**Politechnika Wrocławska**

**Wydział Informatyki i Telekomunikacji**

Kierunek: **ITE**

Specjalność: **INS**

PRACA DYPLOMOWA

MAGISTERSKA

**Wielokryterialna analiza frameworków frontendowych z użyciem wnioskowania rozmytego**

Dominik Tłokiński

Opiekun pracy

**Prof. dr hab. inż. Jan Magott**

Słowa kluczowe: aplikacja internetowa, react, springboot, siłownia, ćwiczenie

WROCŁAW 2024

# Streszczenie

W ciągu ostatnich kilku lat, z powodu epidemii Covid-19, duża część społeczeństwa znacznie ograniczyła swoją aktywność fizyczną. Jest to spowodowane między innymi pracą zdalną oraz nawykami niewychodzenia z domu. Od niedawna część z nich, w obawie o swoje zdrowie oraz z powodu chęci poprawienia swojej kondycji, zaczęła uprawiać sport w różnych formach, takich jak spacery, jazda na rowerze czy chodzenie na siłownię. Ta ostatnia grupa spotyka się z problemem dobierania ćwiczeń do treningu, a co za tym idzie, przeszukuje różne strony internetowe o tematyce siłowni, w celu znalezienia odpowiedniego dla siebie planu treningowego.

W pracy przedstawiono planowanie, projektowanie oraz implementację aplikacji internetowej zbudowanej przy pomocy narzędzi *ReactJS* oraz *Springboot*. Aplikacja ma pozwolić użytkownikowi na wygenerowanie planu treningowego w zależności od sylwetki, wieku oraz stopnia zaawansowania. Będzie to możliwe dzięki dużej liczbie ćwiczeń znajdujących się w bazie danych *PostgreSQL*, która zostanie skonfigurowana za pomocą narzędzia konteneryzacji – *Dockera*. Ćwiczenia uzupełniać będzie *Flyway*, czyli narzędzie migracji bazy danych.

Omówiono również narzędzia użyte do stworzenia aplikacji. Najważniejsze części kodu zostały opisane i przedstawione na listingach. Na koniec zaprezentowano testy najważniejszych funkcjonalności.

# Abstract

In the last few years, due to Covid-19 pandemic, a large part of society has significantly reduced its physical activity. Home office and the habit of staying at home are two of the reasons why it happens. Recently a part of that society in fear of health and willingness to improve physical condition, has engaged in sports like going for a walk, biking, or going to the gym. The last activity mentioned may cause problems when selecting exercises to create a workout plan. To solve this problem many people are searching the Web to find a site that will help them find the best plan.

In this paper planning, designing and implementation of web application are presented. The application will be built using *ReactJS* and *Springboot* and its main goal is to generate a training plan based on figure, age, and advancement in training. It will be possible thanks to enormous number of exercises contained in the *PostgreSQL* database, which is going to be configured using *Docker* containerization. Exercises will be filled using *Flyway* – a database migration tool.

Tools used to create application have also been discussed in this paper. The most significant portions of the code will be described and presented in listings. Ultimately, tests of key functionalities will be introduced.

**Spis treści**

[Wstęp 4](#_Toc121415909)

[1 Wprowadzenie 5](#_Toc121415911)

[2 Faza projektowa 8](#_Toc121415913)

[3 Interfejs użytkownika 12](#_Toc121415920)

[4 Wykorzystane technologie 18](#_Toc121415930)

[5 Implementacja aplikacji 24](#_Toc121415947)

[6 Testowanie aplikacji 38](#_Toc121415959)

[7 Podsumowanie 41](#_Toc121415962)

[Bibliografia 43](#_Toc121415965)

[Spis rysunków 44](#_Toc121415966)

[Spis listingów 45](#_Toc121415967)

[Dodatki 46](#_Toc121415968)

# Wstęp

Przy okazji bycia aktywnym członkiem społeczności klubu sportowego zaobserwowano trudności w doborze ćwiczeń do planu treningowego przez ludzi, którzy zaczynają uczęszczać na siłownię lub chcą zróżnicować wykonywane ćwiczenia. Niejednokrotnie brakowało motywacji do zgłębienia swojej wiedzy bądź poczucia pewności siebie podczas dokonywania wyboru. W związku z tym rozpoczęto poszukiwania narzędzia, które ułatwiłoby proces doboru ćwiczeń i grupowania ich w odpowiednie treningi. Dotychczas nie powstało rozwiązanie, które umożliwiłoby automatyczny, bądź manualny proces tworzenia planu treningowego zgodnie z kryteriami określonymi przez użytkownika. W niniejszej pracy przedstawiono proces tworzenia i implementacji aplikacji internetowej z łatwym w obsłudze interfejsem umożliwiającym generowanie planu treningowego.

## Cel i zakres pracy

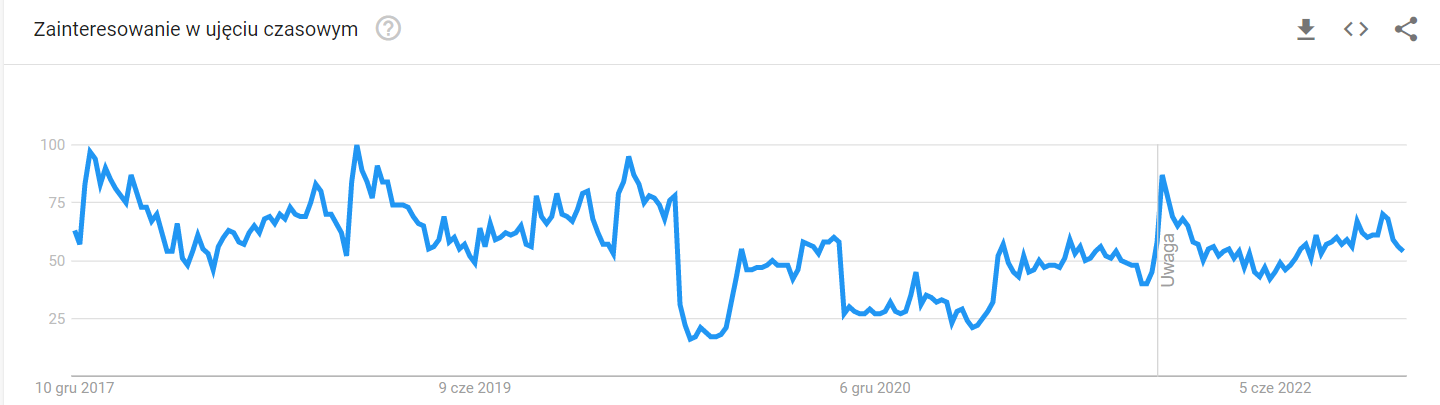
Celem pracy jest stworzenie aplikacji internetowej z graficznym interfejsem użytkownika, pozwalająca na generowanie planów treningowych. Po zalogowaniu wymagane będzie podanie danych takich jak: wiek, płeć, waga, sprawność fizyczna. Dzięki tym informacjom aplikacja stworzy plan treningowy na każdą z partii mięśniowych z określoną ilością powtórzeń i serii dla każdego z ćwiczeń. Aplikacja nie będzie wybierać aktywności, które mogą być niebezpieczne dla konkretnej osoby, np. trudne plany treningowe nie będą generowane dla osób początkujących. Praca obejmuje zdefiniowanie wymagań funkcjonalnych i niefunkcjonalnych.

Zakres pracy składa się ze zbudowania aplikacji internetowej za pomocą języków Java i *Javascript*, biblioteki *ReactJS* oraz frameworku *Springboot*. W celu komunikacji *backendu* i *frontendu* wystawione zostanie RESTowe API po stronie serwera. Tematyka pracy obejmie również stworzenie bazy danych *PostgreSQL*, która zostanie skonfigurowana za pomocą narzędzia *Docker*.

# Wprowadzenie

Wyniki badań przedstawione w raporcie *Polski rynek sportu* Polskiego Instytutu Ekonomicznego z 2019 r. pokazały, że zaledwie 28% Polaków uprawia regularnie sport. Spowodowane pandemią Covid-19, przejście znacznej części społeczeństwa na pracę zdalną wpłynęło na dodatkowe ograniczenie aktywności fizycznej. Przyjmuje się, że odsetek ludzi uprawiających sport lub inną aktywność fizyczną przynajmniej raz w tygodniu zmalał aż o 16%. Wykazano też, że spacery są najczęściej uprawianą formą aktywności fizycznej. Z przeprowadzonych badań społecznych zauważono, że prawie 50% uczestników badania ćwiczy w celu poprawy samopoczucia lub zdrowia. Niecałe 25% ćwiczyło, by uniknąć problemów ze zdrowiem. Udowodnione zostało, że sytuacja epidemiczna, skutkująca przejściem na pracę i naukę zdalną spowodowała spadek aktywności fizycznej wśród uczestników badania. Aż 13% badanych nie uprawiało żadnego sportu, a tylko 9% ćwiczyło w przerwach pomiędzy pracą. [16]

Ludzie w momencie wybuchu pandemii na krótki czas przestali interesować się siłownią (Rysunek 1). Od tamtego momentu znaczny wzrost zainteresowania siłownią jest zauważalny jedynie podczas okresu świąt Bożego Narodzenia, gdzie mniejszy stopień zainteresowania w grudniu, obfituje wysokim jego poziomem w styczniu. Oprócz tych miesięcy, siłownią interesuje się średnio taka sama liczba osób.



Rysunek 1: Popularność siłowni w Polsce na przestrzeni lat 2017-2022

Proces budowy strony internetowej składa się z sześciu etapów. Pierwszym z nich jest badanie potrzeb i analiza, której zadaniem jest ocenienie zapotrzebowania danego produktu oraz wymagań postawionych przez grupę docelową. Jednym z zadań analizy jest dokładne przeanalizowanie produktów oferowanych przez konkurencyjne firmy. Krokiem drugim jest planowanie projektu. Plan powinien zawierać przeanalizowane ważne kwestie, takie jak funkcjonalności produktu, kosztorys oraz nieprzekraczalny termin wykonania. Po planowaniu następuje projekt informatyczny. Gdy cele projektu zostały już określone następuje zebranie ich w całość w odpowiednim dokumencie. Ten krok jest ważny, ponieważ stworzenie czytelnej i zrozumiałej dokumentacji na początku procesu budowy ułatwia programistom implementacje rozwiązania z zachowaniem najwyższych standardów w dalszej części prac nad stroną internetową. Po ukończeniu dokumentacji należy wybrać właściwą firmę lub osoby, które podejmą się wykonania produktu. Wybór powinien zostać podjęty po dokładnym przeanalizowaniu portfolio partnera oraz opinii na jego temat, co wymaga odpowiednich kwalifikacji z zakresu zarządzania zasobami ludzkimi. Po podjęciu decyzji dotyczącej zespołu, który zaimplementuje aplikację, należy wybrać technologie, które pozwolą na wytworzenie wysokiej jakości produktu, który będzie spełniać najnowsze standardy implementacyjne i bezpieczeństwa. Ostatnim krokiem jest zaangażowanie ludzi tworzących produkt. Zgrany i prawidłowo komunikujący się zespół szybko rozwiąże każdy napotkany problem, a oddane rozwiązanie będzie działać prawidłowo, szybko i efektownie. [17]

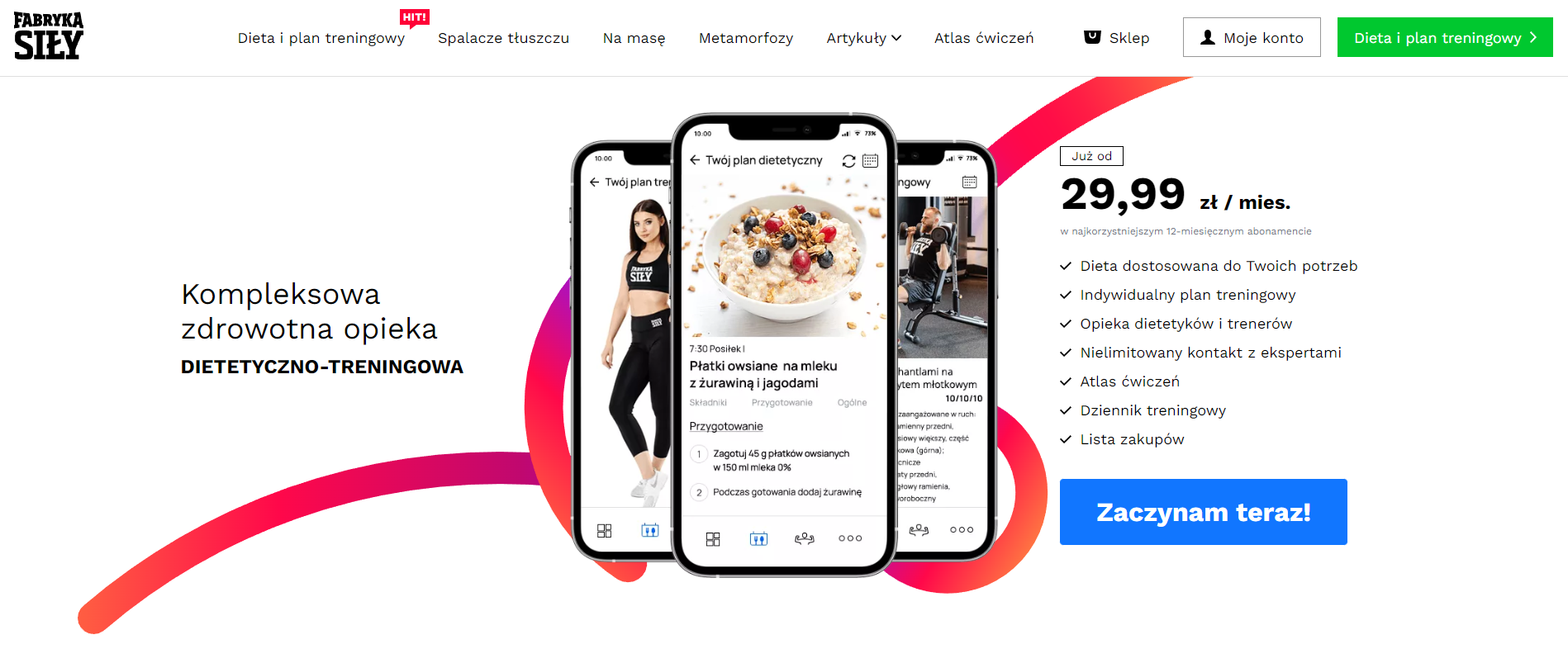
Zgodnie z jednym z punktów procesu budowy strony internetowej najważniejszym w budowaniu strony internetowej jest wybór odpowiednich technologii, który zależy od kilku czynników.

