Bazy danych
Projekt

System korekcji wad postawy

Termin zajęć: środa 13:15 - 15:15
Autor: Michał Kinas 235593
Prowadzący: dr inż. Roman Ptak, W4/K9

Politechnika Wrocławska

Spis treści

1 Wstęp

1.1 Cel projektu

Stworzenie aplikacji PWA (Progressive Web Application) umożliwiającej identyfikację problemu wad postawy oraz doboru ćwiczeń korekcyjnych, z wykorzystaniem sieci neuronowych.

1.2 Zakres projektu

Stworzenie bazy danych zawierającej historię wykrytych schorzeń danego użytkownika, wraz z pytaniami służącymi do identyfikacji schorzenia oraz proponowanymi ćwiczeniami korekcyjnymi.

2 Analiza wymagań

Wybór i opracowanie wstępnych założeń dotyczących wybranych tematów projektów.

2.1 Opis działania i schemat logiczny systemu

Aplikacja umożliwi korzystanie z niej użytkownikom zalogowanym jak i niezalogowanym. Pierwszym etapem jest wybór problematycznej grupy mięśniowej lub części ciała. Dalszy etap to przejście przez wywiad składający się ze specjalnie wyselekcjonowany zestaw pytań dotyczący zadanej grupy, aby uzyskać informację niezbędne do klasyfikacji schorzenia pacjenta. Ostatni etap to dobór zestawu ćwiczeń korekcyjnych dostosowanych do potrzeb pacjenta.

2.2 Wymagania funkcjonalne

- Użytkownik niezalogowany – możliwość skorzystania z klasyfikacji problemu i doboru zestaw ćwiczeń, możliwość dodania komentarza oraz ocenienie klasyfikacji i doboru ćwiczeń
- Użytkownik zalogowany – posiada wszystkie funkcjonalności użytkownika niezalogowanego oraz: rejestracja, logowanie, dodatkowe możliwości personalizacji profilu poprzez historię schorzeń i dopasowanych ćwiczeń oraz możliwość stałej kontroli obecnego stanu schorzenia
- Administrator – zgłaszanie problemów z klasyfikacją schorzeń, możliwość rozszerzania dostępnego zestawu ćwiczeń dostosowanych do konkretnego problemu, zarządzanie kontami użytkowników (nadzór, usuwanie, rozwiązywanie problemów)

2.3 Wymagania niefunkcjonalne

2.3.1 Wkorzystywane technologie i narzędzia

- PostgreSQL
- Ruby on Rails
- Angular/React
- Sass

2.3.2 Wymagania dotyczące rozmiaru bazy danych

Przewidujemy obsługę kilku tysięcy użytkowników tygodniowo, z czego około 1/5 założy własne konto co daje około 10 000 kont stałej puli użytkowników. Poza użytkownikami konta posiadać będzie pięciu administratorów. Raz na pół roku konta nieaktywne będą usuwane.

2.3.3 Wymagania dotyczące bezpieczeństwa

Zapewnienie bezpieczeństwa przesyłanych danych dzięki zastosowaniu szyfrowanego połączenia SSL. Raz w tygodniu odbywał się będzie backup danych które zostaną zapisane na dysku twardym.

2.4 Przyjęte założenia projektowe

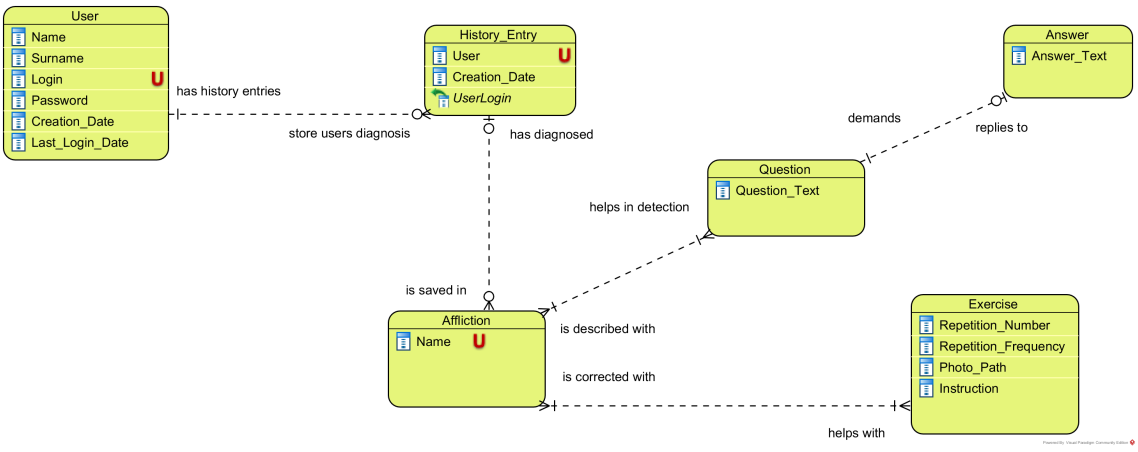
Aplikacja pozwoli użytkownikom na komfortową i bezpieczną klasyfikację wady postawy oraz dostosowanie terapii do indywidualnych potrzeb użytkownika. Logowanie pozwoli użytkownikowi na śledzenie postępów w terapii oraz na stałe monitorowanie stanu swojej postawy. Przewidujemy znaczną liczbę użytkowników danego tygodnia oraz pewną stałą pulę użytkowników. Istnieje także możliwość rozwoju aplikacji przez dodawanie nowych ćwiczeń korekcyjnych oraz obsługi nowych grup mięśniowych, wraz z wykrywaniem bardziej złożonych wad korekcyjnych w przyszłości. Funkcjonalność aplikacji pozwoli na redukcję procentowej ilości osób z wadami postawy wśród społeczeństwa oraz na zwiększenie świadomości dotyczącej tego jak ważne jest utrzymywanie poprawnej postawy ciała.

3 Projekt systemu

Projekt i struktury bazy danych, mechanizmów zapewniania poprawności przechowywanych informacji, oraz kontroli dostępu do danych.

