```

- weterynarze

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON TABLE workers, addresses, animals, vet_visits, animal_health, vets
TO vets;
GRANT VIEW ON TABLE animal_info TO vets;
```

- opiekunowie

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON TABLE addresses, animals, caretakers
TO caretakers;
```

- administratorzy

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON ALL TABLES IN SCHEMA public
TO administrators;
```

4.3.1. Testowanie uprawnień

Do sprawdzenia poprawności działania uprawnień posłużyliśmy się terminalem `psql`.

```
SQL Shell (psql)
Server [localhost]:
Database [postgres]: zoodatabase
Port [5432]:
Username [postgres]: caretakers
Password for user caretakers:
psql (12.1)
WARNING: Console code page (852) differs from Windows code page (1250)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

zoodatabase=> SELECT * FROM vets;
BŁĄD: permission denied for table vets
zoodatabase=> SELECT * FROM vet_visits;
BŁĄD: permission denied for table vet_visits
zoodatabase=> SELECT * FROM administrators;
BŁĄD: permission denied for table administrators
zoodatabase=>
```

Rysunek 4: Zalogowanie przy użyciu konta *caretakers*

Na powyższym obrazie(4) widać sposób w jaki łączymy się z bazą. Nasza baza danych postawiona jest na lokalnym komputerze dlatego połączenie będzie się odwoływać do `localhost`. Następnie musimy podać wszystkie dane logowanie i informacje o bazie danych do której chcemy się podłączyć.

Po zalogowaniu do bazy danych, wywołane zostały trzy skrypty SQL, sprawdzające możliwość wyświetlenia danych do których użytkownik nie ma pozwolenia. Jak widać wywołanie zapytania nie powiodło się zgodnie z założeniem.

Kolejno sprawdziliśmy możliwość edycji widoku, bez uprawnienia do edycji tabeli, na której widok ten bazuje (widok *animal_health*). Celem nadania takiego uprawnienia było ograniczenie dostępu użytkownikowi *vets* do tabeli *animals* nie pozbawiając go celu w jakim został stworzony, czyli zmiana stanu zdrowia zwierząt.

Jak widać na obrazku 5, uprawnienia zadziałały pomyślnie.

```
SQL Shell (psql)
Server [localhost]:
Database [postgres]: zoodatabase
Port [5432]:
Username [postgres]: vets
Password for user vets:
psql (12.1)
WARNING: Console code page (852) differs from Windows code page (1250)
8-bit characters might not work correctly. See psql reference
page "Notes for Windows users" for details.
Type "help" for help.

zoodatabase=> Select * from animals LIMIT 1;
 animal_id |      species      | name | birth_date | health | age | place_id | caretaker_id | address_id
-----+-----+-----+-----+-----+-----+-----+-----+-----
      10 | Bird, bare-faced go away | Phedra | 2016-07-30 | fair   | 3   | 3         | 16            | 65688
(1 row)

zoodatabase=> UPDATE animals SET "name"='abc' WHERE animal_id=10;
BŁĄD: permission denied for table animals
zoodatabase=>
zoodatabase=> SELECT * FROM animal_health WHERE animal_id=10;
 animal_id | health_condition
-----+-----
      10 | fair
(1 row)

zoodatabase=> UPDATE animal_health SET health_condition='very_good' WHERE animal_id=10;
UPDATE 1
zoodatabase=> Select * from animals WHERE animal_id=10;
 animal_id |      species      | name | birth_date | health | age | place_id | caretaker_id | address_id
-----+-----+-----+-----+-----+-----+-----+-----+-----
      10 | Bird, bare-faced go away | Phedra | 2016-07-30 | very_good | 3   | 3         | 16            | 65688
(1 row)

zoodatabase=>
```

Rysunek 5: Testowanie uprawnień użytkownika *vets* do widoku *animal_health* oraz tabeli *animals*

Na poniższym rysunku(6) sprawdzono odczyt tabeli *administrators* oraz możliwości dodania do niej nowego administratora jak i usunięcie konkretnego adresu z tabeli *addresses* przy użyciu konta administratora.

```
SQL Shell (psql)
Server [localhost]:
Database [postgres]: zoodatabase
Port [5432]:
Username [postgres]: administrators
Password for user administrators:
psql (12.1)
WARNING: Console code page (852) differs from Windows code page (1250)
8-bit characters might not work correctly. See psql reference
page "Notes for Windows users" for details.
Type "help" for help.

zoodatabase=> SELECT * FROM administrators LIMIT 1;
 admin_id | type | worker_id
-----+-----+-----
(0 rows)

zoodatabase=> SELECT * FROM workers LIMIT 1;
 worker_id | firstname | lastname | phonenumber | username | worker_password | address_id
-----+-----+-----+-----+-----+-----+-----
          2 | Wittie    | Carnilian | 448 842 9044 | wcarnilian0 | LntGN6y         |        65688
(1 row)

zoodatabase=> INSERT INTO administrators(type,worker_id) VALUES('manager',2);
INSERT 0 1
zoodatabase=> SELECT * FROM administrators LIMIT 1;
 admin_id | type | worker_id
-----+-----+-----
          2 | manager |          2
(1 row)

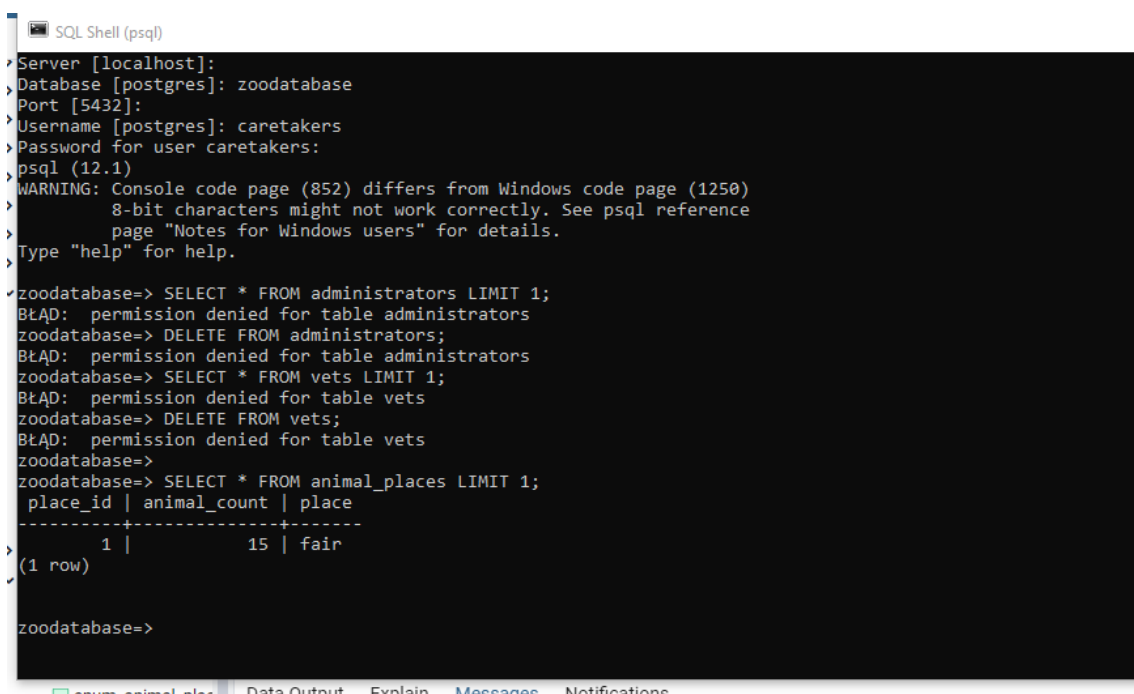
zoodatabase=> SELECT * FROM addresses where address_id=70000;
 address_id | street | building_number | flat_number | city | post_code | country
-----+-----+-----+-----+-----+-----+-----
        70000 | Lakewood |          52 |          90 | Zarzis | 1234 | Tunisia
(1 row)

zoodatabase=> DELETE FROM addresses WHERE address_id=70000;
DELETE 1
zoodatabase=> SELECT * FROM addresses where address_id=70000;
 address_id | street | building_number | flat_number | city | post_code | country
-----+-----+-----+-----+-----+-----+-----
(0 rows)

zoodatabase=>
```