Pierwszym z nich jest użyteczność narzędzia w zależności od wyznaczonych celów i problemów, które trzeba rozwiązać. Każda technologia pozwala na zrealizowanie innych zagadnień. Nie powinno się wybierać języka programowania, który na samym początku postawi przed deweloperem trudności związane z problemami implementacji danego rozwiązania. Kolejnym czynnikiem jest znajomość technologii. Większość projektów informatycznych można zbudować za pomocą różnorodnych narzędzi. Języki programowania, frameworki i biblioteki różnią się od siebie. Jedna technologia może być wspierana lepiej niż inna. Znajomość takich różnic pozwoli na odpowiedni wybór. Popularność to ważny element, którego nie wolno pominąć przy wybieraniu technologii. Popularne technologie gromadzą wokół siebie dużą ilość ludzi z pasją do programowania przy użyciu danego narzędzia. Jest to ważne z jednego, trywialnego powodu. Ważkość tego zagadnienia można przedstawić za pomocą przykładu – pisanie skryptów w języku *Bash* oraz języku *Python*. Pisząc kod w *Pythonie* znacznie łatwiej jest znaleźć rozwiązanie danego problemu w sieci. Natomiast *Bash* budzi mniejsze zainteresowanie społeczności, co powoduje, że w przypadku napotkania błędu jest większa szansa, że nie uda się szybko odnaleźć odpowiedzi. Istotnym elementem, na który trzeba zwrócić uwagę jest też dojrzałość i stabilność technologii rozumianą przez jakość narzędzia, ilość dostępnych frameworków i bibliotek. Gdy rozwiązanie jest stabilne, tworzy się wokół niego społeczność, która dba o jego rozwój. Nie ma wtedy obaw o to, że technologia przestanie być wspierana w ciągu kilku następnych lat. Skalowalność produktu jest cechą poszukiwaną przez firmy, które zamierzają go w przyszłości rozwijać. Przy wyborze technologii należy upewnić się, czy pozwala ona na proste i szybkie skalowanie rozwiązania. Budując produkt należy myśleć o przyszłości, ponieważ problem ze skalowalnością będzie generował duże koszty związane z refaktoryzacją kodu. Zastosowana technologia w dużym stopniu wpływa na koszt budowy projektu. Jest to spowodowane między innymi stawkami godzinowymi specjalistów danej dziedziny. Wysokie stawki powodują wzrost kosztów implementowania rozwiązania. Czas implementowania danego rozwiązania może różnić się w zależności od wybranej technologii. Porównując *Javę* i *Pythona* można zauważyć, że ten sam kod zostanie zaimplementowany szybciej w języku *Python*. Jest to spowodowane jego charakterystyczną zwięzłą i czytelną składnią, która sprawia, że programowanie postępuje znacznie szybciej. Poza samym programowaniem zmienia się też czas potrzebny na wykonywanie czynności naprawiających błędy programu. Wysokie wynagrodzenie specjalisty może być zrekompensowane używając technologii, w której rozwiązania implementuje się szybciej. [18]

## Przegląd aktualnych rozwiązań

W sieci nie istnieje strona internetowa, która umożliwiałaby generowanie planu treningowego. Istnieją aplikacje o tematyce ćwiczeń i siłowni, które oferują jego ułożenie. Niestety otrzymanie takiego planu będzie wymagało dużych nakładów pieniężnych.

Przykładem takiej strony jest *fabrykasily.pl* (Rysunek 2), która za opłatą oferuje układanie indywidualnego planu treningowego wraz z dietą i wsparciem trenera personalnego.



Rysunek 2. Strona główna fabrykasily.pl

Jednak przy kliku tysiącach użytkowników trudno jest, by faktycznie stosować indywidualne podejście do klienta. Nie każda osoba aktywna fizycznie może pozwolić sobie na dietę oraz opłatę za plany treningowe co miesiąc.

Stroną oferującą pomoc w doborze ćwiczeń jest *budujmase.pl*. Użytkownik może za opłatą otrzymać jedną z 3 ofert. Dostępne do wyboru są plan treningowy i dietetyczny, tylko plan treningowy lub tylko plan dietetyczny. Z powtarzających się wad ze strony *fabrykasily.pl* należy podkreślić dwukrotnie większą sumę do zapłaty za wybraną ofertę (Rysunek 3).



Rysunek 3: Koszty pakietów treningowych na stronie budujmase.pl

# Faza projektowa

Pierwszym etapem budowania systemu informatycznego jest faza projektowa, w ramach której należy określić założenia i cele systemu. Istotnym jest, aby przeprowadzić ten etap przed rozpoczęciem programowania i implementacji rozwiązań. W tym celu wyszczególniono wymagania funkcjonalne oraz niefunkcjonalne, a także przygotowano diagram związków encji oraz diagram przypadków użycia wraz z opisem poszczególnych scenariuszy.

## Słownik pojęć

* **Użytkownik** – osoba korzystająca ze strony internetowej, używająca aplikacji w celu stworzenia planu treningowego,
* **Ćwiczenie** – rodzaj aktywności fizycznej, która jest zaplanowana, określona i powtarzalna. Ma na celu poprawę sprawności fizycznej,
* **Ćwiczenie z danymi** – obiekt ćwiczenia rozszerzony o atrybuty *powtórzenia* oraz *serie.* Jest przypisany do jednego dnia treningowego,
* **Dzień treningowy** – obiekt zawierający listę ćwiczeń z danymi, posiada atrybuty tj. *nazwa* i *data*. Jest przypisany do jednego planu treningowego,
* **Plan treningowy** – obiekt zawierający listę dni treningowych. Wygenerowany plan treningowy nie może zawierać więcej niż siedem dni treningowych. Jest przypisany do jednego użytkownika.

## Wymagania funkcjonalne

Wymaganiami funkcjonalnymi aplikacji jest przede wszystkim udostępnienie użytkownikowi możliwości generowania planów treningowych, gdy ten uzupełni formularz odpowiednimi danymi. Aplikacja powinna wyświetlać listę planów treningowych użytkownika, listę dni treningowych zawartych w planie treningowym oraz listę ćwiczeń, które przypisane są do dnia treningowego. Powinna istnieć podstrona, gdzie wyświetlona zostanie paginowana lista ćwiczeń, którą będzie można filtrować po nazwie, trudności ćwiczenia, partii mięśniowej oraz typie ćwiczenia.

Aplikacja powinna umożliwiać przeglądanie: wszystkich ćwiczeń znajdujących się w bazie danych, planów treningowych, dni treningowych zawartych w planie treningowym oraz ćwiczeń zawartych w pojedynczym dniu treningowym.

Użytkownik powinien mieć możliwość stworzenia oraz zalogowania się na konto. Jego przywilejem ma być możliwość stworzenia własnego planu treningowego, samodzielnie wybierając ćwiczenia. W planie treningowym powinien móc zarówno edytować każdy z dni treningowych, jak i usunąć swój: plan treningowy, dzień treningowy, czy ćwiczenie z danego dnia treningowego. Aplikacja udostępnia podstronę ze szczegółami danego ćwiczenia, które użytkownik może sprawdzić.

## Wymagania niefunkcjonalne

W aplikacji jeden plan treningowy powinien obejmować jeden tydzień i zawierać maksymalnie siedem dni treningowych. Baza danych musi zawierać ćwiczenia, które wyświetlane w tabeli będą zawierać atrybuty, takie jak trudność i partia mięśniowa. Szczegółami ćwiczenia powinny być jego opis, trudność, partia mięśniowa oraz film z poradnikiem. Plany treningowe nie mogą być wygenerowane, jeśli użytkownik nie podał wcześniej żadnych danych.

## Diagram ER

Diagram związków encji (Rysunek 4) składa się z dziewięciu tabel. Pięć z nich (oznaczonych kolorem żółtym) przedstawia *encje*, a pozostałe cztery (oznaczone kolorem niebieskim) oznaczają *enumy*.

Przegląd diagramu można rozpocząć od tabeli *Exercise*, która obok tabeli *Gym\_user* jest najważniejszą częścią budującą bazę danych. Zawiera ona niezbędne informacje do opisania danego ćwiczenia oraz dane, które są pomocne przy generowaniu planu treningowego. Tymi danymi są m.in. trudność, partia mięśniowa oraz cztery ostatnie wiersze, gdzie przechowywana jest wiedza na temat tego, czy dane ćwiczenie jest odpowiednie dla osób w podeszłym wieku, otyłych, kobiet, dzieci. Tabela *Exercise* połączona jest relacją *jeden-do-wielu* z tabelą *Exercise\_with\_data.*

*Exercise\_with\_data* powstała z dwóch powodów. Pierwszy z nich to fakt, że poprawnie skonstruowany plan treningowy zawiera ćwiczenia z podaną informacją, ile razy dane ćwiczenie trzeba wykonać. Kolejną przyczyną jest ominięcie relacji *wiele-do-wielu* między tabelami *Exercise* oraz *Training\_day. Exercise\_with\_data* łączy te dwie encje, jednocześnie przechowując informacje o ilości serii i powtórzeń, które osoba ćwicząca na siłowni musi wykonać.

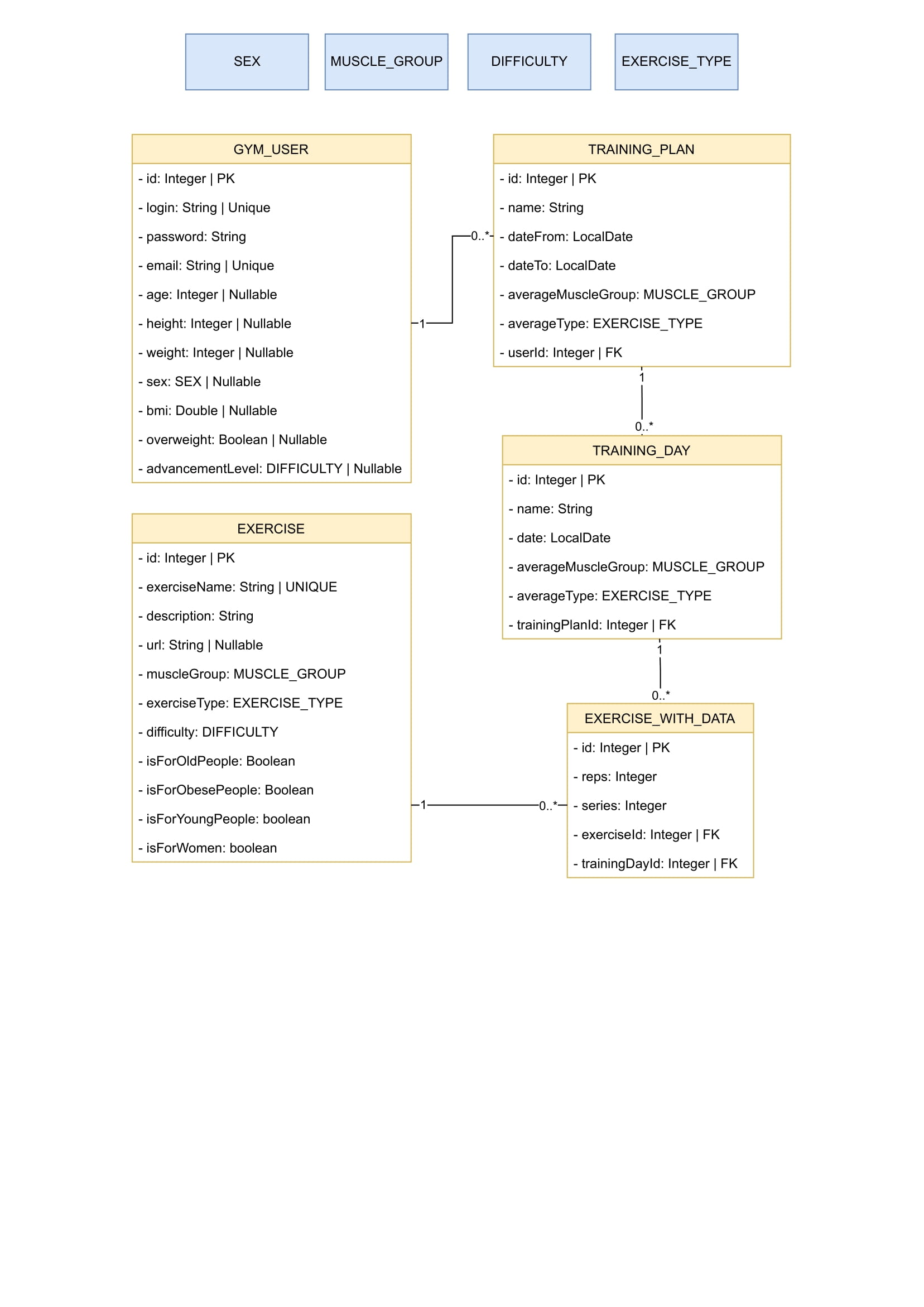
*Training\_day* to tabela przechowująca w sobie listę ćwiczeń z powtórzeniami i seriami. Zawiera w sobie również dane o tym, kiedy użytkownik powinien wykonać dane ćwiczenia, jaka jest najczęściej trenowana partia mięśniowa oraz jaki typ aktywności oferuje trening danego dnia. Lista ćwiczeń encji *Training\_day* nie ma ograniczenia co do jej rozmiaru.