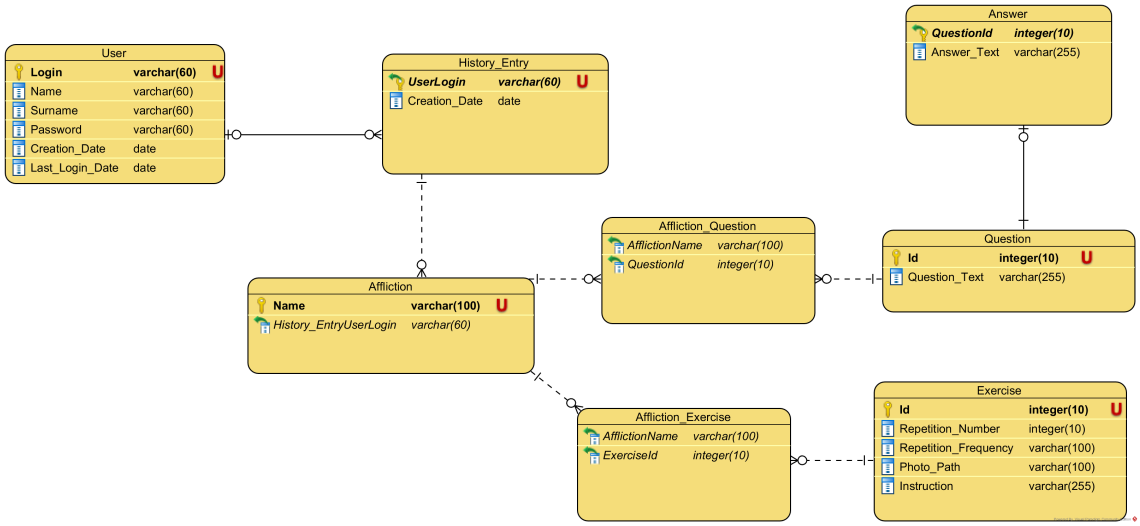
3.1 Projekt bazy danych

3.1.1 Analiza rzeczywistości i uproszczony model koncepcyjny



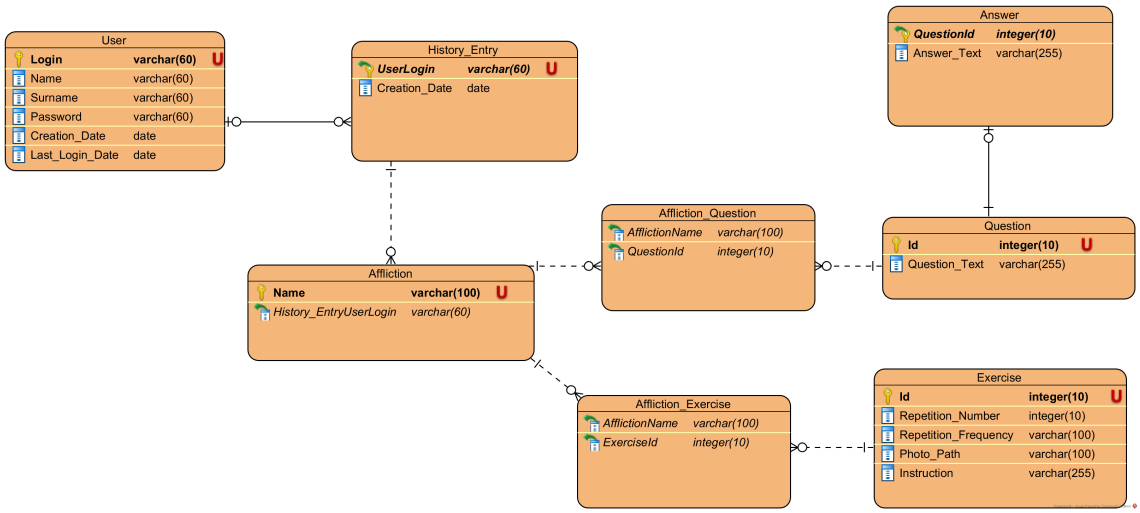
Rysunek 1: Model koncepcyjny

3.1.2 Model logiczny i normalizacja



Rysunek 2: Model logiczny

3.1.3 Model fizyczny i ograniczenia integralności danych



Rysunek 3: Model fizyczny

W przypadku usunięcia danych wybrano podejście kaskadowe tj. jeżeli usunięty zostanie użytkownik lub jego historia to razem z nią usunięte zostaną również wszystkie informacje o jego schorzeniach i stosowanych ćwiczeniach.

3.1.4 Inne elementy schematu - mechanizmy przetwarzania danych

Triggery:

- zawsze podczas tworzenia lub aktualizowania atrybutu Name i Surname wywołana w tabeli User zostanie procedura sprawdzająca czy podane zostało imię i nazwisko jeśli nie zwróci błąd i zapis nie zostanie wykonany
- zawsze przy tworzeniu lub aktualizacji atrybutu Password w tabeli User czy zawiera ono co najmniej jedną: małą i wielką oraz cyfrę jeśli nie informacja o zbyt słabym haśle zostanie zapisana w pliku logu

- raz w miesiącu na jego początku trigger wykorzysta widok zawierający użytkowników niezalogowanych od co najmniej 6 miesięcy na podstawie atrybutu Last_Login_Date a następnie usunie tych nieaktywnych użytkowników z tabeli User

Checki

- w tabeli User atrybucie Password sprawdzenie czy hasło jest dłuższe lub równe ośmiu znakom
- w tabel Exercise atrybucie Repetition_Frequency sprawdzenie czy liczba powtórzeń jest zawsze większa od zera

Widoki:

- w tabeli User widok zwracający użytkowników niezalogowani dłużej niż 6 miesięcy na podstawie atrybutu Last_Login_Date
- w tabeli Exercise widok zwracający ćwiczenia o zadanej liczbie powtórzeń na podstawie atrybutu Repetition_Number

Indeksy:

- Indeksy w tabeli User
  - indeksowanie po dacie utworzenia
- Indeksy w tabeli Exercise
  - indeksowanie po częstotliwości powtórzeń

3.1.5 Projekt mechanizmów bezpieczeństwa na poziomie bazy danych

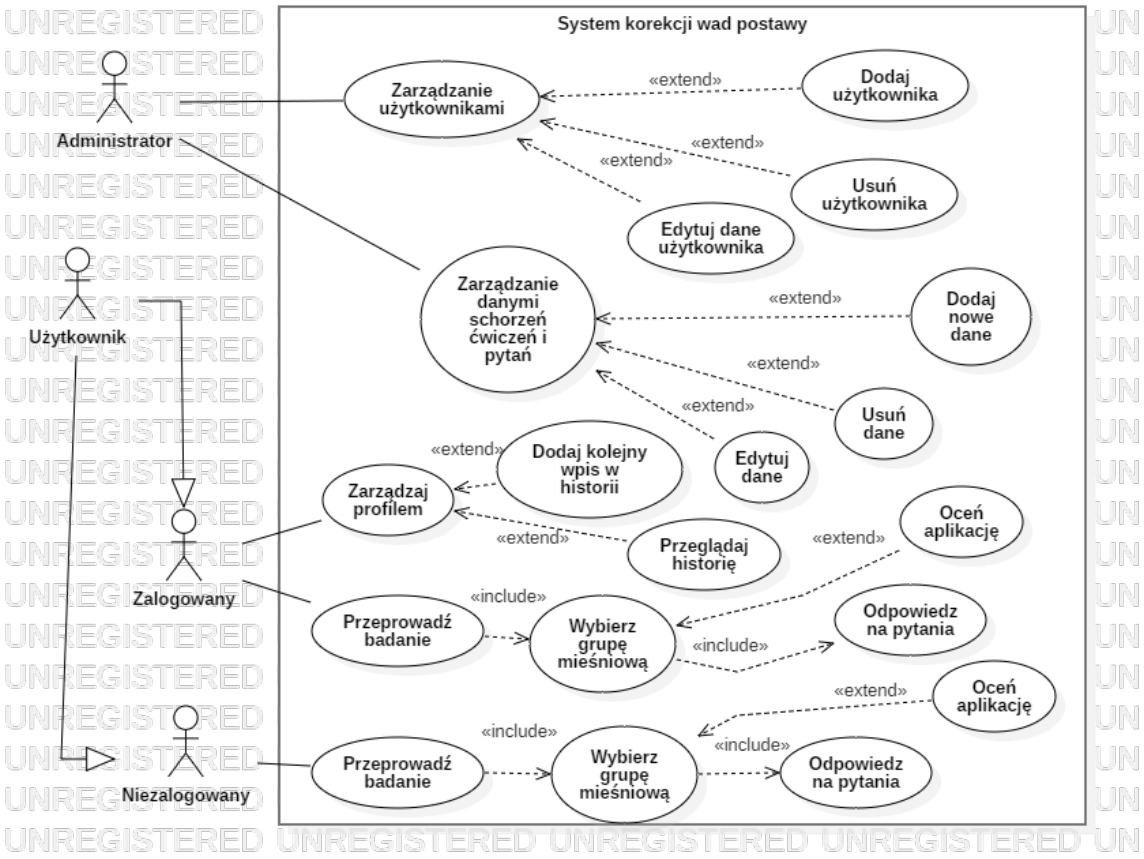
	User	History_Entry	Affliction	Exercise	Question	Answer
Administrator	odczyt danych, dodanie nowego użytkownika, usunięcie użytkownika, modyfikacja danych	odczyt danych, dodanie nowego wpisu, usunięcie wpisu, modyfikacja danych	odczyt danych, dodanie nowego schorzenia, usunięcie schorzenia, modyfikacja danych	odczyt danych, dodanie nowego ćwiczenia, usunięcie ćwiczenia, modyfikacja danych	odczyt danych, dodanie nowego pytania, usunięcie pytania, modyfikacja danych	odczyt danych, dodanie nowej odpowiedzi, usunięcie odpowiedzi, modyfikacja danych
Użytkownik zalogowany	odczyt danych, modyfikacja danych	odczyt danych, dodanie nowego wpisu, usunięcie wpisu, modyfikacja danych	odczyt danych	odczyt danych	odczyt danych	odczyt danych, dodanie odpowiedzi, modyfikacja odpowiedzi
Użytkownik niezalogowany	brak uprawnień	brak uprawnień	odczyt danych	odczyt danych	odczyt danych	odczyt danych, dodanie odpowiedzi