Rysunek 6: Testowanie uprawnień użytkownika *administrators*

Na rysunku 7 zostały przedstawione odpowiedzi z bazy danych, na wykonanie zapytań SELECT, DELETE przez użytkownika *caretakers*.



```
SQL Shell (psql)
> Server [localhost]:
> Database [postgres]: zoodatabase
> Port [5432]:
> Username [postgres]: caretakers
> Password for user caretakers:
psql (12.1)
> WARNING: Console code page (852) differs from Windows code page (1250)
> 8-bit characters might not work correctly. See psql reference
> page "Notes for Windows users" for details.
> Type "help" for help.
>
zoodatabase=> SELECT * FROM administrators LIMIT 1;
Błąd: permission denied for table administrators
zoodatabase=> DELETE FROM administrators;
Błąd: permission denied for table administrators
zoodatabase=> SELECT * FROM vets LIMIT 1;
Błąd: permission denied for table vets
zoodatabase=> DELETE FROM vets;
Błąd: permission denied for table vets
zoodatabase=>
zoodatabase=> SELECT * FROM animal_places LIMIT 1;
 place_id | animal_count | place
-----+-----+-----
1 | 15 | fair
(1 row)

zoodatabase=>
```

Rysunek 7: Testowanie uprawnień użytkownika *caretakers*

4.4. Testowanie bazy danych na przykładowych danych

4.4.1. Wstęp

Systemem zarządzania bazą danych jaką użyliśmy do naszego projektu jest *PostgreSQL*. Jest to, obok *MySQL* oraz *SQLite* jeden z trzech najpopularniejszych systemów do zarządzania. Należy on do RDBMS, czyli do systemów zarządzania relacyjną bazą danych. W naszym projekcie korzystamy z wersji 12.0.

Oprócz SZBD korzystamy także z *psql*, czyli interaktywnego terminala postgres oraz z programu do zarządzania i zwykłego użytkowania bazą danych *pgAdmin* w wersji 4.3.1.

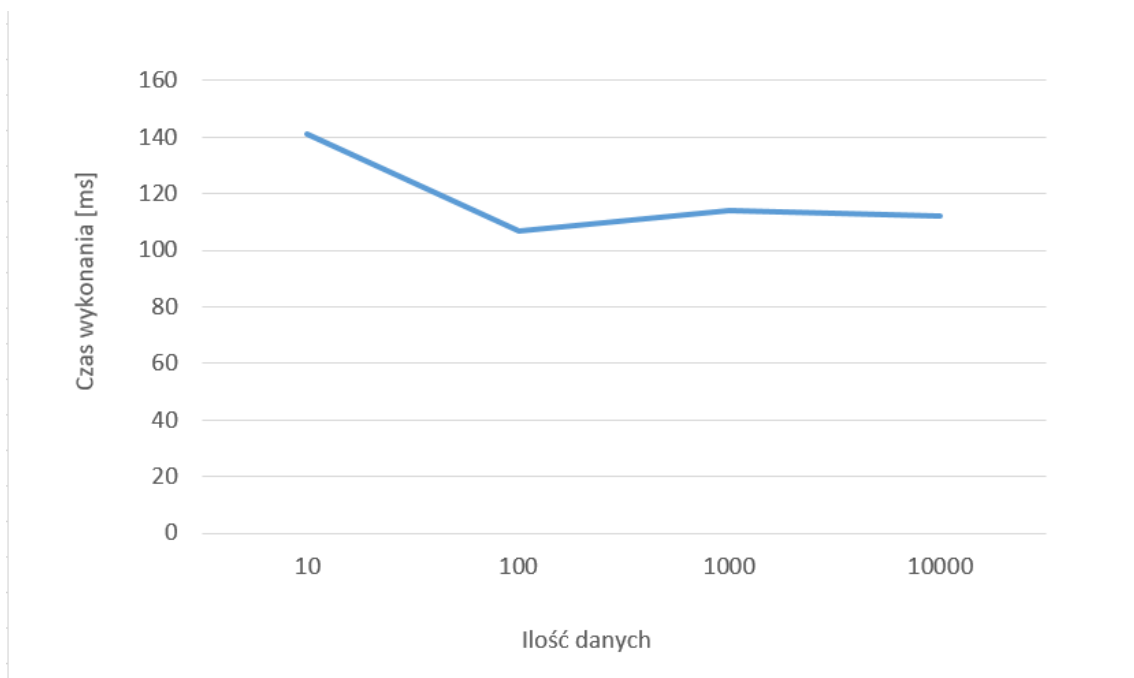
4.5. Sposoby generowania danych

Dane zostały wygenerowane przy użyciu odpowiedniego internetowego narzędzia na <https://mockaroo.com/>. Dodatkowo, w niektórych przypadkach musiały zostać zmienione ze względu na niespełnianie kryteriów poprawności.

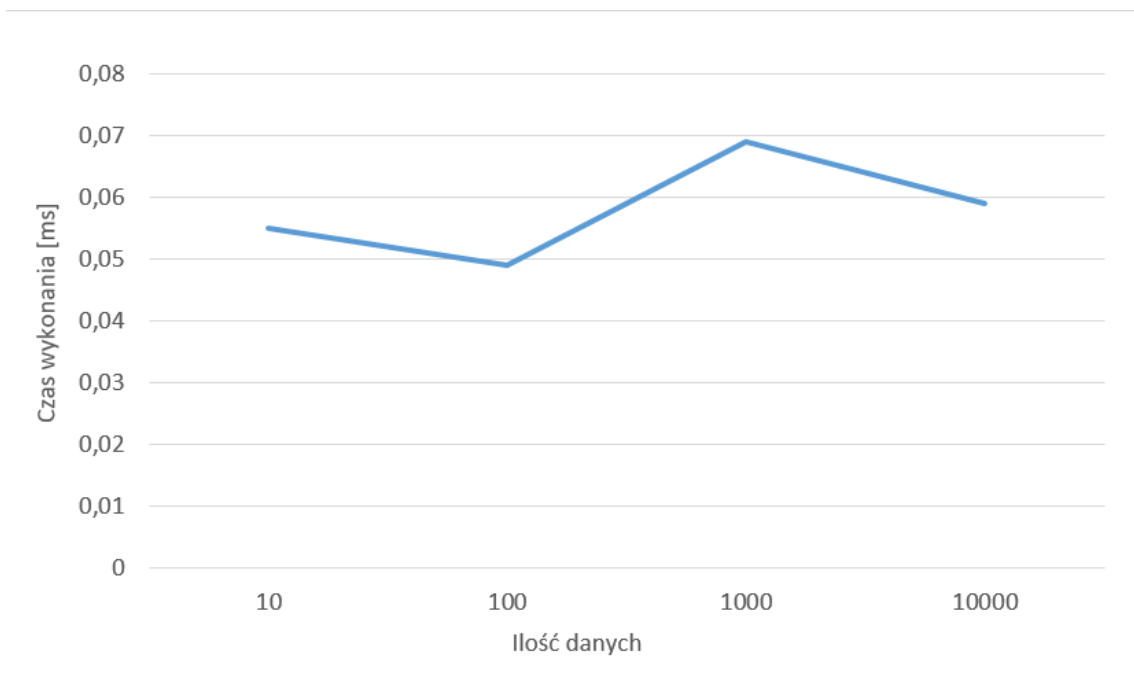
4.5.1. Testy wydajnościowe

Testowanie bazy danych opierało się na wykonywanie różnych skryptów SQL na N danych i mierzenia czasu ich wykonania. Wykonywane Skrypty:

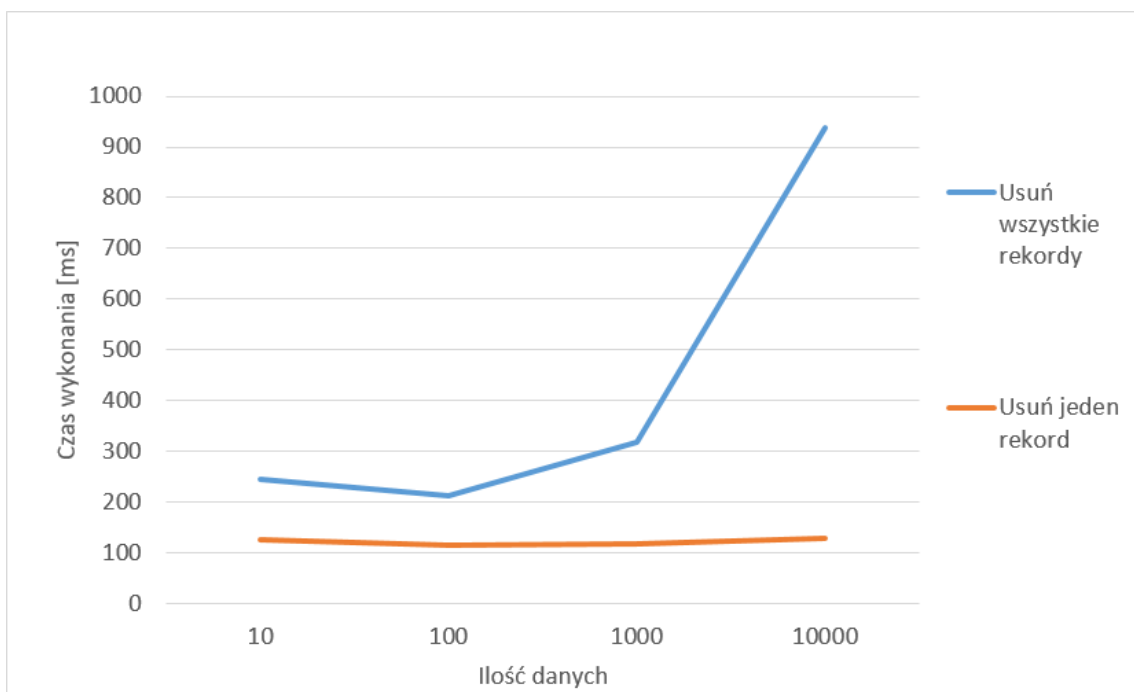
1. `SELECT * FROM addresses`
2. `DELETE FROM addresses`
3. `DELETE FROM addresses`
`WHERE address_id = (SELECT floor(random()*`
`(SELECT MAX(address_ID) FROM addresses));)`
4. `UPDATE public.addresses`
`SET street=?, building_number=?, flat_number=?, city=?, post_code=?,`
`country=?`
`WHERE address_id=?;`
5. `INSERT INTO addresses`
`(street, building_number, flat_number, city, post_code, country)`
`values (?, ?, ?, ?, ?, ?);`



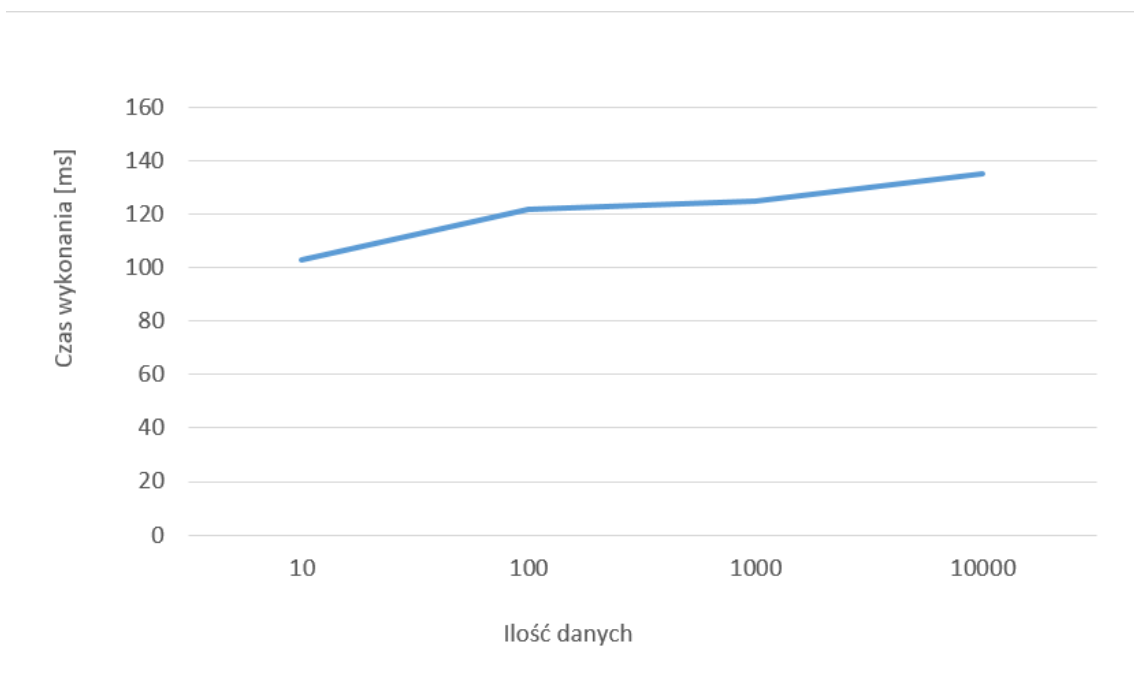
Rysunek 8: Wykres czasu wykonywania zapytania SELECT



Rysunek 9: Wykres czasu wykonania zapytania INSERT dla jednego rekordu



Rysunek 10: Wykres czasu wykonania zapytania DELETE dla dwóch przypadków



Rysunek 11: Wykres czasu wykonania zapytania UPDATE dla jednego zapytania

4.5.2. Omówienie danych

Testy wydajnościowe zostały przeprowadzone pomyślnie. Wszystkie wykonane zapytania zmieściły się w oczekiwanych ramach czasowych. W przypadku zapytania `SELECT(8)` można się spodziewać, że czas wykonania zapytania dla 10 rekordów byłby niższy. Wpływ na to zapewne mają warunki losowe, gdyż czas wykonania jednej kwerendy zazwyczaj różnił się od średniej około 50 ms.

Co więcej, zaskakujące są wyniki wykonania zapytań usunięcia, edycji oraz wyboru jednego rekordu dla 10000 danych. Przy takiej ilości danych można stwierdzić tendencję stałą wykonywanych operacji. Wpływ na to zapewne ma przemyślana architektura i dobór zoptymalizowanych algorytmów przy tworzeniu systemu PostgreSQL. Z wykresów można wywnioskować że wykonanie takich operacji jest stałe w czasie i dla wszystkich zapytań wynosi około 100-120ms.

W przypadku dodania nowego rekordu do bazy danych, czas takiej operacji jest bardzo mały w porównaniu do poprzednich zapytań. Jego wyniki czasowe oscylują w granicach 0,06ms. Do wykonania takiej operacji nie jest wcale potrzebne przeszukiwanie całego zakresu, a jedynie dołączenie go do tabeli. Jedyną operacją mogącą spowolnić jego działanie jest skomplikowana walidacja danych. Aczkolwiek w przy naszych eksperymentach jedyne do sprawdzenia była instrukcja `NOT NULL` dla wszystkich kolumn, co nie znacząco wpłynęło na wynik.

Podsumowując, system zarządzania relacyjną bazą danych PostgreSQL jest znakomitym systemem, gdyż charakteryzuje się skalowalnością. W przypadku małych oraz dużych baz danych zachowuje te same złożoności czasowe

5. Implementacja i testy aplikacji

5.1. Instalacja i konfigurowanie systemu

Ze względu na webową naturę naszej aplikacji, jedynym wymaganym narzędziem jest przeglądarka internetowa.

Dla administratora - serwer. Należy pobrać Node.js z oficjalnej strony oraz podążać za wyświetlanymi poleceniami. Po zakończeniu instalacji, należy sprawdzić jej skuteczność, wpisując w terminale polecenie `node -v`, które zwróci nam jego zainstalowaną wersję (może być konieczny restart komputera lub terminalu, jeśli był uruchomiony przed instalacją). Wraz z Node JS instaluje się NPM - domyślny manager pakietów dla środowiska Node.js. Jego wersję można sprawdzić poleceniem `npm -v`. Następnie należy zainstalować potrzebne

dla aplikacji paczki komendą `npm install`. Komendę należy wywołać w podfolderze z plikiem `package.json`. Wszystkie potrzebne paczki potrzebne do uruchomienia aplikacji znajdują się w tym pliku.