W encji *Training\_plan* znajdują się atrybuty określające nazwę, datę pierwszego i ostatniego dnia treningowego, najczęściej trenowaną partię mięśniową i najczęściej występujący typ ćwiczenia. Jest połączona relacją *wiele-do-jeden* z encją *Gym\_user* oraz *jeden-do-wielu* z encją *Training\_day.* W celu spełnienia jednego z wymogów niefunkcjonalnych, po stronie interfejsu graficznego klienta nałożona jest blokada uniemożliwiająca wysłanie żądania o wygenerowanie planu z ilością dni treningowych w planie większą niż siedem.

*Gym\_user* to encja przedstawiająca użytkownika w systemie. Osoba korzystająca z aplikacji w celu zarejestrowania musi podać trzy wymagane atrybuty *login*, *email* i *hasło*. Pozostałe informacje uzupełniane są w momencie generowania planu treningowego przez zalogowaną osobę.

*Enumy* zawierają w sobie następujące wartości:

* ***difficulty*** – (*easy,* *medium*, *hard*),
* ***exercise\_type*** – (*strength*, *cardio*, *mobility*),
* ***muscle\_group*** – (*back*, *legs*, *belly*, *biceps*, *triceps*, *buttocks*, *chest*, *calves*, *shoulders*, *whole\_body*),
* ***sex*** – (*female*, *male*).



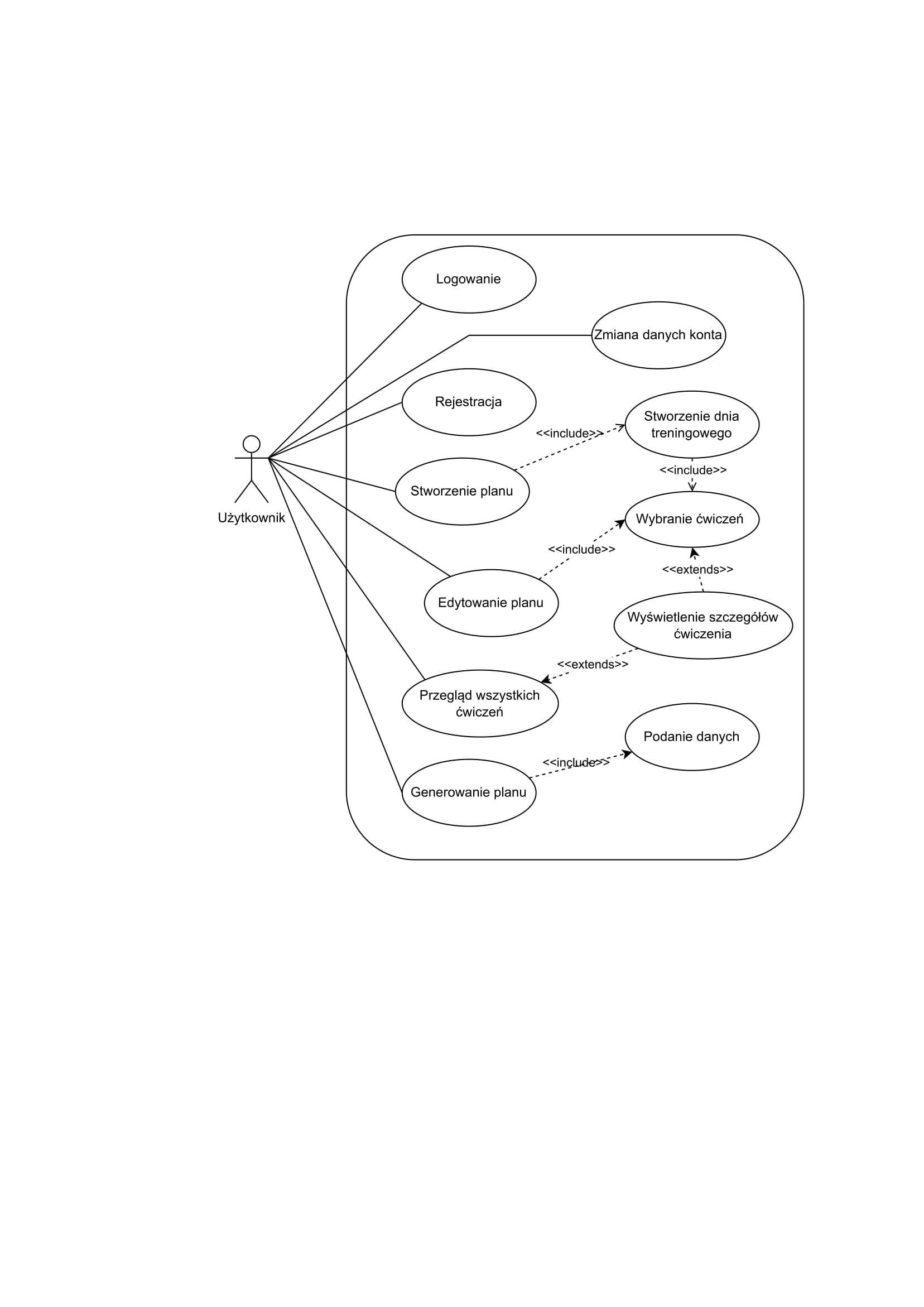
Rysunek 4: Diagram związków encji

## Diagram przypadków użycia

Przypadek użycia to opis interakcji aktora z systemem przy pomocy języka naturalnego. Może przyjmować różną formę. Najprostszą z nich to pojedyncze zdanie określające cel danego przypadku. Zachowanie może być opisane za pomocą informacji tj. [1]:

* nazwa przypadku użycia,
* aktor biorący udział w interakcji,
* scenariusz.

Diagram przypadków użycia przedstawiony jest na Rysunek 5.



Rysunek 5: Diagram przypadków użycia

## Opis i definicja scenariuszy przypadków użycia

Diagram przypadków użycia przedstawia jedynego aktora w systemie - *Użytkownika*. Logowanie, rejestracja oraz zmiana danych konta to najprostsze przypadki użycia w systemie. Wymagają podania prawidłowych danych przez użytkownika. Użytkownik ma możliwość tworzenia i edytowania planu treningowego, które wymagać będzie wyboru ćwiczeń. Wybór ćwiczeń może zostać rozszerzony o wyświetlenie szczegółów danego ćwiczenia, jeśli użytkownik chciałby się o nim więcej dowiedzieć. Szczegóły ćwiczenia są też dostępne z poziomu przeglądu wszystkich ćwiczeń. Generowanie planu możliwe jest po wcześniejszym podaniu danych.

Definicja scenariuszy przypadków użycia znajduje się w Załączniku 1. *Scenariusze przypadków użycia*

# Interfejs użytkownika

Najważniejszym aspektem każdej strony internetowej jest jej czytelność, przejrzystość i łatwość użytkowania. Aplikacja musi być intuicyjna w obsłudze, ponieważ starsze osoby, które z niej korzystają mogą mieć problem, by się po niej poruszać. Skomplikowany i nieczytelny interfejs może powodować frustracje. By jej uniknąć aplikacja została stworzona ze zrozumiałych, dużych i podpisanych komponentów.

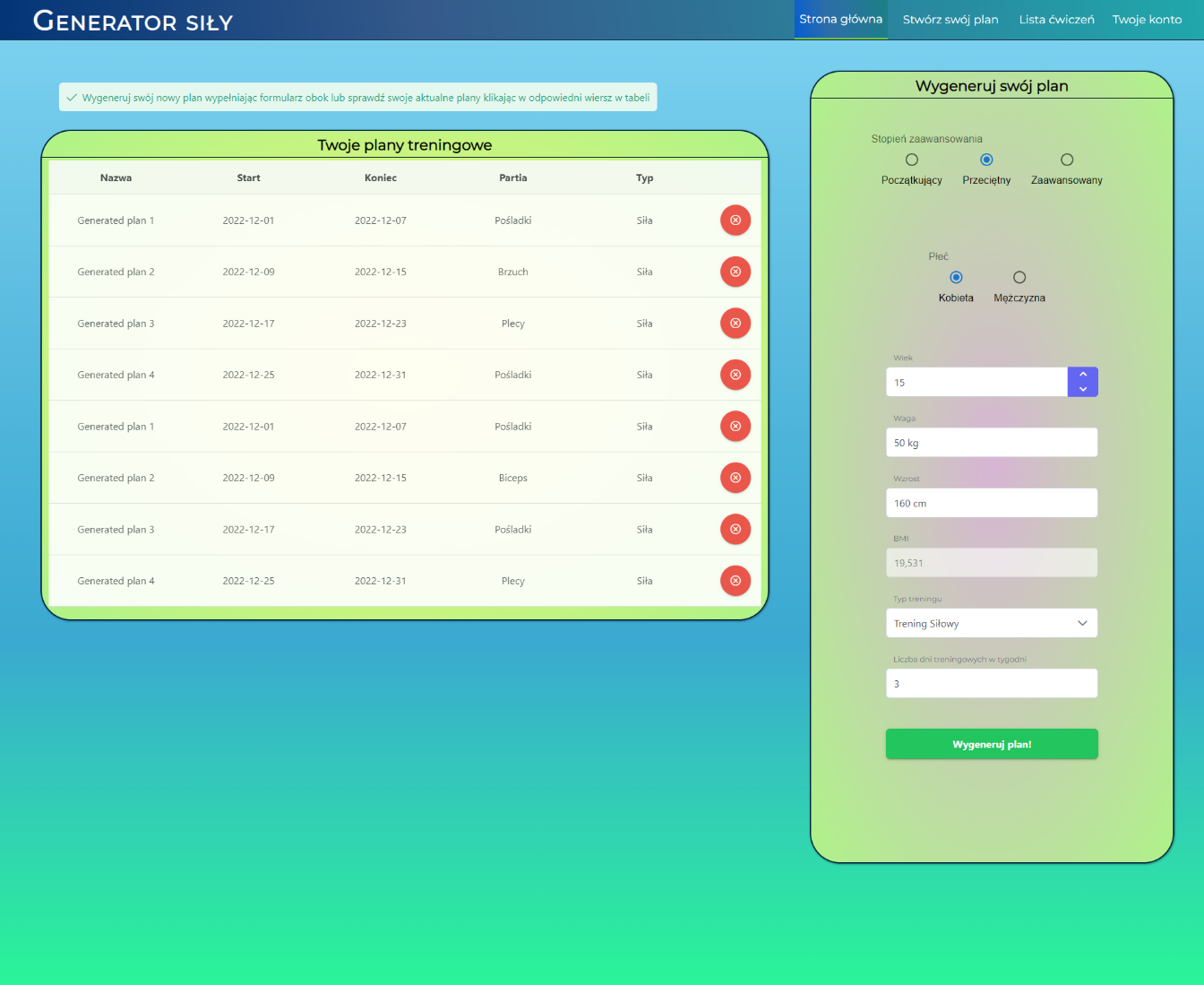
## Widok logowania, rejestracji i edycji danych użytkownika

Interfejs logowania jest jednym z najprostszych ekranów całej aplikacji, zarówno jeśli chodzi o stopień złożoności, jak i jego funkcjonalności. Na tym ekranie użytkownik ma możliwość zalogowania się do aplikacji. Jeśli użytkownik nie posiada założonego konta, może zarejestrować się. W tym celu musi podać login, email i hasło.

Użytkownik po wciśnięciu przycisku *Twoje* *konto*, które znajduje się na pasku menu nawigacji, ma możliwość przejścia na stronę edycji swojego konta. Edycja następuje w momencie, gdy osoba korzystająca z aplikacji poda nowy login, email lub hasło, a następnie kliknie przycisk *Zmień dane*. Encja użytkownika zostaje wtedy nadpisana o nowe wartości.

## Widok menu głównego

Po pomyślnym zalogowaniu użytkownik zostanie przeniesiony do menu głównego. Na tym ekranie osoba korzystająca z aplikacji może przeglądać swoje plany treningowe, przejść na stronę listy dni treningowych lub wypełnić dane w sekcji *Wygeneruj swój plan* , by po kliknięciu przycisku *Wygeneruj plan* otrzymać nowe, stworzone przez system plany treningowe.



Rysunek 6: Ekran menu głównego

## Widok listy dni treningowych z planu treningowego

W celu sprawdzenia dni treningowych w danym planie użytkownik musi kliknąć w wiersz z wybranym planem znajdującym się w sekcji *Twoje plany treningowe* (Rysunek 6). Na ekranie użytkownik może przeglądać listę dni treningowych z wybranego wcześniej planu (Rysunek 7). Z danych z tabeli może dowiedzieć się, na jaki dzień przypada odpowiedni dzień treningowy, jaki będzie najczęstszy typ wykonywanych ćwiczeń oraz jaka będzie najczęściej trenowana partia mięśniowa.