Rysunek 4: Dostęp do tabel w zależności od użytkownika

- login: bd-develop-adm
- hasło: APP-admin123
- konfiguracja bezpiecznego połączenia SSL z bazą danych sprawdzana przez program pgAdmin
- codziennie wykonywany backup danych przy użyciu polecenia **pg_dumpall -h databse.pl -U postgres > /home/user/databaseBackup/fullBackup.exp**

3.2 Projekt aplikacji użytkownika

3.2.1 Architektura aplikacji i diagramy projektowe



Rysunek 5: Diagram przypadków użycia

3.2.2 Interfejs graficzny i struktura menu

3.2.3 Projekt wybranych funkcji systemu

Serce aplikacji stanowić będzie głęboka sieć nueronowa mająca za zadanie zidentyfikować przypadłość na jaką cierpi użtkownik po otrzymaniu wypełnionych przez niego danych. Identyfikacja rozpocznie się od wyboru problematycznej części ciała lub grupy mięśniowej. Następnie użytkownik przekierowany zostanie do komponentu realizującego stepper, który w danym kroku wyrenderuje odpowiedni zestaw pytań, umożliwi przemieszczanie się po zestawach pytań oraz zwaliduje wprowadzone dane. Po pzejściu przez wszystkie kroki steppera dane zostaną zapisane a użytkownik otrzyma informacje o swojej diagnozie oraz możliwość zapamiętania jej w historii choroby.

3.2.4 Metoda połączenia do bazy danych - integracja z bazą danych

Integracja z bazą danych odbędzie się przez gem pg który pozwoli na dostęp do bazy danych z poziomu kodu w Ruby. Zastosowanie modelu ActiveRecords pozwoli na wygodne zarządzanie bazą danych a w razie potrzeby na modyfikację wybranych tabel przez przeprowadzenie migracji.

3.2.5 Projekt zabezpieczeń na poziomie aplikacji

Aby zapewnić bezpieczeństwo i spójność danych w aplikacji zastosowane zostaną zabezpieczenia takie jak:

- walidacje wprowadzanych danych np. liczba powtórzeń musi być większa od 0
- ograniczony czas trwania sesji użytkownika który nie wykonuje żadnych akcji
- dostęp do danych użytkownika możliwy jedynie po zalogowaniu
- sprawdzanie siły hasła tj. minimum osiem znaków, jedna wielka i mała litera oraz jedna cyfra

4 Implementacja systemu baz danych

Implementacja i testy bazy danych w wybranym systemie zarządzania bazą danych.

4.1 Tworzenie tabel, definiowanie ograniczeń, checków

Pierwszym krokiem w implementacji bazy danych było ręczne stworzenie tabel na podstawie modelu fizycznego uwzględniając odpowiednie typy danych, ograniczenia, zależności między tabelami oraz potrzebne checki.

```
CREATE TABLE "User"
(
    "Creation_Date" date NOT NULL,
    "Login" character varying(60)  NOT NULL,
    "Name" character varying(60) NOT NULL,
    "Password" character varying(60) NOT NULL,
    "Surname" character varying(60) NOT NULL,
    "Last_Login_Date" date NOT NULL,
    CONSTRAINT "User_pkey" PRIMARY KEY ("Login"),
    CONSTRAINT "User_Login_key" UNIQUE ("Login")
,
    CONSTRAINT user_password_check CHECK (length("Password"::text) > 8) NOT VALID
);

CREATE TABLE "History_Entry"
(
    "User_Login" character varying(60) NOT NULL,
    "Creation_Date" date NOT NULL,
    CONSTRAINT "History_Entry_pkey" PRIMARY KEY ("User_Login"),
    CONSTRAINT "History_Entry_User_Login_key" UNIQUE ("User_Login")
,
    CONSTRAINT "History_Entry_User_Login_fkey" FOREIGN KEY ("User_Login")
        REFERENCES "User" ("Login") MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE NO ACTION
);

CREATE TABLE "Question"
(
    "Id" integer NOT NULL,
    "Question_Text" character varying(255) NOT NULL,
    CONSTRAINT "Question_pkey" PRIMARY KEY ("Id")
);
```

Rysunek 6: Fizyczna implementacja tabel

```
CREATE TABLE "Exercise"
(
    "Id" integer NOT NULL,
    "Repetition_Number" integer NOT NULL,
    "Repetition_Frequency" character varying(100) NOT NULL,
    "Photo_Path" character varying(100),
    "Instruction" character varying(255) NOT NULL,
    CONSTRAINT "Exercise_pkey" PRIMARY KEY ("Id"),
    CONSTRAINT "exercise_repetitionFrequency_check" CHECK ("Repetition_Number" > 0) NOT VALID
);

CREATE TABLE "Answer"
(
    "Question_Id" integer NOT NULL,
    "Answer_Text" character varying(255) NOT NULL,
    CONSTRAINT "Answer_pkey" PRIMARY KEY ("Question_Id"),
    CONSTRAINT "Answer_Question_Id_fkey" FOREIGN KEY ("Question_Id")
        REFERENCES "Question" ("Id") MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE NO ACTION
);

CREATE TABLE "Affliction"
(
    "Name" character varying(100) NOT NULL,
    "History_EntryUserLogin" character varying(60) NOT NULL,
    CONSTRAINT "Affliction_pkey" PRIMARY KEY ("Name"),
    CONSTRAINT "Affliction_Name_key" UNIQUE ("Name")
,
    CONSTRAINT "Affliction_History_EntryUserLogin_fkey" FOREIGN KEY ("History_EntryUserLogin")
        REFERENCES "History_Entry" ("User_Login") MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE NO ACTION
);
```

Rysunek 7: Fizyczna implementacja tabel

```
CREATE TABLE "Affliction_Question"
(
    "AfflictionName" character varying(100) NOT NULL,
    "QuestionId" integer NOT NULL,
    CONSTRAINT "Affliction_Question_AfflictionName_fkey" FOREIGN KEY ("AfflictionName")
        REFERENCES "Affliction" ("Name") MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    CONSTRAINT "Affliction_Question_QuestionId_fkey" FOREIGN KEY ("QuestionId")
        REFERENCES "Question" ("Id") MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE NO ACTION
);

CREATE TABLE "Affliction_Exercise"
(
    "AfflictionName" character varying(100) NOT NULL,
    "Exercise_Id" integer NOT NULL,
    CONSTRAINT "Affliction_Exercise_AfflictionName_fkey" FOREIGN KEY ("AfflictionName")
        REFERENCES "Affliction" ("Name") MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    CONSTRAINT "Affliction_Exercise_Exercise_Id_fkey" FOREIGN KEY ("Exercise_Id")
        REFERENCES "Exercise" ("Id") MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE NO ACTION
);
```