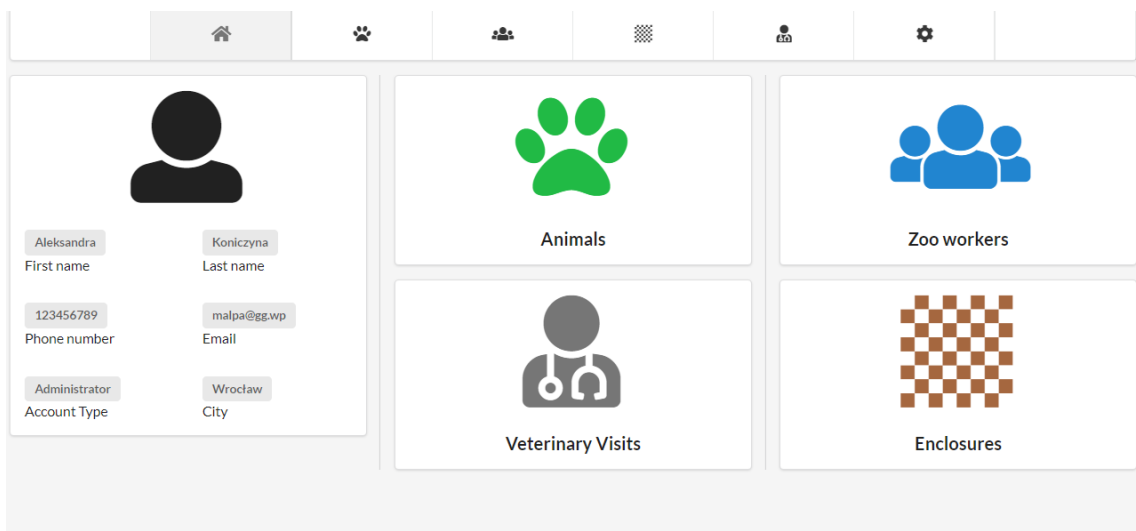
5.2. Instrukcja użytkowania aplikacji

1. Wejść na stronę aplikacji, o adresie podany przez administratora.
2. Na ekranie startowym, zalogować się za pomocą swojego hasła oraz nazwy użytkownika.

The image shows a web application's login interface. At the top, there is a horizontal navigation bar with icons for home, paw print, group of people, a grid, a person, a gear, and a 'Log Out' link. The main content area features a large green logo for 'ZOO WROCŁAW Sp. z o.o.' where the 'O's are stylized as animal silhouettes. Below the logo, there are two input fields: 'Username' with the text 'admin' and 'Password' with masked characters '.....'. A green 'Submit' button is positioned below the password field.

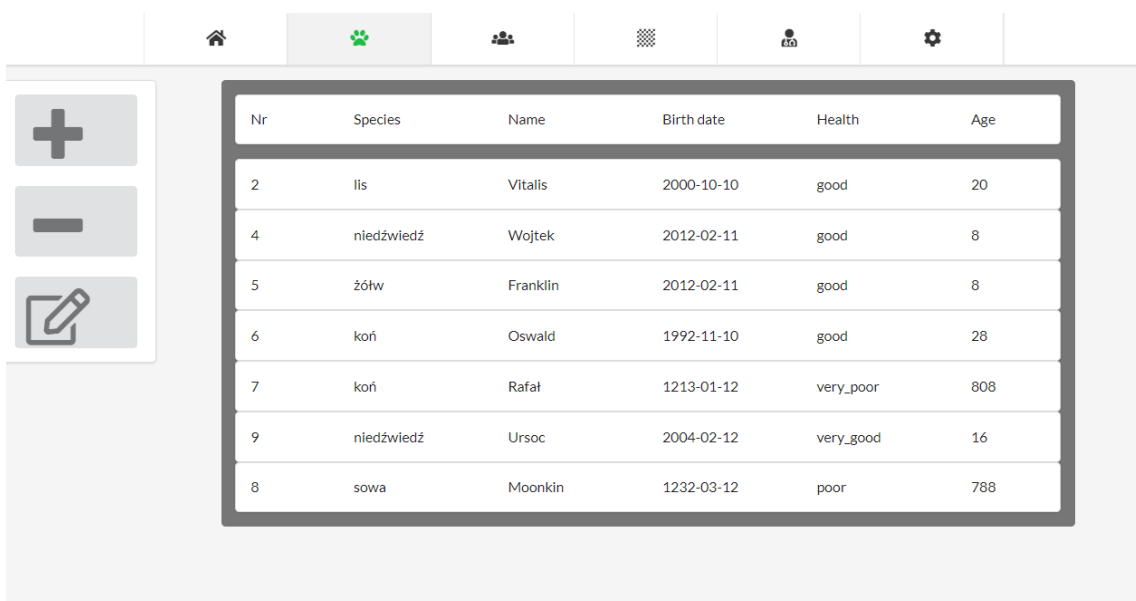
Rysunek 12: Ekran logowania

3. Po zalogowaniu ukazuje się panel użytkownika oraz pasek zakładek, które odpowiadają za wyświetlanie oraz wprowadzanie danych z/ do bazy danych. Dostęp do różnych interfejsów jest ograniczony na podstawie rodzaju konta użytkownika.



Rysunek 13: Ekran główny po zalogowaniu


4. Aby przejść do interesującej nas podstrony należy kliknąć na odpowiedni kafelek lub przycisk na pasku nawigacyjnym.



Rysunek 14: Zakładka Zwierzęta

5. W celu dodania nowego podmiotu do bazy należy nacisnąć przycisk z plusem.

Add New Animal



Name

Species

Health

Place id

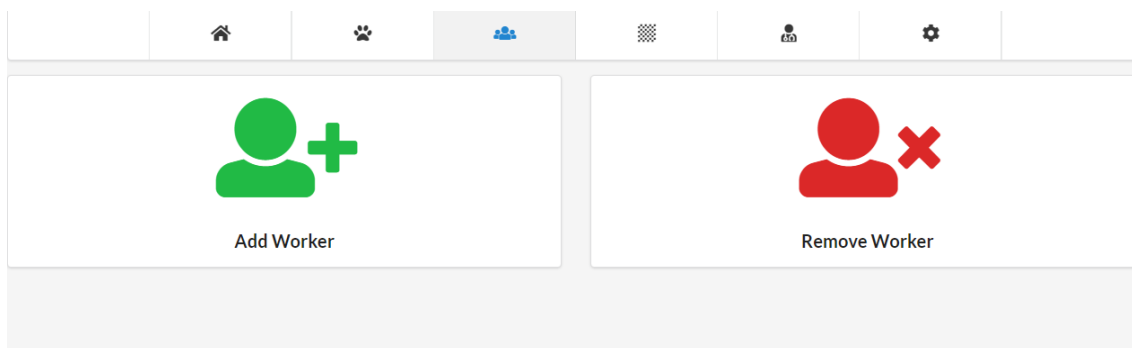
Day of birth

Month of birth

Year of birth

Rysunek 15: Okno modalne dodawania zwierzęcia


6. Otworzy się okno modalne, w którym należy wprowadzić dane.




Rysunek 16: Ekran administratora

7. Użytkownik posiadający typ konta *Administrator* dostaje dostęp do ekranu administratora. Może dodać konto użytkownika lub je usunąć klikając odpowiedni kafelek.