Obraz zawierający stół

Opis wygenerowany automatycznie

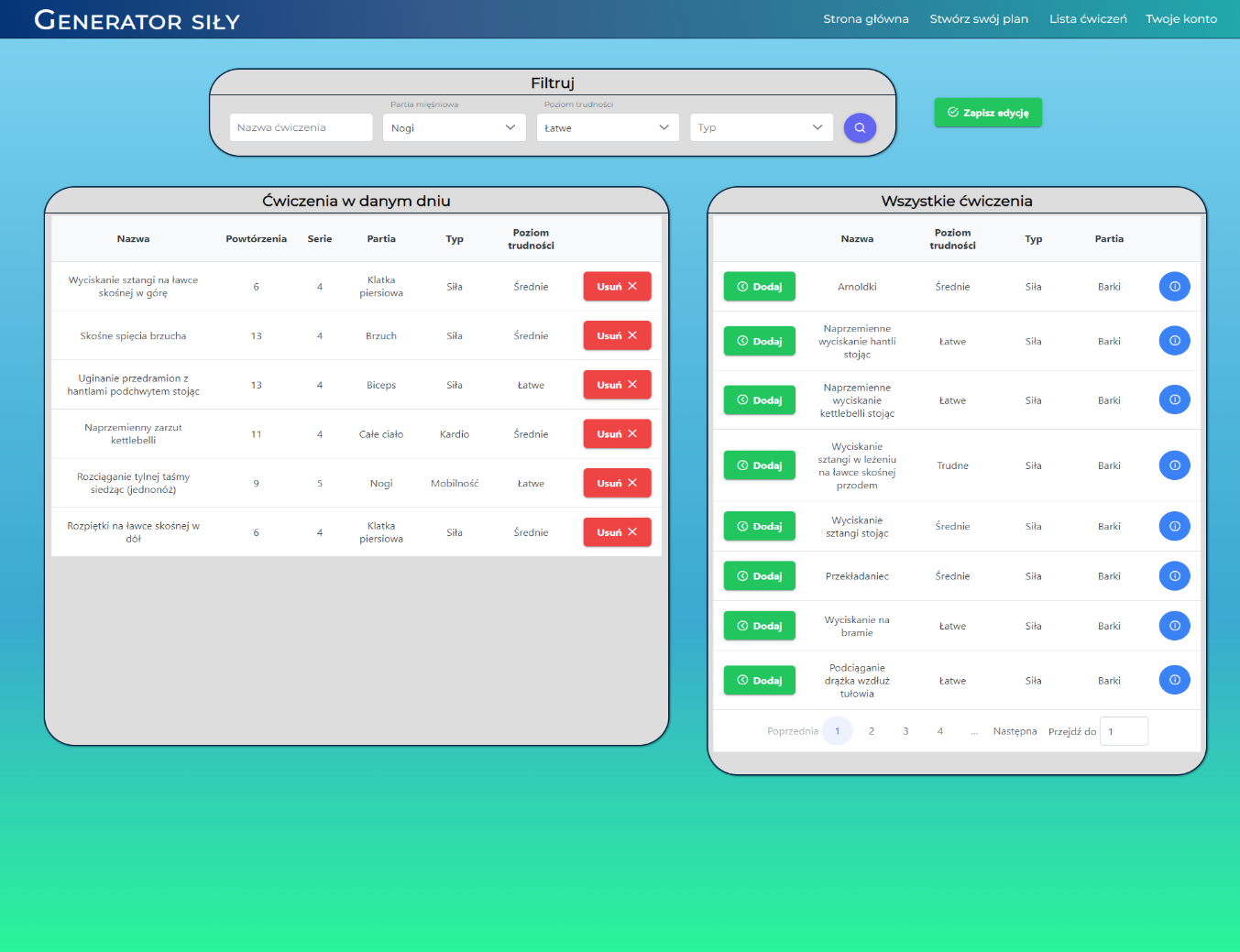
Rysunek 7: Ekran listy dni treningowych

## Widok edycji dnia treningowego

Jednym z wymagań funkcjonalnych aplikacji było udostępnienie użytkownikowi opcji edytowania planu treningowego. Oprócz możliwości usuwania planu lub dnia treningowego, które widzimy odpowiednio na Rysunek 6 i Rysunek 7, istnieje sposób edytowania dnia treningowego, który widzimy na Rysunek 8. Użytkownik może filtrować ćwiczenia znajdujące się w sekcji *Wszystkie* *ćwiczenia*. Wciskając przycisk *Dodaj* wywołuje się dialog omówiony w akapicie *WIDOK DIALOGU PODAWANIA LICZBY POWTÓRZEŃ I SERII*. Po podaniu liczby serii i powtórzeń nowo stworzone ćwiczenie z danymi pojawia się w sekcji *Ćwiczenia* *w* *danym* *dniu*. W tej sekcji znajdują się też przyciski służące do usunięcia ćwiczenia z dnia treningowego. Osoba korzystająca z aplikacji kończy edycję wciskając przycisk *Zapisz* *edycję*.

## Widok dialogu podawania liczby powtórzeń i serii

Podczas tworzenia planu treningowego i edycji dnia treningowego, gdy użytkownik dodaje ćwiczenie do swojego dnia treningowego, wyświetlany zostaje dialog, w którym można podać liczbę powtórzeń oraz serii, by możliwe było stworzenie ćwiczenia z danymi, które przypisane będzie do dnia treningowego.



Rysunek 8: Ekran edycji dnia treningowego

## Widok listy ćwiczeń z danego dnia treningowego

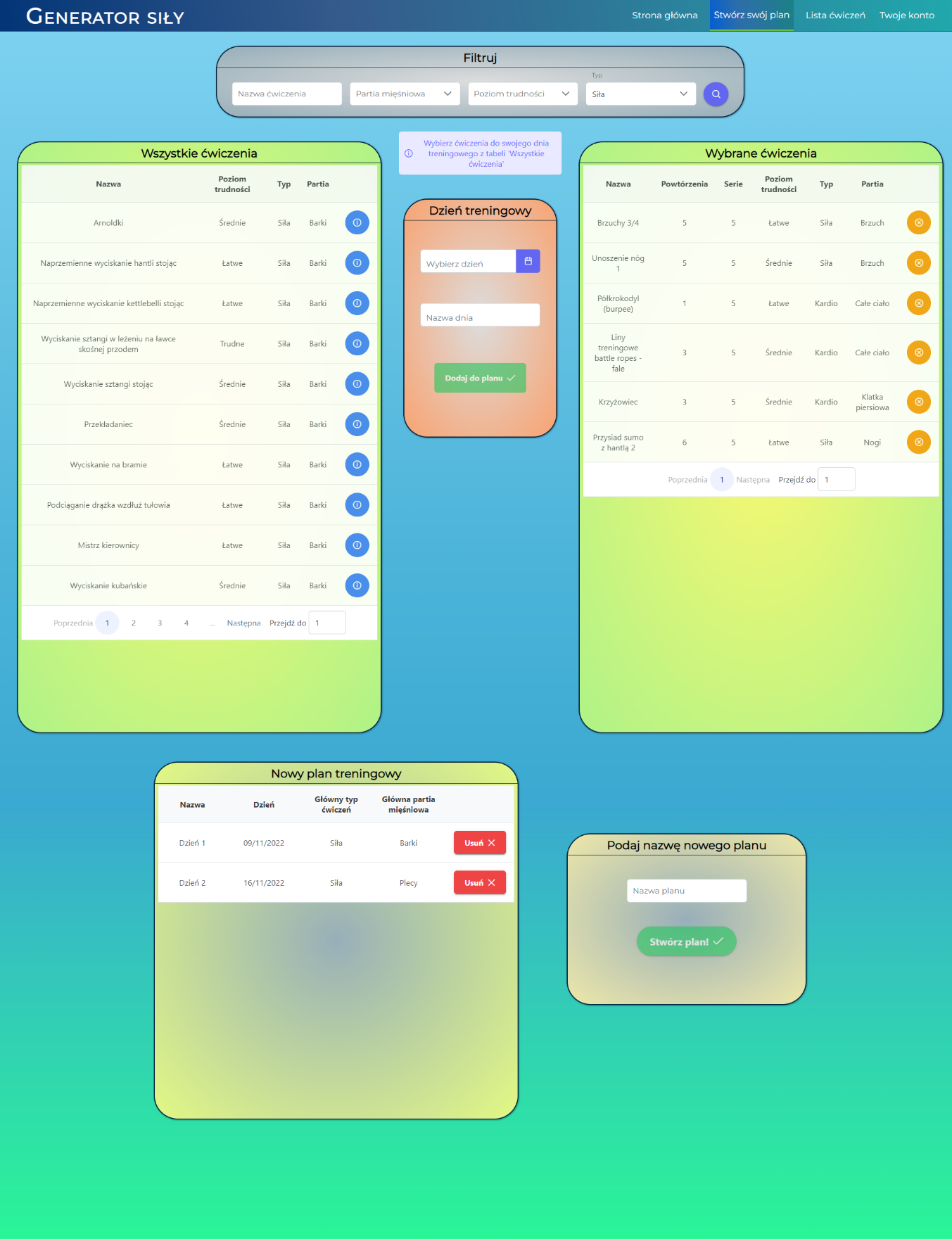
Z ekranu listy dni treningowych (Rysunek 7) użytkownik może zobaczyć ćwiczenia z danego planu klikając na wybrany wiersz z tabeli w sekcji *Dni* *treningowe*. Rysunek 9 przedstawia sposób wyświetlania ćwiczeń z danymi z wybranego wcześniej dnia treningowego. W wiersze z tabeli można kliknąć. Spowoduje to przeniesienie użytkownika na ekran, gdzie wyświetlone są szczegóły ćwiczenia (Rysunek 12).



Rysunek 9: Ekran listy ćwiczeń z danego dnia treningowego

## Widok ekranu ręcznego tworzenia planu treningowego

Najtrudniejszym i najbardziej złożonym do zaimplementowania był ekran tworzenia planu treningowego (Rysunek 10). Znajduje się w nim najwięcej sekcji oraz logiki biznesowej.



Rysunek 10: Ekran tworzenia planu treningowego

Na tej stronie można wyróżnić pięć głównych sekcji potrzebnych do stworzenia planu. Pierwsza z nich to *Wszystkie* *ćwiczenia*. Użytkownik wybiera z niej wiersze, które mógł wcześniej filtrować w celu wyszukania odpowiedniego ćwiczenia. Kliknięcie w wiersz tabeli spowoduje wywołanie dialogu z akapitu *WIDOK DIALOGU PODAWANIA LICZBY POWTÓRZEŃ I SERII*.. Po podaniu liczby serii i powtórzeń pokazuje się ono w sekcji 2 – *Wybrane* *ćwiczenia*. Tabela z tej części strony przechowuje dodane przez użytkownika ćwiczenia. Za pomocą żółtego przycisku znajdującego się z prawej strony wierszy, osoba korzystająca z aplikacji może usunąć ćwiczenia z danymi.

W sekcji 3 – *Dzień* *treningowy* należy podać datę i nazwę dnia treningowego do stworzenia.

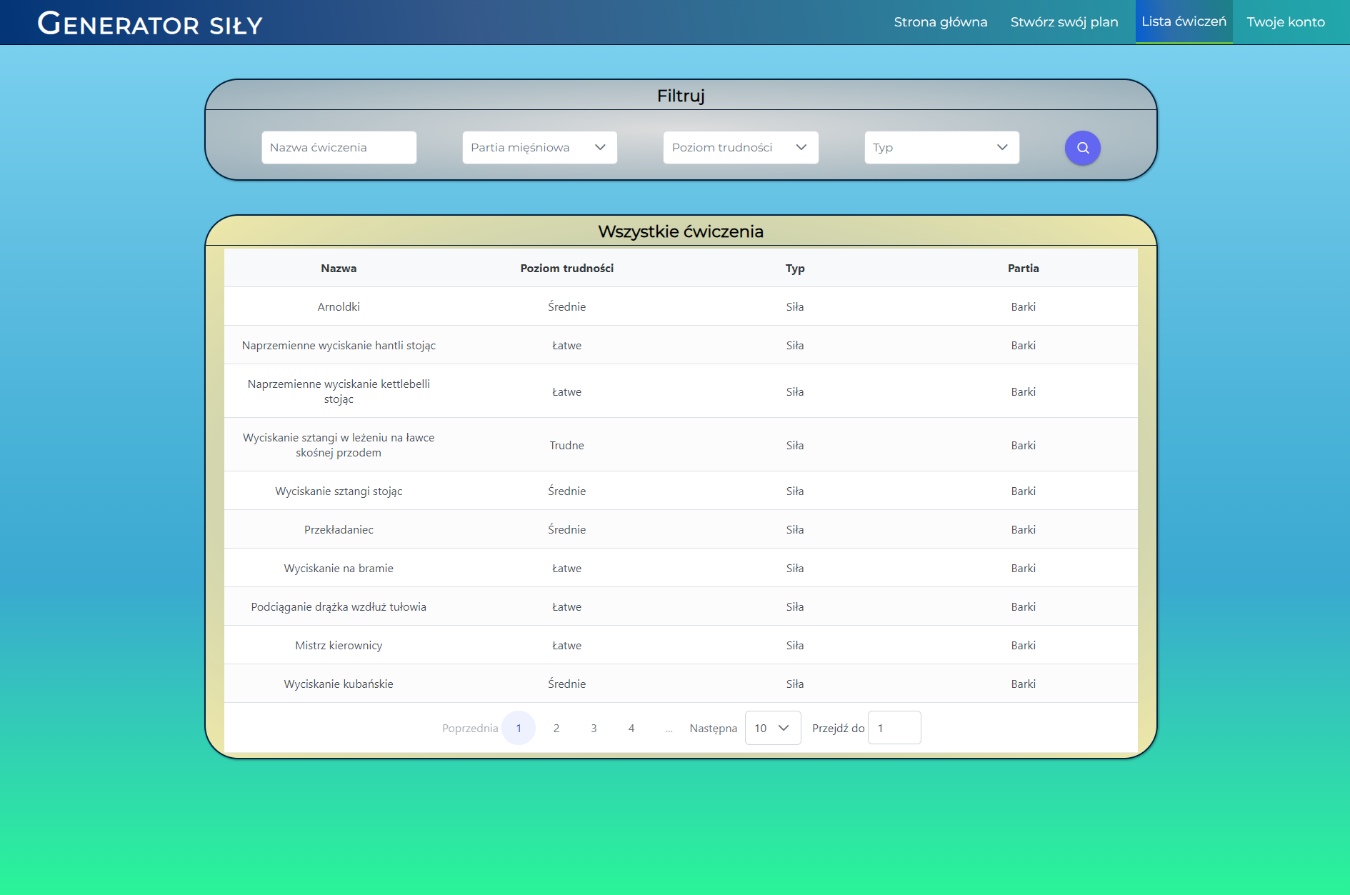
Data i nazwa dnia treningowego podczas tworzenia planu nie może się powtarzać. Podanie złych danych skutkować będzie wyświetleniem komunikatu z szczegółami błędu.