Rysunek 8: Fizyczna implementacja tabel

4.2 Implementacja mechanizmów przetwarzania danych

4.2.1 Widoki

Następnie utworzone zostały widoki, zwracające użytkowników nieaktywnych od pół roku oraz ćwiczenia z liczbą powtórzeń większą niż 10.

```
CREATE VIEW "exercise_repetitionNumber_view" AS SELECT
    "Exercise"."Id",
    "Exercise"."Repetition_Number",
    "Exercise"."Repetition_Frequency",
    "Exercise"."Photo_Path",
    "Exercise"."Instruction"
   FROM "Exercise"
  WHERE "Exercise"."Repetition_Number" > 10;

CREATE VIEW "user_inactive_view" AS SELECT
    "User"."Creation_Date",
    "User"."Login",
    "User"."Name",
    "User"."Password",
    "User"."Surname",
    "User"."Last_Login_Date"
   FROM "User"
  WHERE "User"."Last_Login_Date" <= (date_trunc('month'::text, now()) - '6 mons'::interval);
```

Rysunek 9: Implementacja widoków

4.2.2 Triggery

Kolejny etap implementacji, to implementacja triggerów. Zaimplementowane zostały triggery odpowiadające za sprawdzenie siły hasła przed instrukcjami INSERT oraz UPDATE, usunięcie nieaktywnych użytkowników po instrukcjach INSERT, UPDATE oraz DELETE, sprawdzenie przed instrukcjami INSERT oraz UPDATE czy użytkownik podał imię i nazwisko.

```
CREATE FUNCTION user_check_password_strength()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
AS $BODY$BEGIN
 IF Length(NEW.Password) < 8 THEN
    RAISE EXCEPTION 'Password must be at least 8 characters long.';
 END IF;
 IF NEW.Password ~ '^([A-Z][a-z]+)+$' THEN
     RAISE EXCEPTION 'Password must contain capital letter followed by small one.';
 END IF;
 RETURN NEW;
END;$BODY$;

CREATE TRIGGER user_password_strength_trigger
    BEFORE INSERT OR UPDATE OF "Password"
    ON "User"
    FOR EACH STATEMENT
    EXECUTE PROCEDURE user_check_password_strength();

CREATE FUNCTION user_delete_inactive()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
AS $BODY$BEGIN
 DELETE FROM "user_inactive_view";
 RETURN NULL;
END;$BODY$;
```

Rysunek 10: Implementacja triggerów

```
CREATE TRIGGER user_delete_inactive_trigger
    AFTER INSERT OR DELETE OR UPDATE OF "Last_Login_Date"
    ON "User"
    FOR EACH STATEMENT
    EXECUTE PROCEDURE user_delete_inactive();


CREATE FUNCTION user_name_surname()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
AS $BODY$BEGIN
 IF NEW.Name IS NULL THEN
    RAISE EXCEPTION 'Name cannot be empty.';
 END IF;
 IF NEW.Surname IS NULL THEN
     RAISE EXCEPTION 'Surname cannot be empty.';
 END IF;
 RETURN NEW;
END;$BODY$;

CREATE TRIGGER user_name_surname_trigger
    BEFORE INSERT OR UPDATE OF "Name", "Surname"
    ON "User"
    FOR EACH STATEMENT
    EXECUTE PROCEDURE user_name_surname();
```

Rysunek 11: Implementacja triggerów

4.2.3 Indeksy oraz uprawnienia

Zaimplementowano również indeksy pozwalające na uporządkowanie tabel według najczęściej używanych kolumn. Dodano także uprawnienia dla poszczególnych grup na podstawie tabeli uprawnień.

```
CREATE INDEX users_by_creation_date
    ON "User" ("Creation_Date")

CREATE INDEX exercise_by_repetition_frequency
    ON public."Exercise" ("Repetition_Frequency")

CREATE GROUP administrators
CREATE GROUP users
CREATE GROUP users_unlogged

GRANT ALL PRIVILEGES ON "Affliction", "Affliction_Exercise", "Affliction_Question",
"Answer", "Exercise", "History_Entry", "Question", "User" TO administrators

GRANT ALL PRIVILEGES ON  "Answer", "History_Entry" TO users

GRANT SELECT ON "Affliction", "Exercise", "Question", "Affliction_Exercise",
"Affliction_Question" TO users

GRANT SELECT, UPDATE, INSERT ON "User" TO users

GRANT SELECT ON "Affliction", "Exercise", "Question", "Affliction_Exercise",
 "Affliction_Question" TO users_unlogged

GRANT SELECT, INSERT ON "Answer" TO users_unlogged
```

Rysunek 12: Implementacja indeksów oraz przyznanie uprawnień

4.3 Testowanie bazy danych na przykładowych danych

4.3.1 Tabele

Przy użyciu polecenia INSERT wstawiono przykładowe dany a następnie przy użyciu polecenia SELECT sprawdzono czy dane zostały zapisane poprawnie i pobrane.

```
db_project=# INSERT INTO "User" VALUES ('2019-05-14', 'mkinas', 'Michal', 'TEst1
2345', 'Kinas', '2019-05-02');
INSERT 0 1
db_project=# SELECT * FROM "User";
 Creation_Date | Login  | Name   | Password  | Surname | Last_Login_Date
---------------+--------+--------+-----------+---------+-----------------
 2019-05-13    | tester | Jan    | Jant12345 | Test    | 2019-01-02
 2019-05-14    | mkinas | Michal | TEst12345 | Kinas   | 2019-05-02
(2 wiersze)

db_project=# INSERT INTO "User" VALUES ('2019-03-14', 'danny', 'Danny', 'Danny77
78', 'Texter', '2019-05-10');
INSERT 0 1
db_project=# SELECT * FROM "User";
 Creation_Date | Login  | Name   | Password  | Surname | Last_Login_Date
---------------+--------+--------+-----------+---------+-----------------
 2019-05-13    | tester | Jan    | Jant12345 | Test    | 2019-01-02
 2019-05-14    | mkinas | Michal | TEst12345 | Kinas   | 2019-05-02
 2019-03-14    | danny  | Danny  | Danny7778 | Texter  | 2019-05-10
(3 wiersze)
```

Rysunek 13: Test implementacji tabel

4.3.2 Widoki

Przy użyciu polecenia SELECT ∗ FROM exercise_repetitionNumber_view sprawdzono działanie następującego widoku.

```
 Id | Repetition_Number | Repetition_Frequency | Photo_Path  |
                                                Instruction

----+-------------------+----------------------+-------------+-----------------
---------------------------------------------------------------------------------
---------------------------------
  1 |                20 | 2 series per 20      | ./dummy.jpg | lay down and put
your legs up together
  2 |                15 | 3 series per 10      | ./silly.png | conditioning exer
cise performed from a supine position by raising and lowering the upper torso wi
thout reaching a sitting position
(2 wiersze)
```

Rysunek 14: Test implementacji widoków

4.3.3 Triggery

Sprawdzenie triggerów polegało na próbie wprowadzenia nowego użytkownika z hasłem nie spełniającym podanych kryteriów oraz na stworzeniu użytkownika z ostatnią datą logowania dłuższą niż 6 miesięcy i sprawdzeniu czy został on od razu usunięty.

```
db_project=# INSERT INTO "User" VALUES ('2019-03-14', 'dan', 'Danny7778', 'Texte
r', '2019-05-10');
ERROR:  Password must be at least 8 characters long.
db_project=# INSERT INTO "User" VALUES ('2019-02-14', 'januszTest', 'Janusz', 'j
antest', 'Tester', '2019-05-10');
ERROR:  Password must be at least 8 characters long.
```

Rysunek 15: Test implementacji triggerów

```
db_project=# INSERT INTO "User" VALUES ('2019-02-14', 'johnny', 'John', 'JohnSil
lyOne12', 'Bravo', '2018-05-10');
INSERT 0 1
db_project=# SELECT * FROM "User" WHERE "Login" = 'johnny'
db_project-#
```

Rysunek 16: Test implementacji triggerów

4.3.4 Indeksy

Testy indeksów polegały na sprawdzeniu ich zawartości poprzez program psql odwołując się do właściwości danych tabel.

```
Indeksy:
    "User_pkey" PRIMARY KEY, btree ("Login")
    "User_Login_key" UNIQUE CONSTRAINT, btree ("Login")
    "users_by_creation_date" btree ("Creation_Date")
```