Add New User



 Account


Username

Username

Password

Password

Account Type

 Informations

First Name


First Name

Last Name

Last Name

Phonenumber

Phonenumber

 Address

Street

Street

Building Number

Building Nurr

Flat Number

Flat Number

City

City

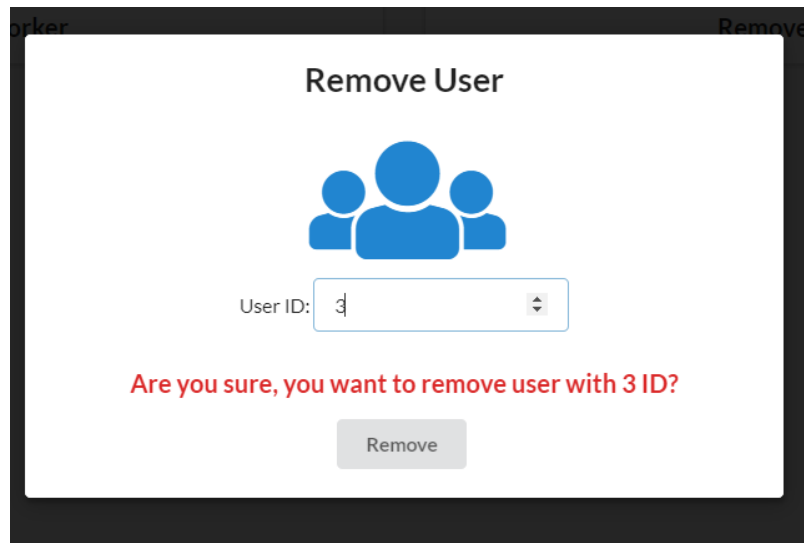
Post code

Post code

Country

Rysunek 17: Okno modalne dodania konta użytkownika

- Po wybraniu opcji "Dodaj pracownika" pokaże się powyższe okno modalne.



Rysunek 18: Okno modalne usunięcia konta użytkownika

9. W celu usunięcia pracownika należy wybrać opcję "Usuń pracownika" i w oknie modalnym wpisać odpowiedni numer identyfikacyjny.

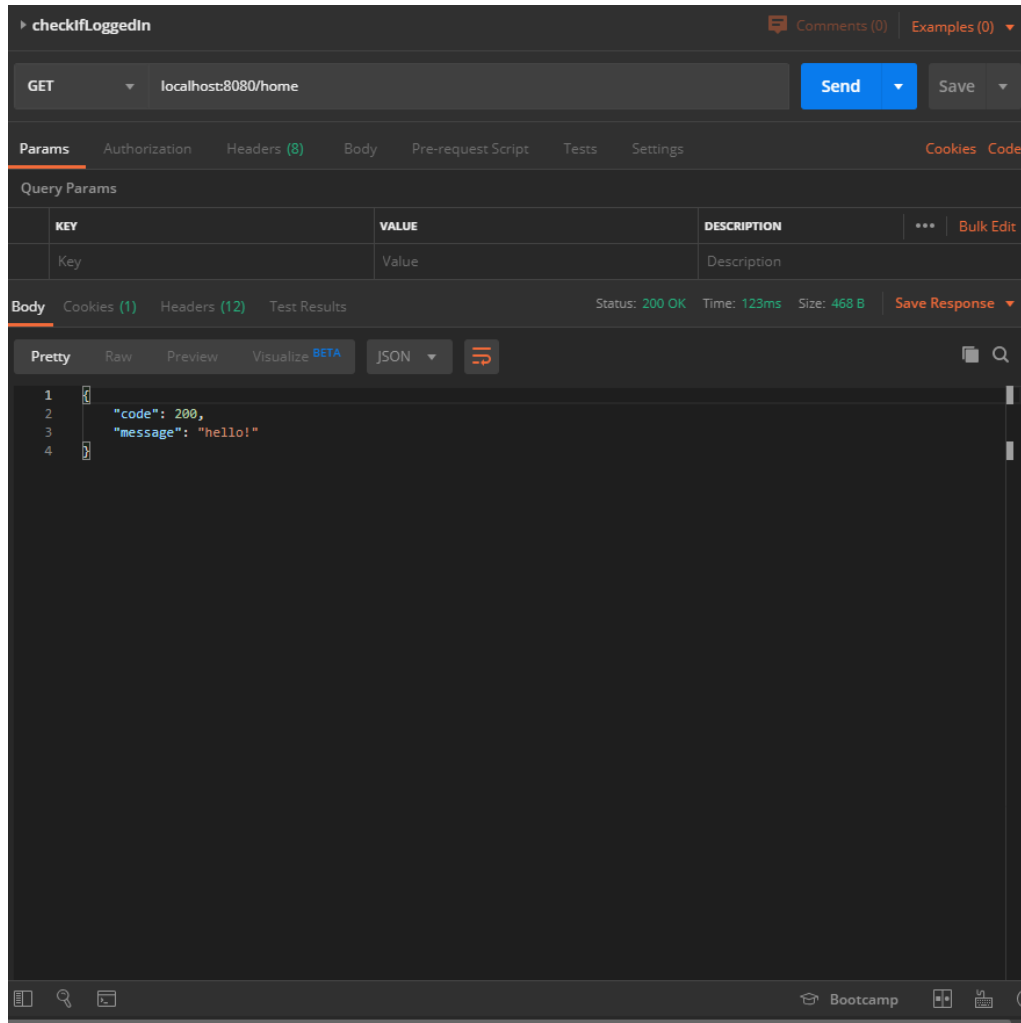
Rysunek 19: Ekran ustawień konta

10. Każdy użytkownik po kliknięciu zębátky na pasku nawigacyjnym może edytować swoje konto.

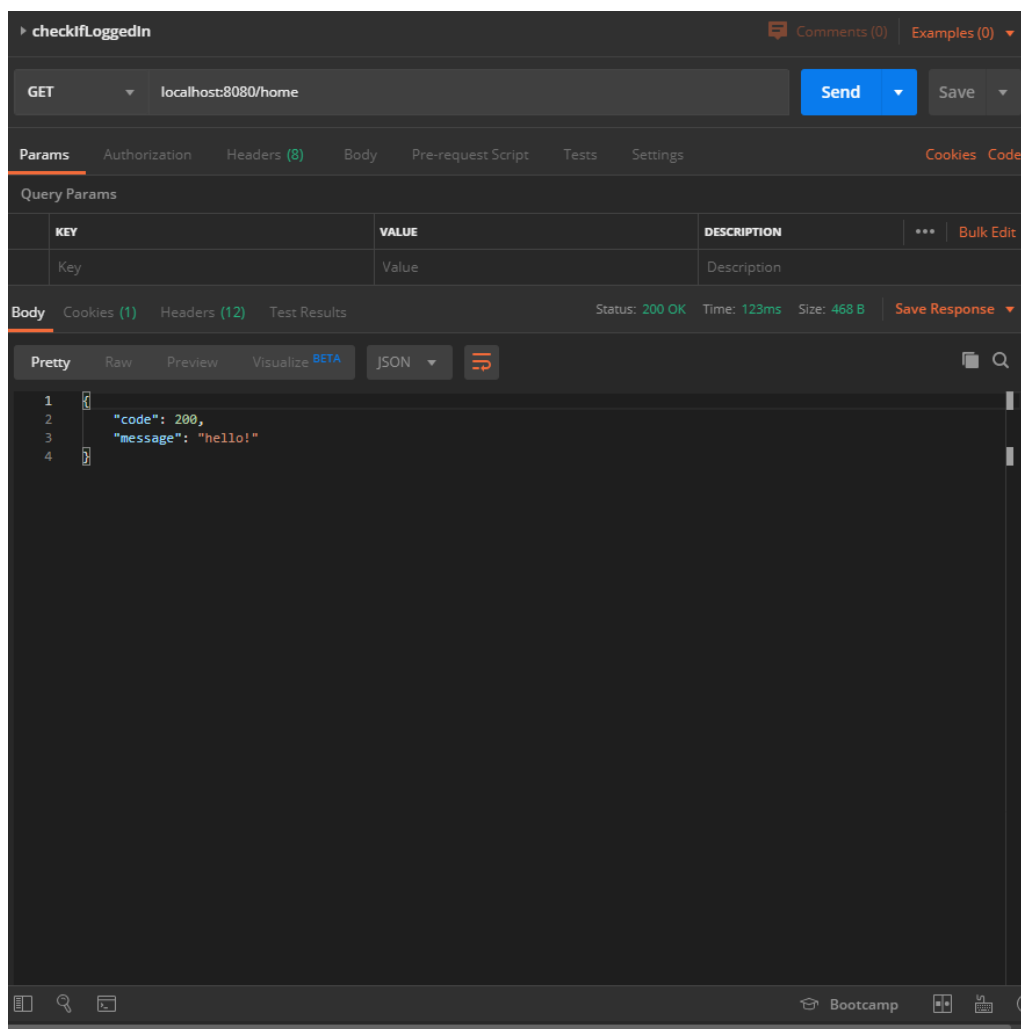
5.3. Testowanie opracowanych funkcji systemu

Aplikacja posiada szereg endpoint'ów pozwalających na wykonywanie różnych operacji CRUD na bazie danych.