Stworzony dzień treningowy wyświetli się w sekcji 4 – *Nowy* *plan* *treningowy*. Lista zawiera informację m.in. o nazwie dnia treningowego i daty, w którym ćwiczenia powinny być wykonane.

Sekcja 5 – *Podaj* *nazwę* *nowego* *planu* to ostatni etap tworzenia planu. Po podaniu nazwy i kliknięciu przycisku *Stwórz* *plan* następuje przekierowanie do menu głównego.

## Widok listy wszystkich ćwiczeń

Czasem użytkownik przed wygenerowaniem lub ręcznym stworzeniem planu będzie chciał przejrzeć ćwiczenia, które oferuje aplikacja w celu zapoznania się z nimi, by w przyszłości łatwiej i szybciej tworzyć plan treningowy. Ekran listy wszystkich ćwiczeń znajduje się na Rysunek 11.



Rysunek 11: Ekran listy wszystkich ćwiczeń

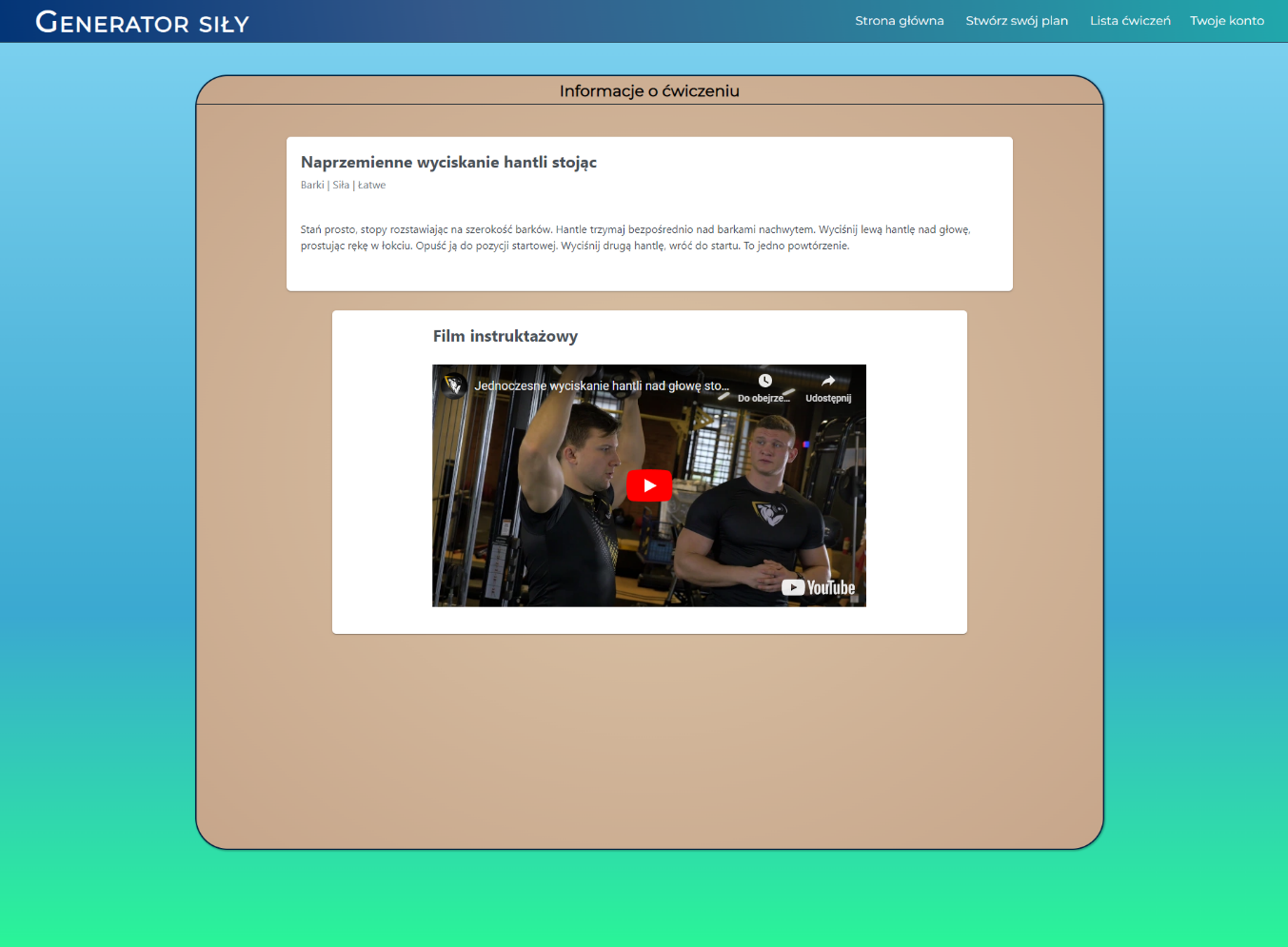
Tabela, w której wyświetlane są ćwiczenia jest wyposażona w komponent paginujący dane, co oznacza, że osobie korzystającej z aplikacji nie zostanie wyświetlonych ponad 250 ćwiczeń z bazy danych naraz. Paginator dzieli dane, które tabela ma wyświetlić na strony, w zależności od tego jaką liczbę ćwiczeń na jednej stronie ma pokazać użytkownikowi. W sekcji *Wszystkie* *ćwiczenia* wyświetlone jest 10 wierszy jednocześnie, zgodnie z oznaczeniem na spodzie tabeli.

Na tym ekranie znajduje się jeszcze jeden komponent. Służy on do filtrowania ćwiczeń, które są wyświetlane w tabeli. Przedstawienie danych spełniających zadane kryteria oraz ukrywanie tych niepotrzebnych jest istotną funkcjonalnością każdej strony internetowej. Jej zadaniem jest ułatwienie użytkownikowi znajdywania interesujących go treści. Istnieje możliwość filtrowania ćwiczeń na 4 sposoby:

* po nazwie,
* po partii mięśniowej,
* po trudności,
* po typie.

## Widok danych szczegółów ćwiczenia wraz z filmem instruktażowym

Celem zaimplementowanej aplikacji jest pomoc w tworzeniu planu treningowych dla osób, które nie miały wcześniej większej styczności z ćwiczeniem i aktywnością fizyczną. W tabelach funkcjonujących w aplikacji wyświetlane są m.in. nazwy ćwiczenia i określenie, na jaką partię mięśniową to ćwiczenie oddziałuje. Dla niedoświadczonych trenujących może być to słaby wyznacznik, jak powinien wyglądać przebieg danego zadania. Strona szczegółów ćwiczenia przedstawiona na Rysunek 12 zaimplementowana została z myślą o takich właśnie ludziach. W centrum uwidaczniają się dwie sekcje. Jedna z nich przedstawia nazwę ćwiczenia, jego atrybuty oraz przebieg/opis wykonania. Druga zawiera film instruktażowy zaczerpnięty z serwisu *Youtube*. Wiedza, którą użytkownik może dowiedzieć się z opisu ćwiczenia lub video, powinny wystarczyć, by mógł samodzielnie i bezpiecznie wykonać zadane ćwiczenie.



Rysunek 12: Ekran szczegółów ćwiczenia

# Wykorzystane technologie

Wybór technologii odpowiednich do potrzeb projektu informatycznego jest procesem złożonym i zależnym od wielu czynników, takich jak użyteczność czy znajomość rozwiązań. Pod pojęciem technologii kryje się szeroki zakres dostępnych metod i narzędzi. Zaliczają się do nich języki programowania, biblioteki, frameworki oraz narzędzia skoncentrowane na obsłudze baz danych i służące do testowania zaimplementowanych rozwiązań w obszarze funkcjonowania aplikacji internetowej. Z perspektywy budowania programu, technologie pełnią istotną rolę w kwestii usprawnienia przepływu informacji i udoskonalenia przebiegających procesów. Ponadto umożliwiają użytkownikom uzyskanie łatwego dostępu do danych odpowiedniego typu, co bezpośrednio wpływa na relacje z klientem oraz ogólny odbiór aplikacji przez użytkownika. Warto zaznaczyć, że technologie funkcjonujące w branży IT są znacznym udogodnieniem dla pracy programistów i pozwalają na szybkie wdrażanie opracowanych koncepcji.

## Wybrane języki

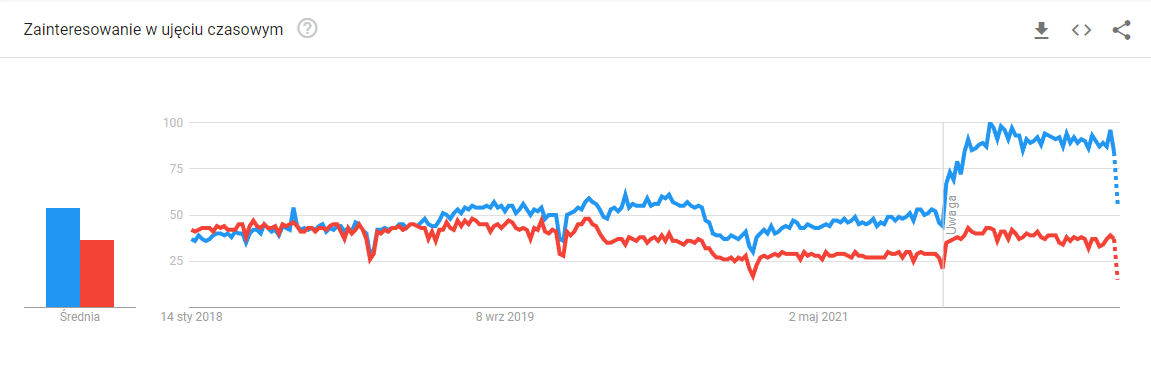
Poza językiem SQL wykorzystanym w skryptach tworzących tabele w bazie danych, aplikacja korzysta z dwóch języków – *Javy* i *Javascriptu*. Pomimo podobnej składni są to różne języki.

*Java* to uniwersalny, współbieżny, obiektowy język programowania oparty na klasach. Jego prosta składnia sprawia, że większość programistów może względnie szybko osiągnąć biegłość w tym języku. *Java* ma wiele wspólnego z językami *C* i *C++*. Jest typowany, co oznacza, że każda zmienna musi mieć zdefiniowany typ (np. *int*, *boolean*). Dzięki temu można rozróżnić dwa typy błędów zachodzących podczas pisania kodu – te występujące w trakcie kompilowania (ang. *compile-time errors*) i w trakcie działania aplikacji (ang. *run-time errors*). Kompilowanie kodu składa się z przetłumaczenia programu na reprezentację kodu bajtowego niezależną od maszyny. *Java* jest językiem wysokiego poziomu. Obsługuje automatyczne zarządzanie pamięcią, najczęściej za pomocą czyszczenia pamięci (ang. *garbage* *collector*), by uniknąć problemów związanych ze zwalnianiem pamięci (które występują w językach *C* i *C++*). [3]

*Javascript* jest językiem programowania sieci. Przeważająca część stron internetowych używa tego języka, a każda z współcześnie używanych przeglądarek jest wyposażona w interpreter *Javascriptu*, czyniąc go najczęściej wdrażanym językiem programowania w historii. W ciągu ostatniej dekady Node.js przyczynił się do używania *Javascriptu* poza przeglądarkami. Jego ogromny sukces oznaczał, że *Javascript* jest najczęściej używanym językiem programowania wśród programistów. *Javascript* jest dynamicznym, interpretowanym językiem wysokiego poziomu, który idealnie wpasowuje się w obiektowe i funkcjonalne style programowania. Jego składnia opiera się na *Javie*, lecz w przeciwieństwie do niej, *Javascript* jest nietypowany. Języki te nie mają ze sobą nic więcej wspólnego. [4]

## Narzędzia strony interfejsu użytkownika

Obecnie na rynku istnieje wiele bibliotek do języka *Javascript*, które ułatwiają i przyspieszają proces pisania kodu. Najpopularniejszymi z nich są *Angular* oraz *ReactJS*. Analizując dane przedstawione na stronie *trends.google.pl* (Rysunek 14), łatwo można wywnioskować, że na przestrzeni ostatnich czterech lat to *React* cieszy się większym zainteresowaniem. Ten fakt oraz chęć nauczenia się nowej technologii, zadecydowały o wyborze biblioteki, która została użyta do stworzenia strony internetowej.