Rysunek 17: Test implementacji indeksów

```
Indeksy:
    "Exercise_pkey" PRIMARY KEY, btree ("Id")
    "exercise_by_repetition_frequency" btree ("Repetition_Frequency")
```

Rysunek 18: Test implementacji indeksów

4.3.5 Uprawnienia

Nadane uprawnienia przetestowano poprzez ich sprawdzenie dla poszczególnych grup.

```
db_project=# SELECT grantee, string_agg(privilege_type, ', ') AS privileges
FROM information_schema.role_table_grants
WHERE table_name='User'
GROUP BY grantee;
    grantee     |                        privileges
----------------+-----------------------------------------------------------
 postgres       | INSERT, SELECT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER
 administrators | INSERT, SELECT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER
 users          | SELECT, UPDATE
(3 wiersze)
```

Rysunek 19: Test implementacji uprawnień grup

5 Testowanie pozostałych aspektów bazy danych

5.1 Unique oraz Not null

Sprawdzono również poprawność działania ograniczeń Unique i not null poprzez próbę wprowadzenia użytkownika o takim samym indeksie oraz użytkownika z niekompletnym zestawem danych.

```
db_project=# INSERT INTO "User" VALUES ('2019-02-14', 'johnny', 'John', 'JohnSil
lyOne12', 'Bravo', '2019-05-10');
ERROR:  duplicate key value violates unique constraint "User_pkey"
SZCZEGÓŁY:  Key ("Login")=(johnny) already exists.
```

Rysunek 20: Test ograniczeń unikalności

```
db_project=# INSERT INTO "User" VALUES ('2019-02-14', NULL, 'John', 'JohnSillyOn
e12', 'Bravo', '2019-05-10');
ERROR:  null value in column "Login" violates not-null constraint
SZCZEGÓŁY:  Failing row contains (2019-02-14, null, John, JohnSillyOne12, Bravo,
 2019-05-10).
```

Rysunek 21: Test ograniczeń braku danych

6 Implementacja i testy aplikacji

6.1 Instalacja i konfigurowanie systemu

Aby poprawnie zainstalować aplikację należy w pierwszej kolejności uruchomić bazę danych zgodnie z poniższą instrukcją.

Pierwszy krok to ustawienie zmiennych w pliku konfiguracyjnym służącym do połączenia z bazą danych w taki sposób aby odnosiły się do naszej bazy.

```
1  const { Pool } = require('pg');
2  const pool = new Pool({
3    user: 'DB_USER',
4    host: 'DB_HOST',
5    database: 'DB_DB',
6    password: 'DB_PASS',
7    port: 5432,
8  });
9
10 module.exports = {
11     query: (text, params) => pool.query(text, params)
12 };
```

Rysunek 22: Konfiguracja bazy danych

Następnie kolejny krok to instalacja najnowszej wersji NodeJS używając polecenia:

sudo apt get install nodejs

Na systemach uniksowych lub pobrać najnowszą wersję z oficjalnej strony na Windowsie.

Kolejny krok to wykonanie polecenia

npm install

w katalogu głównym aplikacji.

Po poprawnym wykonaniu tego procesu należy uruchomić serwer aplikacji wykonując polecenie:

node ./server/server.js

Ostatnim krokiem potrzebnym do poprawnego uruchomienia aplikacji jest wykonanie polecenia:

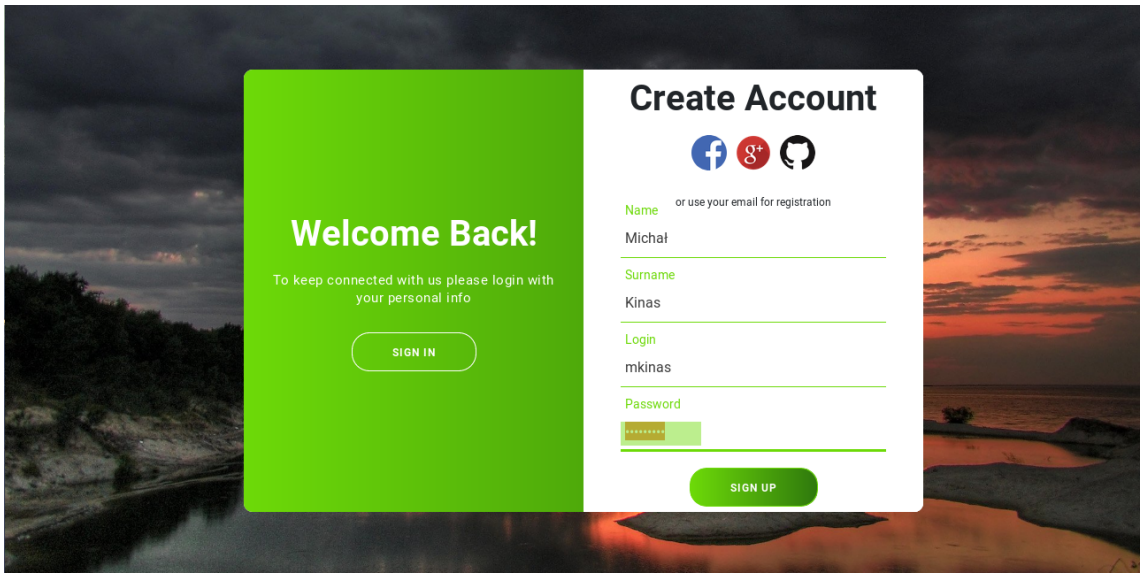
npm start

Główna aplikacja domyślnie działa na porcie 3000, serwer na porcie 4000, natomiast baza danych na porcie 5432.

6.2 Instrukcja użytkowania aplikacji

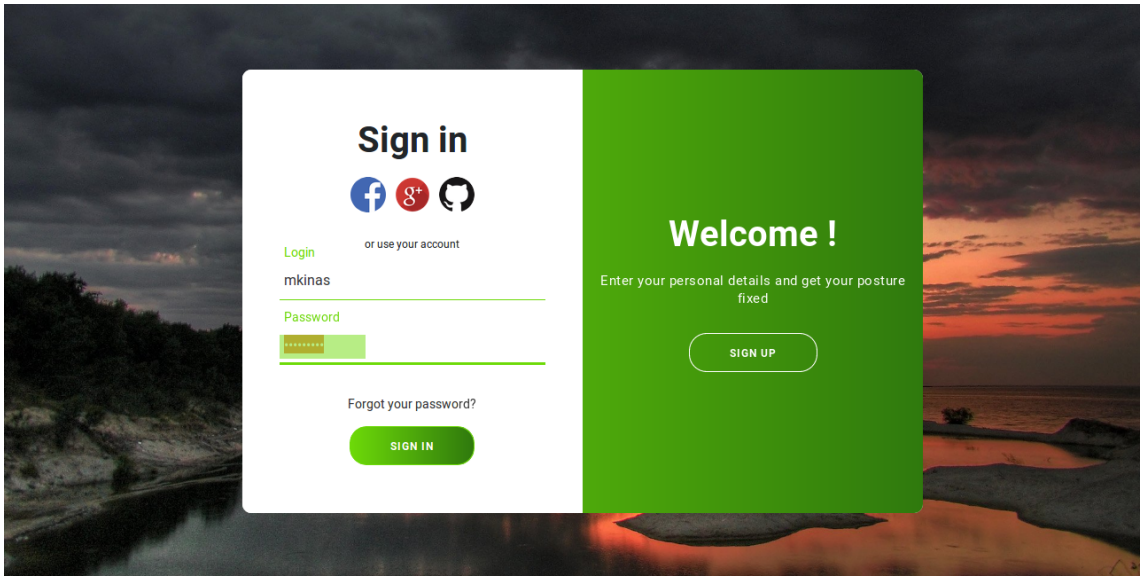
Poniżej przedstawione zostaną kroki ukazujące przykład użycia aplikacji.