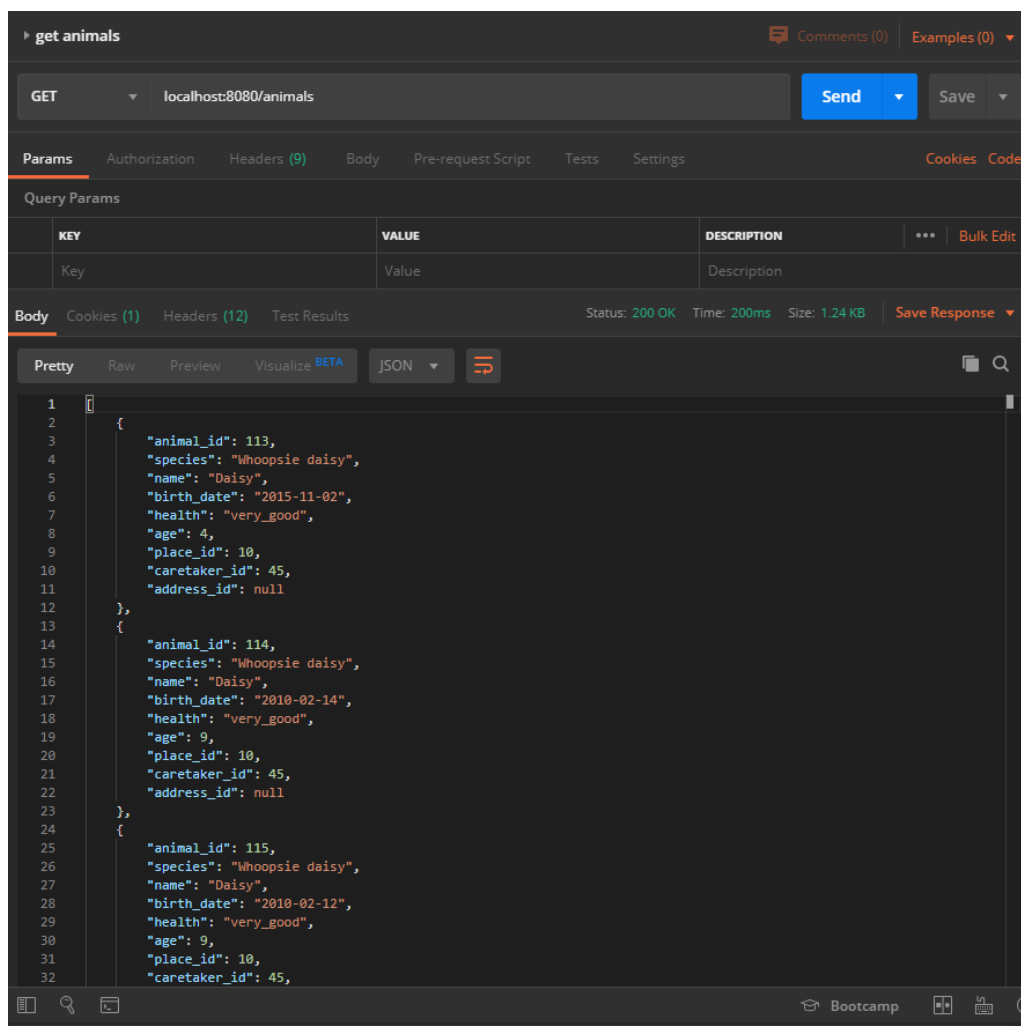
Do testowania opracowanych funkcji systemu posłużyliśmy się aplikacją Postman, która pomaga w sprawdzaniu funkcjonalności aplikacji z podejściem restowym.