****

Rysunek 13: Porównanie zainteresowania Angularem (linia czerwona) i ReactemJS (linia niebieska)

### ReactJS

*React* jest popularną biblioteką używaną do tworzenia interfejsu graficznego użytkownika. Stworzyła go firma *Facebook*, która borykała się z problemami związanymi z wysoko skalowalnymi aplikacjami opierających się na dużej ilości danych. *React* został wydany w 2013 roku.

Biblioteka nie dostarcza ze sobą dużej ilości narzędzi, których można by się spodziewać po tradycyjnym frameworku języka *Javascript.* Oznacza to, że deweloper ma dowolność w sprawie importowania dodatkowych narzędzi i bibliotek. [5]

### Biblioteki do budowania komponentów

Aplikacja korzysta z dwóch dodatkowych bibliotek, mających na celu wizualną poprawę sekcji wyświetlanych na stronie. Pierwszą z nich jest *PrimeReact*, który jest bogatym zbiorem natywnych komponentów *Reacta* o otwartym kodzie źródłowym. W projekcie użytych jest wiele komponentów z tej biblioteki, m.in.:

* *InputText,*
* *Button,*
* *Toast,*
* *Message,*
* *DataTable,*
* *Dialog,*
* *Card.*

Instalacja *PrimeReact* jest bardzo prosta. Wystarczy w otwartym terminalu w folderze z projektem wpisać komendę: *npm install primereact primeicons*.

Drugą z bibliotek używanych w projekcie jest *Material UI.* Oferuje ona wyczerpujący zestaw narzędzi UI, które pozwalają szybciej dostarczać nowe funkcjonalności. W projekcie użyto czterech komponentów:

* *FormControl,*
* *FormLabel,*
* *FormControlLabel,*
* *RadioGroup.*

## Narzędzia strony serwerowej

Wybór narzędzia *Springboot* do stworzenia strony serwerowej był znacznie prostszy. Spowodowany był on doświadczeniem w pracy z tym frameworkiem nabytym w trakcie trwania studiów.

Aplikacja serwerowa została stworzona przy użyciu narzędzia *Maven*, które pozwala na automatyzację oprogramowania na platformę *Java.*

### Springboot

*Spring* to framework z otwartym kodem źródłowym, którego pierwsza wersja stworzona i opisana została w 2002 roku przez Rod Johnsona. *Spring* odpowiedzialny jest za wiele rzeczy, ale jego główna, fundamentalna misja brzmi: *Spring ma ułatwić rozwój aplikacji Java*. Framework ten unika zaśmiecania kodu aplikacji swoim API i prawie nigdy nie zmusza użytkownika do implementowania interfejsu specyficznego dla *Springa*. Aplikacja opierająca się na tym narzędziu często nie wskazuje na to, że opiera się właśnie na nim. W najgorszym przypadku klasa może mieć jedną z jego adnotacji, ale poza tym jest ona klasą POJO (ang. *Plain Old Java Object*)[[1]](#footnote-1).

Spring opiera się o wstrzykiwanie zależności (ang. *dependency injection*). Celem głównym tego zabiegu jest uproszczenie kodu, sprawienie, że jest łatwiejszy do zrozumienia oraz testowania. [6]

Można zauważyć, że w ramach dodania Springboot do projektu dodaje się zależności o artefaktach zakończonych skrótami *JDBC* oraz *JPA*.

*JDBC* to interfejs API dla języka *Java*. Umożliwia łączenie się z tradycyjnymi relacyjnymi bazami danych. *Spring JPA* dostarcza narzędzia obsługi repozytorium danych dla *Java Persistance API* (*JPA*). Zaletą tego rozwiązania jest ułatwienie procesu zarządzania dostępem do źródeł danych.

### Mapstruct

Narzędzie to jest generatorem kodu, który ułatwia implementacje mapowań pomiędzy klasami *Java*. *Mapstruct* używa prostych metod, co sprawia, że jest szybki i prosty do zrozumienia. Użytkownik za pomocą adnotacji ma pełną kontrolę nad generowaniem funkcji mapujących.

### Lombok

Lombok to przydatne narzędzie ułatwiające utrzymanie czystości kodu poprzez automatyczne generowanie funkcji dostępu do zmiennych i konstruktorów. Przy użyciu prostych adnotacji nad definicją klasy, np. *@Getter,* przy uruchomieniu aplikacji automatycznie wygenerowany zostanie kod, dzięki któremu będziemy mogli pobrać prywatne zmienne danej klasy.

### Konfiguracja Lombok i Mapstruct

Korzystanie jednocześnie z *Lomboka* i *Mapstructa* może okazać się problematyczne. Jeśli niepoprawnie skonfigurujemy te narzędzia, generowanie funkcji mapujących nie powiedzie się. *Mapstruct* korzysta bowiem z funkcji *GET* i *SET.* Błąd spowodowany jest tym, że jeśli *Mapstruct* uruchomi się jako pierwszy, to funkcje *Lomboka* nie zostały jeszcze wygenerowane. Mapowanie po prostu nie ma prawa się udać, bo nie istnieją jeszcze funkcje dostępu do zmiennych. Rozwiązanie jest trywialne, wystarczy w sekcji *plugins* zadeklarować te narzędzia w odpowiedniej kolejności.

## Narzędzia bazy danych

Baza danych to zorganizowany zbiór danych. Dane zapisuje i odczytuje się elektronicznie. Design bazy danych obejmuje techniki formalne i względy praktyczne, wliczając w to modelowanie danych, wydajne reprezentowanie danych i ich przechowywanie, języki *query*, bezpieczeństwo i prywatność poufnych danych.

### PostgreSQL

Istotą każdej aplikacji, która nawiązuje interakcje z użytkownikiem jest pokazywanie, pobieranie, usuwanie i aktualizowanie odpowiednich informacji wyświetlanych na UI. Nie byłoby to możliwe bez narzędzia przechowującego dużej ilości danych.

*PostgreSQL* to system zarządzania relacyjnej bazy danych (ang. *object-relational database management system*) ORDBMS oparta na POSTGRES, opracowana na Uniwersytecie Kalifornijskim w Berkeley na wydziale informatyki.

*PostgreSQL* to narzędzie o otwartym kodzie źródłowym. Wspiera dużą część standardów SQL oferując przy tym wiele nowych opcji, tj.:

* rozbudowane zapytania *query*,
* klucze obce,
* wyzwalacze,
* widoki, które można zaktualizować.

Na dodatek, *PostgreSQL* może być rozszerzony przez użytkownika o nowe:

* typy danych,
* funkcje,
* operatory,
* metody indeksowe,
* języki proceduralne. [7]

Aplikacja *springboot* w celu połączenia się z bazą danych *PostreSQL* musi mieć skonfigurowaną zależność w pliku *pom.xml*.

### Flyway

Po raz pierwszy uruchomiona aplikacja *springboot,* po połączeniu z bazą danych, powinna być gotowa do interakcji z użytkownikiem. By to umożliwić, skorzystano z narzędzia *Flyway*.

*Flyway* to narzędzie do migracji bazy danych. Opiera się o siedem prostych komend: *Migrate*, *Clean*, *Info*, *Validate*, *Undo*, *Baseline* i *Repair*. Migracje mogą być napisane w języku SQL lub Java. Jedną z wspieranych baz danych jest oczywiście PostgreSQL. [8].

Migracja *Flyway* polega na uruchomieniu skryptów wykonywanych na bazie danych w trakcie uruchamiania aplikacji. By było to możliwe należy odpowiednio skonfigurować plik *pom.xml*.

### Docker

Nie każdy użytkownik aplikacji będzie miał zainstalowane narzędzia *PostgreSQL* i *PgAdmin* (graficzne narzędzie zarządzania serwami z bazami danych *PostgreSQL*). By aplikacja mogła zawsze połączyć się z bazą danych użyto narzędzia *Docker*.

*Docker* to technologia wirtualizacji kontenerów, innymi słowy jest to maszyna wirtualna zajmująca bardzo mało przestrzeni dyskowej. *Docker* powstał w celu rozwiązania problemu, którym jest ogromna ilość pamięci zajętej przez zwykłą maszynę wirtualną. Maszyna wirtualna obejmuje zwykle system operacyjny, na którym można znaleźć aplikację. Powoduje to kilka problemów dla szybkości i wydajności. *Docker* ma na celu wyprodukowanie środowiska, które będzie lekkie oraz bardziej elastyczne. Kontener *Dockerowy* uruchamia się szybko i możliwe jest uruchomienie wielu kontenerów jednocześnie. Powoduje to dużą skalowalność tego narzędzia.

Jeden z przypadków, dla którego deweloper będzie chciał użyć *Dockera* to ciągła integracja i ciągłe wdrażanie. Będąc tak lekkim zasobem, programiści mogą zbudować wiele kontenerów na ich maszynach, które będą kopią niektórych środowisk produkcyjnych. [9]

W aplikacji *docker* stworzy środowisko, które będzie zawierało bazę danych *PostgreSQL*, do której zostaną wykonane skrypty za pomocą narzędzia *Flyway*. Jest to bardzo wygodne dla programisty, który w celu uruchomienia całej aplikacji od strony *backendu* i bazy danych będzie musiał wcisnąć dwa przyciski lub w gorszym przypadku wpisać dwie komendy w terminalu. Jedyne, co będzie od niego wymagane, to zainstalowane narzędzie *Docker* na swojej maszynie.

## Narzędzia testowania

Poprawność działania aplikacji jest najistotniejszym elementem dla przeciętnego użytkownika. Aplikacja została przetestowana od strony *backendowej* za pomocą narzędzi *JUnit* i *Mockito*, które są najlepszym wyborem dla aplikacji *springboot.* Testując aplikacje programista jest w stanie wychwycić niezgodności oraz poprawić wydajność i czytelność kodu.

### JUnit

*JUnit* to jeden z najbardziej popularnych framworków unit-testowych dla języka Java. Pierwsza wersja tego narzędzia została wydana w 1997 roku. Od tamtego czasu *JUnit* stał się standardem, który został zaimplementowany w wielu innych językach.

Wersja 4.0 ułatwiła i usprawniła testy jednostkowe poprzez odejście od konwencji nazewnictwa oraz wprowadzenie wielu innych funkcji, jak *time-outy* oraz testowanie wyjątków. *JUnit* *4.0* wydano w 2006 roku, a jego ostatnia aktualizacja 4.12 została wydana w grudniu 2014.

*JUnit 5* składa się z trzech podprojektów [10]:

* JUnit Platform – dostarcza fundamenty do uruchamiania JVM frameworków testowych,
* JUnit Jupiter – dostarcza nowy model pisania testów, implementuje API *TestEngine* zdefiniowane przez JUnit Platform, dzięki temu można uruchomić testy Junit,
* JUnit Vintage – dostarcza implementacje *TestEngine* dla wersji 3 i 4 JUnit.

Dodanie narzędzia testowego do projektu wymagało dodania zależności i pluginu do pliku *pom.xml*.

### Mockito

*Mockito* to względnie nowy framework do mockowania. Został wydany pod koniec 2007 roku i szybko stał się produktem wysokiej jakości. Oferuje pełną kontrolę nad procesem mockowania. [11]

Ale czym tak właściwie jest mock? Mock to obiekt, którego używa się zamiast rzeczywistej implementacji w trakcie testów jednostkowych [12]. W trakcie testów aplikacji *springboot* używa się wywołań funkcji repozytoriów operujących na danych w bazie danych lub innych interfejsów. Operacje na bazie danych są niepożądanym efektem testów jednostkowych. *Mockito* pozwala na podstawienie sztucznego interfejsu, który będzie udawał wykonywanie funkcji, zwracając dane, które wcześniej zdefiniujemy w teście.

# Implementacja aplikacji

Implementacja i testy rozwiązania są ostatnimi etapami projektu informatycznego. Implementacja polega na przeniesieniu założeń projektowych i dokumentacji do kodu, który swoim prawidłowym działaniem zwieńczy zakończenie pracy nad projektem.

## Środowiska programistyczne

Aplikacja została napisana w dwóch środowiskach programistycznych, osobno dla *backendu* – strony serwerowej oraz *frontendu* – strony interfejsu użytkownika.

W sieci można spotkać wiele dobrych IDE dla języka Java, lecz jeden z nich nie ma sobie równych. Jest nim środowisko *Intellij* firmy Jetbrains, które dzięki licencji Politechniki Wrocławskiej zainstalowano w wersji *Ultimate*. W tym środowisku można wygodnie korzystać z aplikacji opartych o *Mavena*, ponieważ jest z nim w pełni zintegrowane, co pomaga w zautomatyzowaniu budowania projektu. W *Intellij* można połączyć się z bazą danych i bezpośrednio wykonywać skrypty z konsoli wbudowanej w to narzędzie. Dodając do tego przejrzysty, nowoczesny interfejs, jest to najlepsze IDE do budowania *backendu* w *Javie* na rynku.

Po stronie *frontendowej* użyto *Visual Studio Code* będące lekkim narzędziem, które można spersonalizować na mnóstwo sposobów. Jest to obecnie najpopularniejsze, darmowe środowisko programistyczne, jakie można znaleźć w sieci.