- Pierwszy krok dla nowego użytkownika to rejestracja podczas której należy podać: imię, nazwisko, login oraz poprawne hasło. Użytkownik ma również możliwość rejestracji przez Githuba, Facebooka oraz G+



Rysunek 23: Rejestracja użytkownika

- Jeśli użytkownik posiada już własne konto powinien skorzystać z możliwości zalogowania podając login oraz hasło. Użytkownik ma również możliwość logowania przez Githuba, Facebooka oraz G+

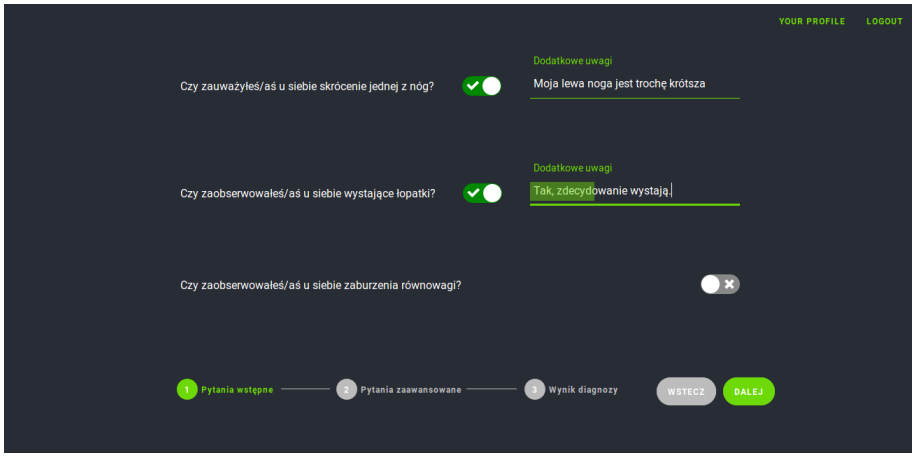


Rysunek 24: Logowanie użytkownika

- Użytkownik po zalogowaniu znajduję się w panelu głównym aplikacji skąd może wybrać problematyczną część ciała lub przejść do swojego profilu

Rysunek 25: Panel główny aplikacji

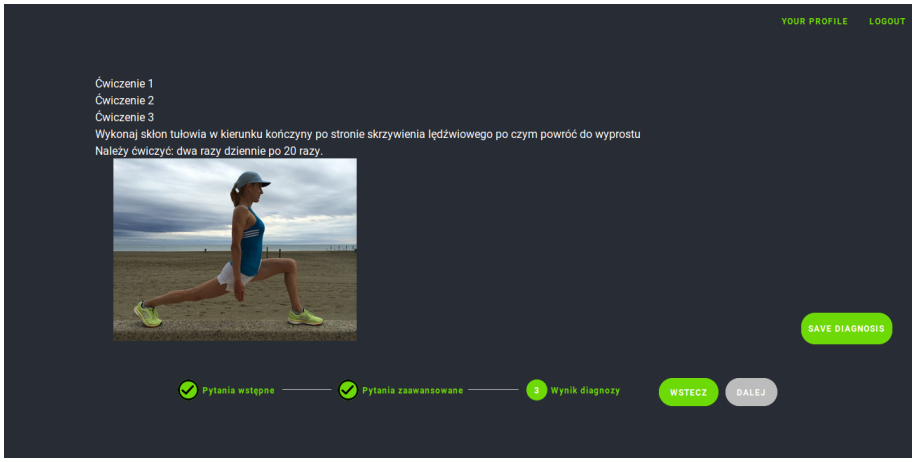
- Po wybraniu części ciała użytkownik zostaje przekierowany do komponentu odpowiadającego za zebranie odpowiedzi na kluczowe dla identyfikacji pytania.



Rysunek 26: Pierwszy zestaw pytań

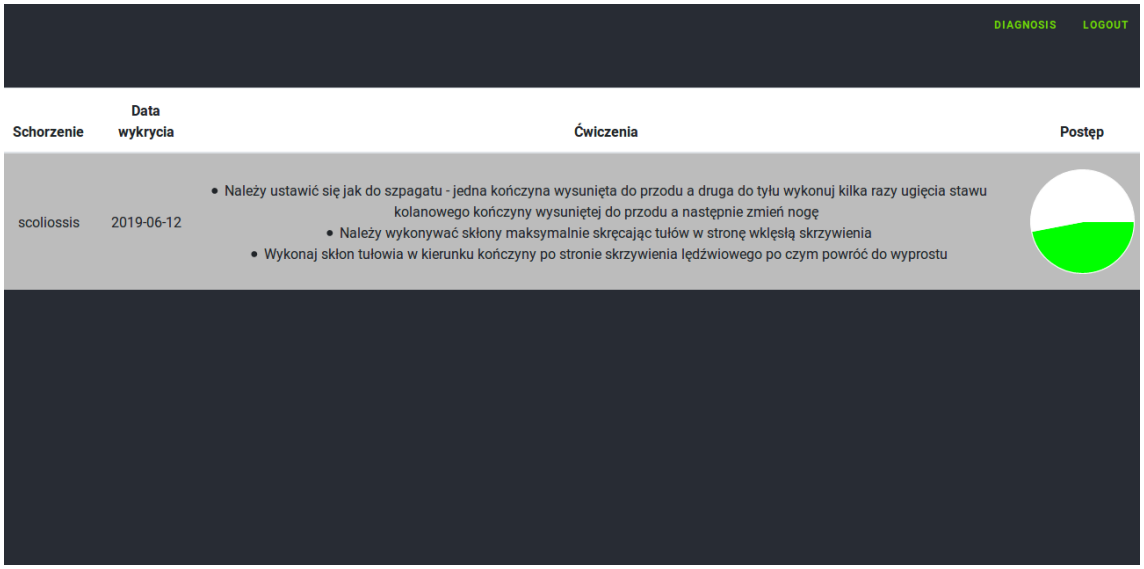
Rysunek 27: Drugi zestaw pytań

- Ostatni krok steppera to diagnoza zaprezentowana w formie proponowanych ćwiczeń w formie rozwijanego akordeonu z opisem i zdjęciem. W tym etapie użytkownik może również zapisać proponowaną diagnozę wraz z ćwiczeniami.



Rysunek 28: Pierwszy zestaw pytań

- Po zapisaniu diagnozy użytkownik może udać się do swojego profilu aby zobaczyć listę przypisanych do niego diagnoz z ćwiczeniami oraz progresem w ćwiczeniu wady postawy.