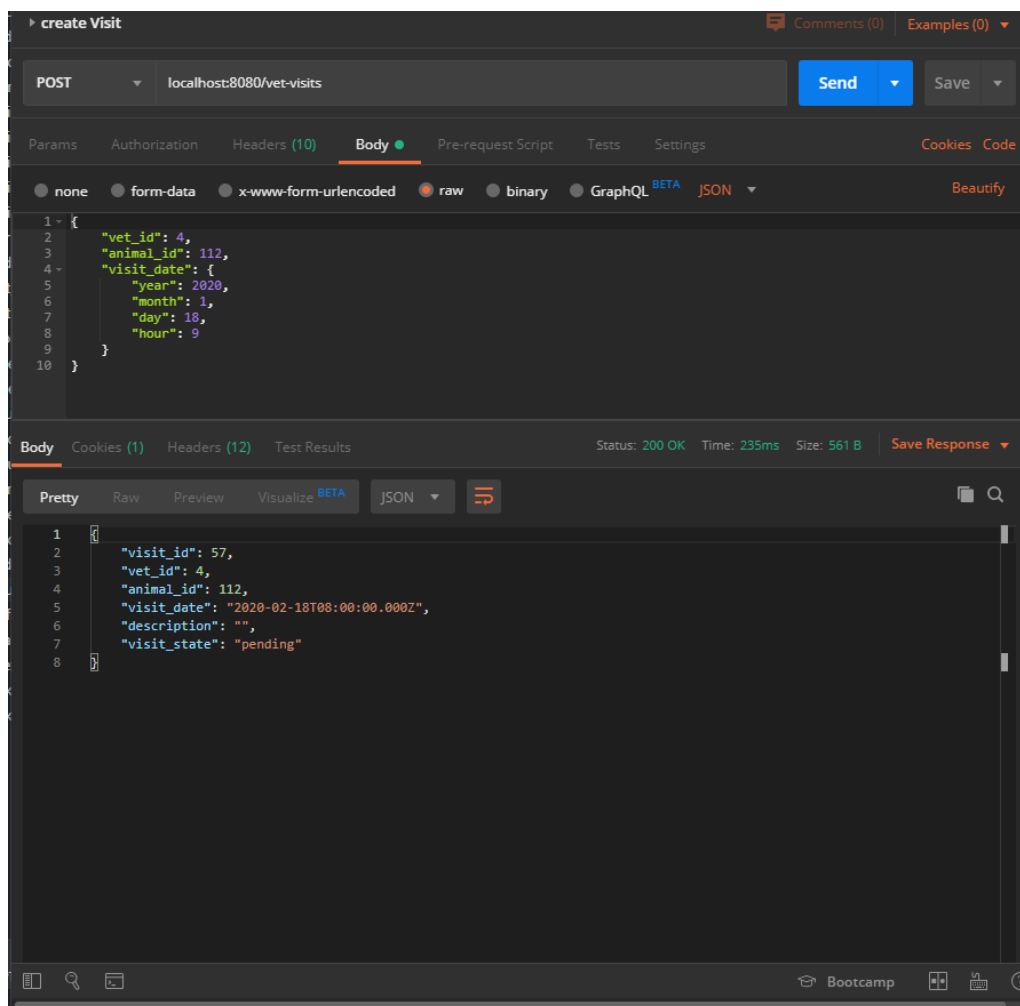
Rysunek 20: Powodzenie logowania



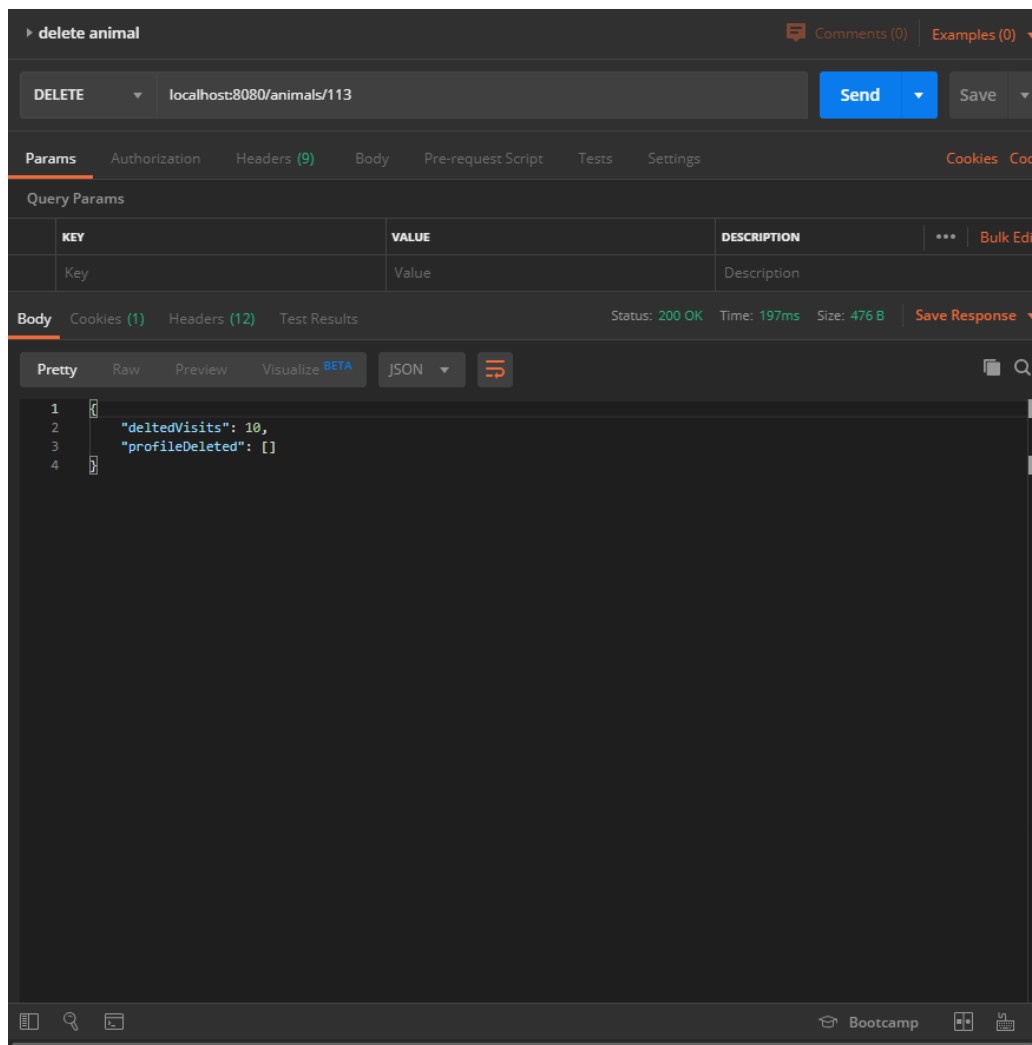
Rysunek 21: Sprawdzenie czy jest się zalogowanym



Rysunek 22: Sprawdzenie pobierania zwierząt zalogowanego opiekuna



Rysunek 23: Sprawdzenie możliwości utworzenia nowej wizyty przez opiekuna



Rysunek 24: Sprawdzenie usuwania zwierzęcia przez opiekuna

5.4. Omówienie wybranych rozwiązań programistycznych

5.4.1. Implementacja interfejsu dostępu do bazy danych

Dostęp do bazy danych odbywa się przez cztery rodzaje kont (w przypadku postgresa są to role). Każda z ról ma przypisane pozwolenia na dostęp i modyfikacje określonego zestawu danych. W przypadku implementacji dostępu do bazy danych po stronie aplikacji, w naszym backendzie zostały zapisane dane logowania w pliku `config.js`.

Listing 1: "Dane logowania do bazy danych"

```

1 const admins_role = {user:"administrators" ,password:"role_admin"};
2 const caretakers_role = {user:"caretakers" ,password:"
  role_caretakers"};

```

```

3 const vets_role = {user:"vets" ,password:"role_vets"};
4 const guests_role = {user:"guests" ,password:"role_guests"};
5
6 module.exports = {
7   database: "zoodatabase",
8   options:{
9     host:"localhost",
10    port: "5432",
11    dialect: "postgres",
12    logging: false,
13    define:{
14      timestamps:false
15    }
16  },
17  admins: admins_role,
18  caretakers: caretakers_role,
19  vets: vets_role,
20  guests: guests_role
21 }

```

W celach pobierania, modyfikacji oraz tworzenia nowych danych w bazie danych za pomocą języka JavaScript, posłużyliśmy się biblioteką ORM Sequelize. Wewnątrz tej biblioteki wykorzystywany jest interfejs node-postgres odpowiadający za komunikację z bazą danych postgres.

Listing 2: Przykładowe użycie biblioteki sequelize wewnątrz jednej z funkcji routingu biblioteki expressjs. Za dostęp do konkretnych danych odpowiadają modele, na których są wykonywane różne akcje takie jak na przykład findAll. Każdy taki model reprezentuje tabelę lub widok. Wartość raw przekazywana w parametrze odpowiada za zwrócenie surowych danych, nie obudowanych w funkcjonalności biblioteki.

```

1 module.exports.getAllAnimalPlaces = async function(req, res) {
2   const animalPlaces = await
3     req.sequelize.caretakers.models.animal_places.findAll(
4     { raw: true }
5   );
6   res.status(200).json(animalPlaces);
7 }

```

Listing 3: Przykładowa definicja modelu tabeli "Ópiekunowie". W linii 28 funkcja associate odpowiada za mapowanie relacji pomiędzy encjami.

```

1 module.exports = function(sequelize, DataTypes) {
2   let caretakers = sequelize.define(
3     'caretakers',
4     {
5       caretaker_id: {
6         type: DataTypes.INTEGER,
7         allowNull: false,
8         autoIncrement: true,
9         primaryKey: true

```

```

10         },
11         shift: {
12             type: DataTypes.ENUM('morning', 'evening', 'night')
13             ,
14             allowNull: false
15         },
16         worker_id: {
17             type: DataTypes.INTEGER,
18             allowNull: false,
19             primaryKey: true
20         },
21     },
22     {
23         tableName: 'caretakers'
24     }
25 );
26
27 caretakers.associate = (models) => {
28     caretakers.belongsTo(models.workers, { foreignKey: '
29         worker_id' });
30     caretakers.hasMany(models.animals, { foreignKey: 'animal_id
31         ' });
32 };
33
34 return caretakers;
35 };

```

5.4.2. Implementacja wybranych funkcjonalności systemu

W poniższym listingu zaimplementowana została funkcjonalność edycji własnych danych przez użytkownika. W liniach 2-24 pobierane są dane z body oraz z nagłówka http.

W linii 25 został wprowadzony try-catch w celu wyłapania błędów napotkanych na drodze komunikacji z bazą danych. W razie takiego wyjątku klient zostanie poinformowany o problemach serwera kodem 500. W liniach 26-34 za pomocą biblioteki sequelize pobrane zostają dane zwierzęcia, który jest pod opieką użytkownika wysyłającego dane żądanie do serwera. Dane opiekuna pobrane zostają z id sesji. W przypadku administratora wywoływana jest inna funkcja. W linii 36 sprawdzany jest zwrócony obiekt. Jeżeli nie zwrócono żadnego obiektu, wówczas oznacza to nieposiadanie w opiece danego zwierzęcia i zwrócony zostaje kod 404 (linia 48). W liniach 37-43 zostają zaktualizowane dane, a następnie za pomocą instrukcji update() wywołanej na obiekcie animalProfile, wysyłają zapytanie UPDATE do bazy danych. Jeżeli wszystko się powiedzie zostaje zwrócony kod 200, z odpowiedzią serwera (linia 46).