## Podział na pakiety i strony

### Backend

Pomysłem na strukturę serwera był podział klas i interfejsów na pakiety, gdzie każdy z pakietów miałby odpowiadać jednej tabeli w bazie danych. Dodatkowo zawarto pakiet *common*, który zawiera wspólne klasy wyjątków oraz *enumy*. Pakiet *pwr* ma w sobie dodatkowo dwie klasy *PlanGeneratorApplication*, która zawiera metodę *main()* i uruchamia serwer oraz *WebConfig*, który służy do nadania uprawnień źródłowi żądań *http://localhost:3000*, by *frontend* i *backend* mogły komunikować się ze sobą za pomocą *REST* API.

Użyty w projekcie podział na pakiety spowodowany jest tym, że by móc poprawnie zmapować klasy Javy do tabel w bazie danych i móc z nich korzystać, potrzebujemy następujących plików:

* klasy *Entity*, która 1 do 1 odpowiada encji w bazie danych,
* interfejsu rozszerzającego interfejs *JPARepository,*
* klasy posiadającej adnotację *@RestController*, która obsługiwać będzie zapytania REST,
* ogólnej klasy obsługującej logikę zawartą w danym pakiecie, która sprawia, że kod jest bardziej podzielony, przez co bardziej czytelny i prostszy do zrozumienia,
* innych obiektów dto[[2]](#footnote-2), translatorów, serwisów.
* Obraz zawierający tekst

  Opis wygenerowany automatycznie
* Rysunek 14: Struktura pakietów na przykładzie TrainingPlan

Pakiet *TrainingPlan* (jak i każdy inny w projekcie) podzielony został na pakiety:

* *api* - znajdują się w nim klasa *Controller* oraz interfejs *Adapter*. Służą one do obsługi żądań RESTowych. Interfejs *Adapter* rozszerza klasę *Mediator*, której zadaniem jest obsługa logiki danego żądania.
* *exceptions* – klasy w niej zostały stworzone z myślą o tym, by w momencie podania błędnych danych lub wyjątku typu *RuntimeException*, rzucać wyjątki, które dostarczają programiście trochę więcej informacji, by łatwiej mu było zlokalizować przyczynę problemu,
* *model* – został rozszerzony o pakiety *entity* i *dto:*
  + entity – zawiera klasę odpowiadającą tabeli w bazie danych,
  + dto – zawiera klasy, którymi serwer wymienia się z *frontendem*, by ten otrzymał jedynie potrzebne mu informacje i dane.
* *repository* – pakiet przechowujący interfejs, który przy pomocy *JPA*, komunikuje się bazą danych.

Pozostałe klasy, zawarte w pakiecie *TrainingPlan* to m.in. wcześniej wspomniany *Mediator* – klasa rozszerzona o interfejs *Adapter*, *Translator* – interfejs z adnotacją *@Mapper,* który służy do zaimplementowania w nim narzędzia *Mapstruct* oraz pomocnicze klasy obsługujące logikę biznesową.

### Frontend

Strona interfejsu użytkownika została podzielona na foldery *components*, które zgodnie z nazwą zawierą komponenty użyte do budowy strony internetowej oraz *styles*, w której znajdują się pliki o rozszerzeniu *.css* służące do stylizacji komponentów wyświetlanych na stronie.

W folderze *components* wyróżnione są 4 podfoldery:

* *footer* – zawierający plik *Footer.js*, który wyświetla stopkę na stronie,
* *navigation* – zawierający plik *Navbar.js*, odpowiadający za wyświetlanie menu nawigacji znajdującego się na górze strony oraz obsługujący logikę tegoż menu,
* *pages* – występują w nim podfoldery, gdzie każdy z nich odpowiada za inną stronę aplikacji internetowej, np. podfolder *create-plan* odpowiada za logikę i widok strony (Rysunek 10),
* *services* – znajdują się w nim pliki obsługujące logikę, która może zostać użyta wielokrotnie (stworzony z myślą o zasadzie *DRY* – *nie powtarzaj się*).

Każdy z podfolderów folderu *pages* zawiera w sobie kilka plików, które tworzą widok konkretnej strony. Wyjątkiem jest folder *account*, który zawiera w sobie jeszcze foldery *login* i *register.* Strona zbudowana jest w następujący sposób:

1. istnieje plik, który początkowo odpowiada za wyświetlanie pustej strony,
2. do tego pliku importowane są komponenty zbudowane w innych plikach,
3. implementowana jest logika, komunikacji pomiędzy komponentami wyświetlanymi na stronie.

Przykładem może być folder *create-plan*. Zawiera on plik *CreatePlanPage.js*, który wyświetla pozostałe komponenty. Na stronie tworzenia planu znajdują się komponenty:

* filtrowania ćwiczeń – plik *FilterExercises.js,*
* wszystkich ćwiczeń i ćwiczeń użytkownika – ponieważ sekcje te są bardzo podobne udało się je wyświetlać używając tylko komponentu zachowując zasadę *DRY* – plik *ExercisesTable.js,*
* tworzenia dnia treningowego – plik *TrainingDay.js,*
* wyświetlania listy dni treningowych – plik *TrainingPlanTable.js,*
* podawania nazwy do stworzenia planu – plik *CreatePlanSection.js.*

## Implementacja bazy danych

Tworzenie bazy danych należało rozpocząć od zainstalowania *Dockera*. W systemie Windows 10 wystarczyło zainstalować aplikację *Docker Desktop*. W projekcie *backendu* stworzony został plik *docker-compose.yaml*, w którym skonfigurowano tworzenie kontenera bazy danych *PostreSQL*.

W *services* zostaje stworzony kontener *postgres*, w którym ustawiane jest kilka zmiennych:

* *container\_name* – nazwa konteneru,
* *image* – *Postgres* oznacza oficjalny *dockerowy* obraz. Brak podania wersji oznacza, że z strony *https://hub.docker.com/\_/postgres* zostanie pobrana jego najnowsza wersja. *Docker Image* to plik zawierający kod, którego wykonanie zbuduje kontener bazy danych,
* *environment* – w tej sekcji ustawiane zostają zmienne środowiskowe bazy danych,
* *volumes* – by dane zawarte w kontenerze były trwałe należy użyć *volume,* które jest w tworzone i zarządzane przez *dockera*, każdy nowy *volume* zapisywany jest na dysku hosta, co oznacza, że w przypadku zabicia kontenera, nasze dane nie zostaną naruszone.
* *ports* – 8090 oznacza, na którym porcie dostępny będzie baza danych na maszynie lokalnej, 5432 oznacza port po stronie kontenera,
* *networks* – odwołuje się do sieci stworzonej w sekcji *networks,*
* *restart*: określa sposób uruchomienia kontenera.

W pliku *application.properties* ustawiającym właściwości serwera podano adres url, nazwę użytkownika oraz hasło odpowiednio dla serwera i narzędzia *Flyway.* Dodatkową linijką jest *spring.jpa.hibernate.ddl-auto=update*. Dzięki niej, serwer będzie tworzył bazę tylko raz, a później ją nadpisywał. Po zakończeniu pracy serwera nie nastąpi usunięcie każdej z tabel bazy danych. Linijka *spring.flyway.enabled=true* oznacza, że *Flyway* będzie wykonywał skrypty zawarte w plikach *.sql* znajdujących się w folderze *migrations*.

Skrypt tworzący encje i relacje między nimi został wygenerowany bezpośrednio z diagramu relacji encji za pomocą narzędzia *Visual Paradigm*. Na Listing 1 przedstawiono przykładowe wywołanie tworzące tabelę *Exercise*.

**CREATE TABLE** EXERCISE (  
 **id SERIAL NOT NULL**,  
 **exercise\_name varchar**(255) **NOT NULL UNIQUE**,  
 **description varchar**(6000) **NOT NULL**,  
 **url varchar**(255),  
 **muscle\_group MUSCLE\_GROUP NOT NULL**,  
 **exercise\_type EXERCISE\_TYPE NOT NULL**,  
 **difficulty DIFFICULTY NOT NULL**,  
 **is\_For\_Old\_People boolean NOT NULL**,  
 **is\_For\_Obese\_People boolean NOT NULL**,  
 **is\_For\_Young\_People boolean NOT NULL**,  
 **Is\_For\_Women boolean NOT NULL**,  
 **PRIMARY KEY** (**id**)  
);

Listing 1: Skrypt do generowania tabeli Exercise

Skrypt po drobnych poprawkach, zgodnie z konwencją narzędzia *Flyway*, został wklejony do pakietu *db.migration* znajdującego się w folderze *resources*. Plik ze skryptem został nazwany *V1\_\_createTablesInitDefaultData.sql*.

Do poprawnej implementacji bazy danych brakuje jedynie wypełnić ją ćwiczeniami, by użytkownik po zalogowaniu mógł w pełni korzystać z aplikacji. Plik *V2\_\_InsertExerciseData.sql* zawiera ponad 250 komend *Insert* do tabeli *Exercise*.

Po uruchomieniu serwera, połączy się on z bazą danych *PostgreSQL* stworzoną przez *Dockera*, a kończąc inicjalizację narzędzie *Flyway* wykona skrypty znajdujące się w pakiecie *migration*.

## Implementacja serwera

W punkcie 5.2 został opisany sposób podziału plików w pakietach. Odnosząc się do tabeli *Exercise* (Listing 1) przedstawiony zostanie pakiet *exercise*. Klasa odpowiadająca encji (Listing 2) znajdującej się w bazie danych znajduje się w pakiecie *model.entity*. Pierwsze dwie adnotacje oznaczają, że klasa ta odnosi się do encji w bazie danych o nazwie *exercise*. Kolejne oznaczenia pochodzące z rozszerzenia *Lombok* to:

* *Getter* –generowana jest funkcja dostępu dla każdego atrybutu w klasie,
* *Setter -* generowana jest funkcja ustawiania wartości dla każdego atrybutu w klasie,
* *AllArgsConstructor* – tworzy konstruktor ze wszystkimi atrybutami w klasie,
* *NoArgsConstructor* – tworzy domyślny konstruktor klasy
* *Builder* – pomocny przy tworzeniu nowych obiektów danej klasy przy użyciu funkcji *builder()* i *build()*.

Ostatnia adnotacja zostanie opisana w punkcie 5.6.

@Entity  
@Table(name = **"exercise"**)  
@Getter  
@Setter  
@AllArgsConstructor  
@NoArgsConstructor  
@Builder  
@TypeDef(  
 name = **"pgsql\_enum"**,  
 typeClass = PostgreSQLEnumType.**class**)  
**public class** Exercise {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private** Integer **id**;  
 **private** String **exerciseName**;  
 **private** String **description**;  
 **private** String **url**;  
 @Enumerated(EnumType.***STRING***)  
 @Column(columnDefinition = **"muscle\_group"**)  
 @Type(type=**"pgsql\_enum"**)  
 **private** MuscleGroup **muscleGroup**;  
 @Enumerated(EnumType.***STRING***)  
 @Column(columnDefinition = **"exercise\_type"**)  
 @Type(type=**"pgsql\_enum"**)  
 **private** ExerciseType **exerciseType**;  
 @Enumerated(EnumType.***STRING***)  
 @Column(columnDefinition = **"difficulty"**)  
 @Type(type=**"pgsql\_enum"**)  
 **private** Difficulty **difficulty**;  
 **private** Boolean **isForOldPeople**;  
 **private** Boolean **isForObesePeople**;  
 **private** Boolean **isForYoungPeople**;  
 **private** Boolean **isForWomen**;  
 @OneToMany(mappedBy = **"exercise"**)  
 **private** List<ExerciseWithData> **exercisesWithData**;  
}

Listing 2: Klasa Exercise

Klasa oznaczona adnotacją *@Entity* musi zawierać pole, które oznaczone jest adnotacją *@Id*, która oznaczać będzie klucz główny. Warto przy takim polu dodać jednocześnie adnotacje *@GeneratedValue*, która określi sposób generowania id. W tym przypadku podana została strategia *IDENTITY* oznaczająca, że każdy nowy obiekt tej klasy zapisany w bazie danych będzie miał id zwiększone o 1, względem poprzedniego obiektu.

Encja *Exercise* jest w relacji *jeden-do-wielu* z encją *ExerciseWithData*. By odwzorować to zachowanie po stronie serwera należało stworzyć atrybut zawierający listę obiektów tej encji. W klasie *ExerciseWithData* zdefiniowano obiekt klasy *Exercise*, który oznaczono adnotacjami:

* *@ManyToOne* – typ kaskady *Detach* oznacza, że w momencie usuwania obiektu *ExerciseWithData* obiekt *Exercise* zostanie „odłączony” i nie zostanie usunięty,
* *@JoinColumn* – oznaczające, że dane pole to klucz obcy.

W celu komunikacji się z bazą danych należało stworzyć interfejs, który *Springboot* nazywa repozytorium. Takie interfejsy oznaczone są adnotacją *@Repository* oraz rozszerzone o *JpaRepository*, której jako typy generyczne przekazujemy klasę mapującą encję w bazie danychoraz typ zmiennej, który w tej klasie oznaczony jest adnotacją *@Id.*

W tym interfejsie można tworzyć funkcje na 3 różne sposoby:

* funkcja używa słów rozpoznawalnych przez *Springboot* takich jak: *find*, *by*, *id*, *and* przez co można zbudować funkcję np. *findExerciseById*, która zgodnie z nazwą znajdzie w bazie danych obiekt *Exercise* o id podanym w parametrze funkcji,
* funkcja z dodaną adnotacją *@Query*, gdzie w adnotacji za pomocą języka *JPQL* bezpośrednio przekazuje się funkcji, jakie dane mają zostać pobrane z bazy danych,
* funkcja z dodaną adnotacją *@Query*, gdzie w adnotacji zostaje ustawiony parametr *nativeQuery*, a następnie za pomocą języka *SQL* określone zostają dane do zwrócenia.