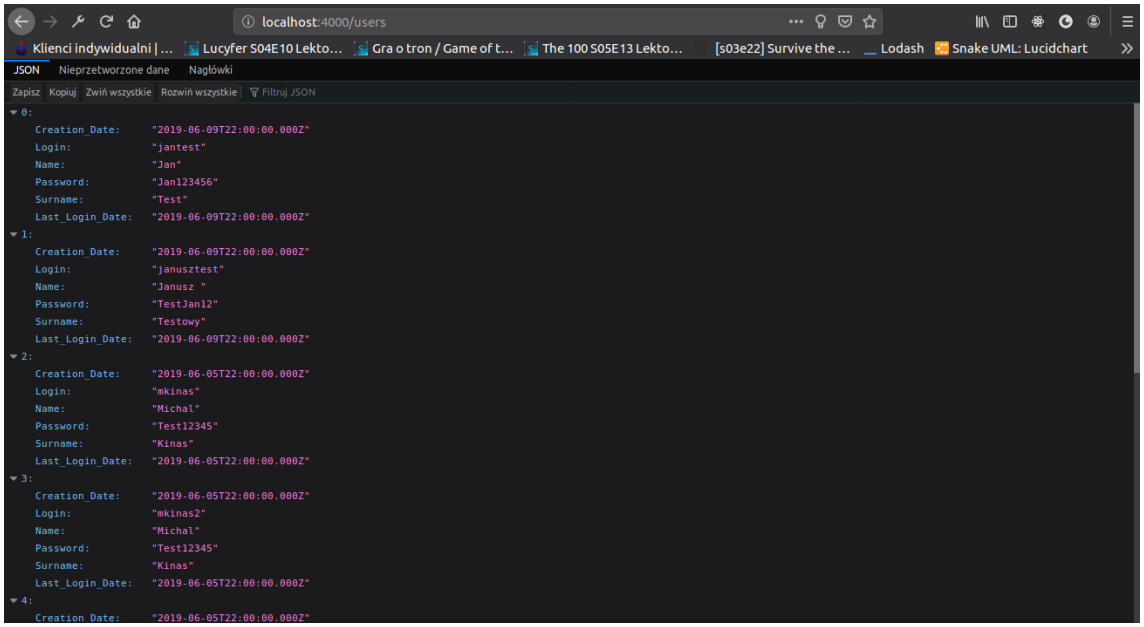


Rysunek 29: Profil użytkownika

6.3 Testowanie opracowanych funkcji systemu

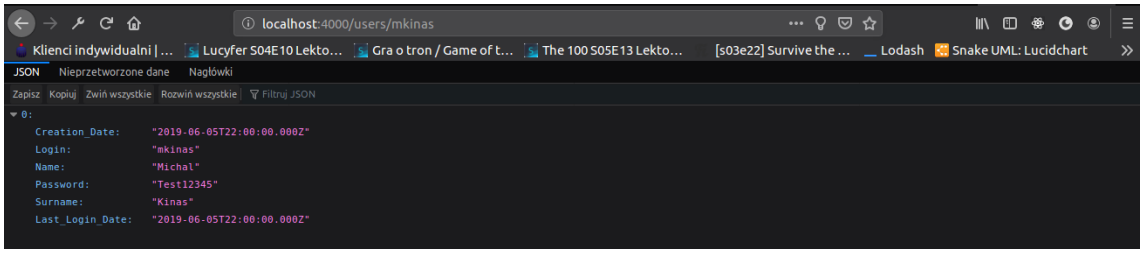
Instrukcja przedstawiona powyżej zawiera testy wszystkich funkcjonalności systemu łącznie z widokami, natomiast poniżej umieszczono testy endpointów API wykonane w przeglądarce firefox:

- GET/users



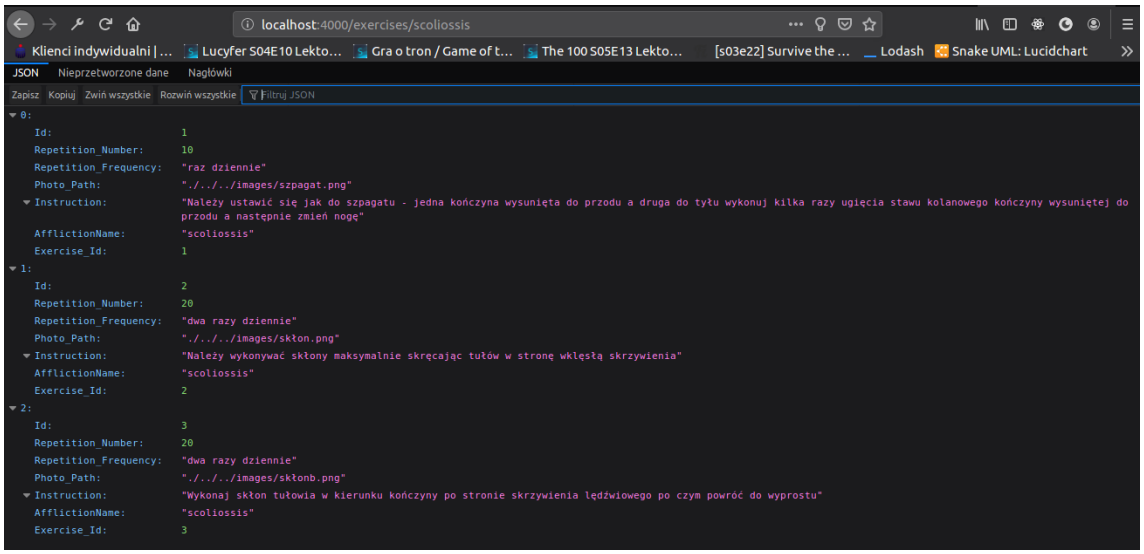
Rysunek 30: Lista użytkoników

- GET/users/:user_id



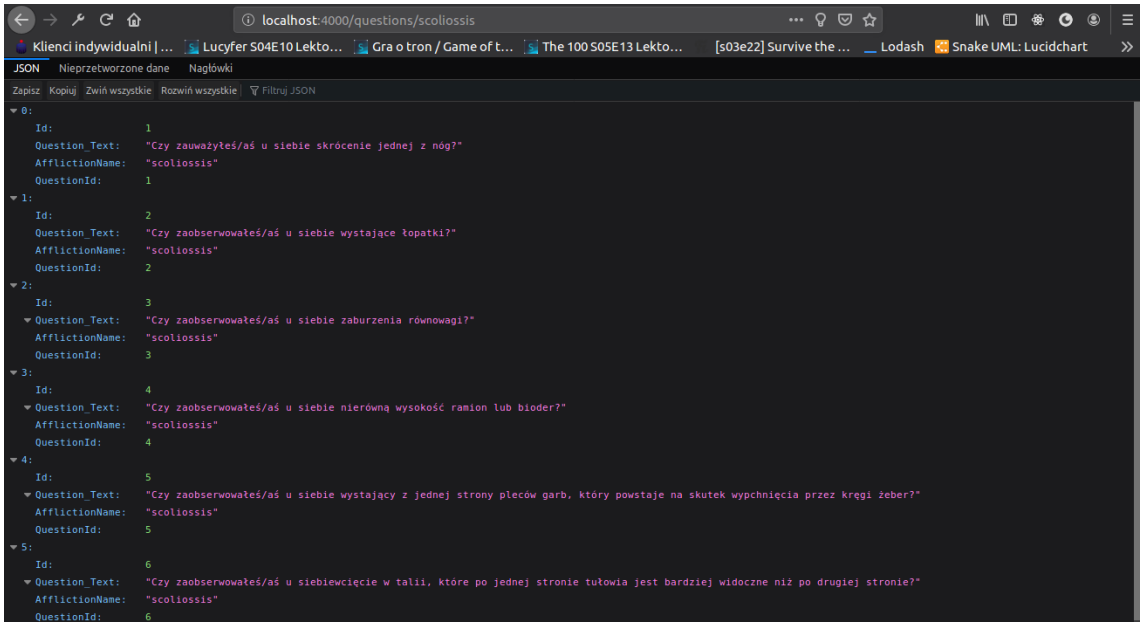
Rysunek 31: Użytkownik

- GET/exercises/:affliction_name



Rysunek 32: Ćwiczenia

- GET/questions/:question_id



Rysunek 33: Pytania

6.4 Omówienie wybranych rozwiązań programistycznych

6.4.1 Implementacja interfejsu dostępu do bazy danych

Dostęp do bazy danych został zapewniony przy użyciu biblioteki node-postgres która wykorzystuje połączenie ODBC. Poniżej przedstawiono implementację prostego serwera dostępnego przez standardowy protokół HTTP oraz przykładowego handlera API.

```
const express = require('express');
var cors = require('cors')
const bodyParser = require('body-parser');
const app = express();
const handlers = require('./handlers/users');
const handlersQuestion = require('./handlers/questions');
const handlersExercises = require('./handlers/exercises');
const handleHistory = require('./handlers/history');

const port = 4000;

app.use(bodyParser.json())
app.use(
  bodyParser.urlencoded({
    extended: true,
  })
);
app.use(cors());

app.get('/', (req, res) => {
    res.json({ message: 'Welcome to Databse Project!' });
});

app.post('/history', handleHistory.createHistoryEntry);
app.get('/history/:id', handleHistory.getHistoryEntryByUser);
app.get('/history//:id', handleHistory.getAfflictionName);
app.get('/questions/:id', handlersQuestion.getQuestionsByAffliction);
app.get('/exercises/:id', handlersExercises.getExercisesByAffliction);
app.get('/users', handlers.getUsers);
app.get('/users/:id', handlers.getUserById);
app.post('/users', handlers.createUser);
app.put('/users/:id', handlers.updateUser);
app.delete('/users/:id', handlers.deleteUser);

app.listen(port, () => {
    console.log(`App running on port ${port}.`)
```