Listing 4: Implementacja funkcjonalności "Pracownik może aktualizować swoje dane

```

1 module.exports.updateAnimalProfile = async function(req, res) {
2   const ANIMAL_ID = req.params.id;
3   const userProfile = res.locals.user;
4   const {
5     species,
6     name,
7     birth_date,
8     health,
9     age,
10    place_id,
11    caretaker_id,
12    address_id
13  } = req.body;
14
15  const newProfileInfo = {
16    species,
17    name,
18    birth_date,
19    health,
20    age,
21    place_id,
22    caretaker_id,
23    address_id
24  };
25  try {
26    const animalProfile = await
27      req.sequelize.caretakers.models.animals.findOne(
28        {
29          where: {
30            caretaker_id:
31              userProfile.caretaker.caretaker_id,
32            animal_id: ANIMAL_ID
33          },
34          raw: false
35        }
36      );
37    if (animalProfile) {
38      for (let key in animalProfile.dataValues) {
39        if (key !== "animal_id" && key !== "health") {
40          animalProfile[key] =
41            newProfileInfo[key] || animalProfile[key];
42        }
43      }
44      animalProfile.address_id = 76475;
45      const response = await animalProfile.update();
46      res.status(200).json(response);
47    } else {
48      res.status(404).send("Caretaker does not have this
49        animal");
50    }
51  }
52 }

```

```

50     } catch (error) {
51         console.log("[ERROR]:", error.message);
52         res.status(500).send(error.message);
53     }
54 };

```

5.4.3. Implementacja mechanizmów bezpieczeństwa

Pierwszym z mechanizmów zastosowanych w naszej aplikacji jest hashowanie hasła za pomocą Base64. Jest to rodzaj kodowania ciągów bajtów za pomocą znaków.

Następnym elementem bezpieczeństwa jest autoryzacja użytkownika za pomocą hasła oraz nazwy użytkownika. Dostęp do aplikacji mają tylko zapisane konta, które może tworzyć jedynie admin co jest następnym zabezpieczeniem.

Listing 5: Implementacja autoryzacji w routingach. Funkcje auth oraz authAdmin odpowiadają za autoryzację użytkownika

```

1     this.expressApp.get('/home', routes.auth, routes.hello);
2     this.expressApp.post('/users', routes.auth, routes.authAdmin,
3         routes.createUser);
4     this.expressApp.put('/users', routes.auth,
5         routes.updateUserProfile);
6     this.expressApp.delete('/users/:id', routes.auth,
7         routes.authAdmin, routes.deleteUserProfile);

```

Listing 6: Implementacja funkcji auth

```

1 module.exports.auth = async function(req, res, next) {
2     if (!req.session._id) {
3         res.status(403).send("Forbidden");
4     } else {
5         try {
6             res.locals.user = await getUserInfo(
7                 req.sequelize.admins,
8                 req.session._id
9             );
10        } catch (error) {
11            console.log("[ERROR]:", error.message);
12            res.status(505).send();
13        }
14        next();
15    }
16 };

```

Listing 7: Implementacja funkcji authAdmin

```

1 module.exports.authAdmin = async function(req, res, next) {

```

```

2     await authAccountType(req, res, next, [
3       WORKER_TABLE_ADMINISTRATOR_FIELD]);
  
```

Listing 8: Implementacja funkcji authAccountType

```

1 module.exports.authAccountType = async function(
2   req,
3   res,
4   next,
5   accountTypeArray
6 ) {
7   if (!req.session._id) {
8     res.status(403).send('Forbidden');
9   } else {
10    try {
11      const response = await getUserAccountType(
12        req.sequelize.admins,
13        req.session._id
14      );
15      let IsOneOfType = false;
16      for (let accountType of accountTypeArray) {
17        if (response[accountType]) {
18          IsOneOfType = true;
19          break;
20        }
21      }
22      if (!IsOneOfType) {
23        return res.status(403).send('Account access is
24          forbidden. ');
25      }
26      next();
27    } catch (error) {
28      console.log('[ERROR]: ', error.message);
29      res.status(505).send();
30    }
31  };
  
```

W trosce o niepowołany dostęp nieautoryzowanych użytkowników do danych ogrodu zoologicznego, wprowadzona została identyfikacja sesji.

Listing 9: Konfiguracja sesji w aplikacji.

```

1   this.expressApp.use(
2     session({
3       name: SESS_NAME,
4       resave: false,
5       cookie: {
6         maxAge: 2 * 60 * 60 * 1000,
7         sameSite: true,
8         // SECURE MUST TRUE IN PRODUCTION!!!
  
```



```

9         secure: false,
10      },
11      saveUninitialized: false,
12      secret: 'topsecret',
13  })
14  );

```

maksymalny wiek sesji został ustawiony na dwie godziny, nazwa została sprecyzowana w systemie oraz po tej nazwie będzie się ją identyfikować w programie.

Parametr `secure` odpowiada za przesyłanie sesji tylko w połączeniach bezpiecznych szyfrowanych **HTTPS**.

Parametr `resave` odpowiada za nadpisywanie przesłanej sesji w bazie sesji po stronie serwera, nawet jeśli ta nic nie modyfikowała.

Parametr `sameSite` odpowiada za przesyłanie sesji tylko w tej samej domenie.

Parametr `secret` mówi że pod tą nazwą będzie podpisana sejsa.

6. Podsumowanie i wnioski

6.1. Podsumowanie

- Udało się nam zaimplementować w pełni działającą bazę danych zgodną z założeniami,
- Serwer działa sprawnie, obsługuje wszystkie wysyłane żądania,
- Aplikacja pozwala na zarządzanie ogrodem zoologicznym, przez tworzenie, modyfikacje i usuwanie każdej z encji,
- Udało się zaimplementować różne rodzaje kont użytkowników. Utworzone zostały trzy rodzaje kont: administrator, opiekun oraz weterynarz,
- Wszystkie zamierzone funkcje zostały zaimplementowanych w aplikacji dostępowej,

6.1.1. Wnioski

W trakcie tworzenia projektu napotkaliśmy wiele problemów, jak na przykład korzystanie z bazy danych innej niż MongoDB czy też zabezpieczenia i autoryzacja użytkowników w architekturze restowej. Wszystkie problemy okazały się proste w realizacji. Do utworzenia połączenia z bazą danych PostgreSQL, wystarczyło posłużyć się biblioteką

wykorzystującą interfejs `node-postgres`. W przypadku zabezpieczeń w architekturze restowej, wystarczyło uwzględnić w implementacji serwera id sesji użytkownika, a hasła należało zaszyfrować, aby nie były przypadkiem odczytane przez osoby niepowołane. Cała aplikacja opiera się o JavaScript co pokazuje jak uniwersalnym oraz skutecznym jest narzędziem. Pomimo skorzystania z wielu różnych technologii, całość ze sobą współgrała.

Literatura

- [1] https://pl.wikipedia.org/wiki/Ogród_zoologiczny
- [2] <https://sjp.pwn.pl/slowniki/ogr%C3%B3d%20zoologiczny.html>
- [3] <https://www.postgresql.org/docs/9.3/external-interfaces.html>
- [4] <https://sequelize.org/>
- [5] <https://nodejs.org/en/>
- [6] https://www.w3schools.com/nodejs/nodejs_get_started.asp

Spis rysunków

1	Model logiczny	7
2	Model fizyczny i konceptualny	8
3	Diagram przypadków użycia	10
4	Zalogowanie przy użyciu konta <i>caretakers</i>	18
5	Testowanie uprawnień użytkownika <i>vets</i> do widoku <i>animal_health</i> oraz tabeli <i>animals</i>	19
6	Testowanie uprawnień użytkownika <i>administrators</i>	20
7	Testowanie uprawnień użytkownika <i>caretakers</i>	21
8	Wykres czasu wykonywania zapytania SELECT	23
9	Wykres czasu wykonania zapytania INSERT dla jednego rekordu	23
10	Wykres czasu wykonania zapytania DELETE dla dwóch przypadków	24
11	Wykres czasu wykonania zapytania UPDATE dla jednego zapytania	24
12	Ekran logowania	26
13	Ekran główny po zalogowaniu	27
14	Zakładka Zwierzęta	27
15	Okno modalne dodawania zwierzęcia	28
16	Ekran administratora	28
17	Okno modalne dodania konta użytkownika	29
18	Okno modalne usunięcia konta użytkownika	30
19	Ekran ustawień konta	30
20	Powodzenie logowania	31
21	Sprawdzenie czy jest się zalogowanym	32
22	Sprawdzenie pobierania zwierząt zalogowanego opiekuna	33
23	Sprawdzenie możliwości utworzenia nowej wizyty przez opiekuna	34
24	Sprawdzenie usuwania zwierzęcia przez opiekuna	35

Spis tablic