Rysunek 34: Implementacja serwera

```
const getUserById = (request, response) => {
  const id = request.params.id
  console.log(id);
  db.query('SELECT * FROM "User" WHERE "Login" = ' + '\'' + id + '\'', (error, results) => {
    if (error) {
      throw error
    }
    response.status(200).json(results.rows)
  })
}

const createUser = (request, response) => {
  console.log(request.body);
  const { name, surname, password, login, creation_date, last_login_date } = request.body

  db.query('INSERT INTO "User" ("Login", "Creation_Date", "Name", "Password", "Surname", "Last_Login_Dat
    if (error) {
      throw error
    }
    response.status(201).send(`User added with login: ${login}`)
  })
}

const updateUser = (request, response) => {
  const id = request.params.id
  const { Name, Surname, Password, Login, Creation_Date, Last_Login_Date } = request.body
  db.query(
    'UPDATE "User" SET "Login" = $1, "Creation_Date" = $2, "Name" = $3, "Password" = $4, "Surname" = $5,
    [Login, Creation_Date, Name, Password, Surname, Last_Login_Date],
    (error, results) => {
      if (error) {
        throw error
      }
      response.status(200).send(`User modified with login: ${Login}`)
    }
```

Rysunek 35: Implementacja handlera API

6.4.2 Implementacja wybranych funkcjonalności systemu

Interfejs aplikacji został zaimplementowany przy użyciu bardzo popularnej biblioteki React pozwalającej tworzyć reaktywne i reużywalne komponenty łącząc JavaScript z HTMLem. Dane do komponentów pobierane są z bazy danych przy użyciu opisanych wcześniej endpointów API przy użyciu biblioteki do programowania asynchronicznego axios. Poniżej przedstawiono komponent Router odpowiedzialny za przemieszczanie się użytkownika po aplikacji, przykładowy kod asynchroniczny wywołujący endpointy API oraz część HTMLa odpowiedzialną za renderowanie komponentu ze stepperem.

```
import React, { PureComponent } from 'react';
import { Router, Route, Redirect, Switch } from 'react-router-dom';
import history from './history';
import MainPage from './components/MainPage/MainPage';
import Login from './components/LoginPage/Login';
import Steps from './components/Steps/Steps';
import UserProfile from './components/UserProfile/UserProfile';

class DBRouter extends PureComponent {

  constructor(props) {
    super(props);
    this.props = props;
}

  render() {
    return (
      <Router history={history}>
        <Switch>
          <Route path='/profile' component={UserProfile} />
          <Route path='/steps' component={Steps} />
          <Route path='/main' component={MainPage} />
          <Route path='/login' component={Login} />
          <Route exact path='/' component={Login} />
          <Redirect from='*' to='/' />
        </Switch>
      </Router>
    );
  }
};

export default DBRouter;
```

Rysunek 36: Implementacja Routera

```
  componentWillMount() {
    axios.get('http://localhost:4000/questions/' + this.props.history.AfflictionName)
    .then(res => {
        console.log(res.data, this.props);
        this.setState({questions: res.data});
    });
    axios.get('http://localhost:4000/exercises/' + this.props.history.AfflictionName)
    .then(res => {
        this.setState({exercises: res.data});
    });
  }
```

Rysunek 37: Przykładowe asynchroniczne wywołania handlera API

```
206   render() {
207     return (
208       <div className="main-container">
209        <ul id="nav">
210             <button className="diagnosis" type="button" onClick={this.navigateToProfile}> Your profile</button>
211             <button className="logout" type="button" onClick={this.logout}> Logout</button>
212         </ul>
213         <div className="questions-container">
214             { this.state.step1State === 'active' && this.state.questions.length !== 0 &&  <div className="column">
215                 <div className="center-space">
216                     <div className="question-text">{this.state.questions[0].Question_Text}</div>
217                     <Switch onChange={this.handleChange1} checked={this.state.question1} id="normal-switch"/>
218                     { this.state.question1 &&
219                         <div className="group">
220                             <input type="text" value={this.state.answer1} onChange={this.updateAnswer1}/>
221                             <span className="highlight" />
222                             <span className="bar" />
223                             <span className="label">Dodatkowe uwagi</span>
224                         </div>}
225                 </div>
226                 <div className="center-space">
227                     <div className="question-text">{this.state.questions[1].Question_Text}</div>
228                     <Switch onChange={this.handleChange2} checked={this.state.question2} id="normal-switch"/>
229                     { this.state.question2 &&
230                         <div className="group">
231                             <input type="text" value={this.state.answer2} onChange={this.updateAnswer2}/>
232                             <span className="highlight" />
233                             <span className="bar" />
234                             <span className="label">Dodatkowe uwagi</span>
235                         </div>}
236                 </div>
237                 <div className="center-space">
238                     <div className="question-text">{this.state.questions[2].Question_Text}</div>
239                     <Switch onChange={this.handleChange3} checked={this.state.question3} id="normal-switch"/>
240                     { this.state.question3 &&
```

Rysunek 38: Część kodu HTML w komponencie renderującym Stepper

6.4.3 Implementacja mechanizmów bezpieczeństwa

Aplikacja, poza procedurami w bazie danych zawiera następujące mechanizmy bezpieczeństwa:

Zabezpieczenie przed atakami sql-injection przy użyciu modułu sql-injection który jest odpowiedzialny za detekcję oraz przerywanie takich ataków poprzez wysyłanie odpowiedzi 403.

```
10
11  const sqlinjection = require('sql-injection');
12
13  app.configure(() => {
14     app.use(sqlinjection);
15  });
16
```

Rysunek 39: Zabezpieczenie przez sql-injection

Zabezpiecznenie przed atakami DOS poprzez ograniczenie jednoczesnej liczby requestów.

```
1  const rateLimit = require("express-rate-limit");
2
3  const limiter = rateLimit({
4    windowMs: 15 * 60 * 1000, // 15 minutes
5    max: 100 // limit each IP to 100 requests per windowMs
6  });
7
8  app.use(limiter);
```

Rysunek 40: Zabezpieczenie przed atakami DOS

7 Podsumwanie i wnioski

Odnosząc się do całości wykonanych prac projekt można uznać za zakończony sukcesem. Mając na uwadze wstępne wymagania biznesowe stworzono kolejno projekty: konceptualny, logiczny i na końcu fizyczny bazy danych. Po przeprowadzeniu normalizacji zaimplementowano: indeksy, widoki,

wyzwalacze, ograniczenia dostępu oraz wszystkie inne potrzebne funkcjonalno]'sci. Po implementacji przeprowadzono również dokładne testy wszystkich funkcjonalości. Finalnie powstała aplikacja składająca się z bazy danych, części serwerowej oraz wizualnej części aplikacji. Aplikacja posiada zabezpieczenia oraz funkcjonalności pozwalające jej na powstanie wersji produkcyjnej oraz korzystanie z niej przez użytkowników.

7.1 Literatura

- Oficjlana dokumentacja PostgresSQL https://www.postgresql.org/docs/11/index.html
- Oficjalna dokumentacja NodeJs https://nodejs.org/en/docs/
- Oficjalna dokumentacja ReactJS https://reactjs.org/docs/getting-started.html
- Wykłady z baz danych dr. inż Romana Ptaka http://roman.ptak.staff.iiar.pwr.wroc.pl

Spis rysunków