By aplikacja mogła komunikować się z serwerem należało zdefiniować w nim klasę odpowiadającą za API (Listing 4). Adnotacje tej klasy oznaczają:

* *@CrossOrigin* – koncept bezpieczeństwa zabraniający pobierania żądań od źródeł innych niż podane,
* @RestController – oznaczająca, że klasa jest wystawia API,
* *@RequestMapping* – oznaczające url, w które aplikacja webowa powinna uderzać,
* *@RequiredArgsConstruction* – oznacza, że Lombok wygeneruje konstruktor z wymaganymi polami (w tym przypadku pole *adapter*).

Metody zawarte w tej klasie oznaczone są adnotacjami @*GetMapping* oraz *@PostMapping* oznaczająco odpowiednio metodę API *GET* oraz *POST*. W adnotacjach tych podano też dalszy ciąg url.

Przed parametrami funkcji znajdują się adnotacje *@PathVariable* i *@RequestBody*. Oznaczają one, że wartość zmiennej będzie znajdować się odpowiednio w linku url w nawiasach klamrowych lub ciele otrzymanego żądania.

Każdy z pakietów zawiera też interfejs *translator* (Listing 6) używany przez narzędzie *Mapstruct* – jest to oznaczone przez adnotację *@Mapper* znajdującą się nad definicją interfejsu. Teraz należy już tylko zdefiniować funkcję z klasami, które chcemy zmapować.

@Repository  
**public interface** ExerciseRepository

**extends** JpaRepository<Exercise, Integer> {

Optional<Exercise> findExerciseById(**int** id);  
 List<Exercise> findAll(Specification<Exercise> toCriteria);  
 Optional<Exercise> getExerciseById(Integer id);  
 List<Exercise> getExercisesByDifficultyAndExerciseType(  
 Difficulty difficulty, ExerciseType exerciseType  
 );  
 @Query(  
 nativeQuery=**true**,  
 value=

**"SELECT \* FROM exercise e ORDER BY random() LIMIT ?1"** )  
 List<Exercise> fetchRandomExercises(  
 Integer numberOfExercisesToPick  
 );  
 @Query(**"SELECT e FROM Exercise e"** +  
 **" WHERE e.difficulty in ?1 and e.exerciseType = ?2 "** + **" and e.isForWomen = true"**)  
 List<Exercise> getExercisesByDifficultyAndExerciseTypeAndIsFemale(  
 List<Difficulty> difficulty, ExerciseType exerciseType  
 );

Listing 3: Repozytorium Exercise

@CrossOrigin(origins = **"http://localhost:3000"**)  
@RestController  
@RequestMapping(value =**"/api/exercise"**)  
@RequiredArgsConstructor  
**public class** ExerciseController {  
 **private final** ExerciseAdapter **adapter**;  
 @GetMapping(**"/details/{id}"**)  
 **public** ResponseEntity<ExerciseDetails> getExerciseDetails(  
 @PathVariable Integer id  
 ) {  
 ExerciseDetails exerciseDetails = **adapter** .getExerciseDetails(id);  
 **return** ResponseEntity.*ok*(exerciseDetails);  
 }  
 @PostMapping(value = **"/get"**)  
 @ResponseBody  
 **public** ResponseEntity<GetFilteredExerciseResp> getFilteredExercises(  
 @RequestBody GetFilteredExercisesReq request  
 ) {  
 GetFilteredExerciseResp resp = **adapter** .getFilteredExercises(request);  
 **return** ResponseEntity.*ok*(resp);  
 }  
}

Listing 4: Klasa ExerciseController

@Component  
@RequiredArgsConstructor  
**public class** ExerciseMediator **implements** ExerciseAdapter {  
 **private final** ExerciseRepository **repository**;  
 **private final** ExerciseTranslator **translator**;  
 @Override  
 **public** ExerciseDetails getExerciseDetails(Integer id) {  
 Optional<Exercise> maybeExercise = **repository** .getExerciseById(id);  
 **if** (maybeExercise.isEmpty()) {  
 **throw new** ExerciseNotFoundException(id);  
 }  
 **return translator**.toExerciseDetail(maybeExercise.get());  
 }

…

Listing 5: Klasa ExerciseMediator

@Mapper(componentModel = **"spring"**)  
**public interface** ExerciseTranslator {  
 ExerciseTranslator ***INSTANCE*** = Mappers.*getMapper*(  
 ExerciseTranslator.**class** );  
 ExerciseTableData toTableData(Exercise exercise);  
 List<Difficulty> toDifficultyEntityModels(  
 List<pl.edu.pwr.common.enums.Difficulty> difficulties  
 );  
 List<MuscleGroup> toMuscleGroupEntityModels(  
 List<pl.edu.pwr.common.enums.MuscleGroup>   
 muscleGroups  
 );  
 List<ExerciseType> toExerciseTypeEntityModels(  
 List<pl.edu.pwr.common.enums.ExerciseType>  
 exerciseTypes  
 );  
 ExerciseDetails toExerciseDetail(Exercise exercise);  
}

Listing 6: Translator klasy Exercise

Pozostała jeszcze jedna główna klasa pakietu e*xercise - Mediator* (Listing 5) implementująca interfejs *Adapter*. Odpowiada ona za logikę aplikacji, walidowanie danych i zwracanie ich do kontrolera.

W przypadku bardziej skomplikowanych operacji, tworzone zostawały dodatkowe klasy. Taka sytuacja miała miejsce przy generowaniu planu dla użytkownika w pakiecie *trainingPlan*, gdzie funkcje generowania planu były za duże do przechowania w jednej klasie.

Generowanie planu treningowego rozpoczyna się od sprawdzenia, czy obiekt wysłany w żądaniu nie jest pusty, a użytkownik, dla którego zostanie stworzony plan istnieje w bazie danych. Jeśli te dane są poprawne, następuje przypisanie atrybutów żądania do zmiennych zdefiniowanych w ciele funkcji.

Kolejnym etapem generowania planu jest określenie, jaki stopień zaawansowania określił użytkownik. Wykonuje się to w operacji *switch(),* która jako parametr przyjmuje wartość *difficulty*. Omówiona zostanie ścieżka generowania łatwego planu treningowego.

W przypadku *EASY* obliczana zostaje ilość wszystkich ćwiczeń do wygenerowania. Wartość ta wynika z pomnożenia liczby dni treningowych w jednym planie treningowym, liczby planów treningowych do wygenerowania oraz liczby ćwiczeń w jednym dniu treningowym. Następnie wywoływana jest funkcja z klasy *Service generate().*

W tej funkcji za pomocą funkcji *getExercisesToGenerateTrainingPlan()* oraz *generate()* następuje generowanie całego planu treningowego na kolejne tygodnie.

Funkcja dobierająca ćwiczenia do generowania planu jest najważniejszym elementem tej operacji. Bierze ona jako parametry: liczbę ćwiczeń do wygenerowania, typ tych ćwiczeń, wartość prawda/fałsz dotyczącą tego, czy użytkownik jest otyły, w podeszłym wieku lub młody, a także jego płeć (wartość płeć przyjmuje wartość prawda/fałsz w zależności tego, czy użytkownik jest kobietą).

Następnie ustawiana jest flaga, która zostanie opisana w punkcie 5.6 oraz wywoływana jest funkcja *getExercisesBasedOnFlag()*, która pobiera z bazy danych dokładnie te ćwiczenia, które w 100% spełniają kryteria użytkownika.

Funkcja *getExercisesToGenerateTrainingPlan()* kończy się przez wywołanie metody *getExercisesToCreatePlan()*.Metoda ta filtruje pobrane do tej pory ćwiczenia i wyrzuca z nich te, które nie będą trenowane dla tego poziomu trudności. Następnie, jeśli nie pobrano wystarczająco ćwiczeń z bazy danych, funkcja wchodzi do pętli *while()*, która wykonuje się tak długo, aż liczba ćwiczeń pobranych nie równa się liczbie ćwiczeń wymaganych lub liczba iteracji pętli nie przekroczy 50.

W środku pętli pobierane zostają losowe ćwiczenia. Rozmiar pobranej listy zależy od ilości brakujących ćwiczeń. Następnie przechodząc przez wszystkie pobrane ćwiczenia sprawdzamy, czy spełniają one warunki:

* ćwiczenie nie może być trudne oraz
  + typ ćwiczenia musi się zgadzać z typem planu treningowego lub
  + musi zgadzać się warunek zależny od flagi lub
  + zmienna losowa będzie większa od zadanej wartości.

Gdy ćwiczenie spełni warunek następują ostatnie dwa sprawdzenia. Pierwsze z nich sprawdza, czy rozmiar listy ćwiczeń do wygenerowania planu nie jest większy od liczby ćwiczeń, które należało pobrać. Natomiast drugie sprawdza, czy lista ćwiczeń nie zawiera już sprawdzanego ćwiczenia – jeśli tak, to pomija dodawanie go.

Program przechodząc przez te wszystkie kroki, w wyjątkowych sytuacjach, nadal może nie mieć wystarczającej liczby ćwiczeń do wygenerowania planu, dlatego wywoływana jest ostatnia funkcja *getMissingExercisesIfNecessaary()*. Jej zadaniem jest sprawdzenie, czy liczba ćwiczeń jest wystarczająca. Jeżeli odpowiedź jest przecząca, następuje pobranie brakującej ilości ćwiczeń. Nie jest wówczas przeprowadzana żadna walidacja ćwiczeń.

Po pobraniu wszystkich ćwiczeń algorytm przechodzi do tworzenia planu treningowego funkcją *generateTrainingPlan()*. Następuje iteracja przez liczbę planów treningowych do wygenerowania. W każdej iteracji zachodzi:

* zapisanie nowego planu treningowego w bazie danych w celu wygenerowania id,
* wywołanie funkcji *generateTrainingDays()*,
* przypisanie użytkownika i dni treningowych do planu.

Logika funkcji *generateTrainingDays()* jest podobna do funkcji tworzenia planu treningowego. Funkcja oblicza liczbę ćwiczeń dla jednego dnia, a następnie w pętli tworzy nowe dni treningowe i ustawia ich dane, w tym ćwiczenia z danymi przy użyciu funkcji *generateExercisesWithData()*. Pętla kończy się, gdy liczba wygenerowanych dni treningowych będzie odpowiadała ich wymaganej liczbie.

Funkcja *generateExercisesWithData()* iteruje od 0 do liczby dni na jeden dzień treningowy, tworząc listę ćwiczeń z danymi poprzez ustawianie liczby powtórzeń, liczby serii, losowo wybranych ćwiczeń z wcześniej stworzonej listy ćwiczeń (która będzie zmniejszana z każdą iteracją pętli), a następnie zwraca tę listę, by przypisać ją do dnia treningowego.

## Implementacja interfejsu użytkownika

Podział folderów w projekcie strony internetowej został opisany w punkcie 5.2. Na przykładzie *main-page* opisana zostanie implementacja logiki tworzącej aplikację internetową.

Głównym plikiem, w którym znajduje się szkielet strony oraz logika obsługująca zachowanie strony głównej jest *MainPageContainer.js.* Biblioteka *React* jest dość specyficzna. Strona zbudowana jest z funkcji, które zwracają kod HTML w pliku o rozszerzeniu *Javascript*. Takie jest przynajmniej pierwsze wrażenie, gdy ktoś niezaznajomiony spojrzy na kod aplikacji. Tak naprawdę, funkcje te zwracają kod JSX, które jest rozszerzeniem składni *Javascript*, wprowadzonym przez *Reacta*.

Główną funkcją w pliku jest *MainPageContainer()*, której pierwszym zadaniem jest zweryfikowanie, czy użytkownik jest zalogowany. Używa do tego *hooka[[3]](#footnote-3)* *useEffect()*, który uruchamia funkcję lambda zawsze podczas pierwszej inicjalizacji oraz w zależności od tego, jakie parametry podaliśmy jako drugi parametr. Pusta lista oznacza, że *hook* uruchomi się tylko raz.

Wywołana zostaje funkcja wyeksportowana z pliku *CheckIfUserIsLoggedIn.js.* Parametry tej funkcji to:

* drugi parametr *hooka* *useState()*, gdzie *hook* ten możemy porównać do zdefiniowania zmiennej *x* oraz funkcji ustawiającej wartość tej zmiennej w jednej linijce kodu,
* zmienna *navigate* – będąca *hookiem* *useNavigate()*, służącego do nawigowania się po różnych stronach za pomocą kodu.

Metoda *isUserLoggedIn()* sprawdza, czy w pamięci przeglądarki zapisany jest obiekt *user*. Jeśli nie ma takiego obiektu to użytkownik zostaje przekierowany na stronę logowania. Jeśli obiekt taki istnieje i jego parametr *isUserLoggedIn* ma wartość *true* to na taką wartość ustawiona zostaje też zmienna *userId*.

export function isUserLoggedIn(setUserId, navigator) {

    let userFromMem = JSON.parse(window.localStorage.getItem('user'));

    if(!userFromMem || (userFromMem && (!userFromMem.userId || !userFromMem.isUserLoggedIn))) {

        navigateToLoginPage(navigator);

    }

    if(userFromMem &&  userFromMem.isUserLoggedIn) {

        setUserId(userFromMem.userId);

    }

}

Listing 7: Funkcja sprawdzająca, czy użytkownik jest zalogowany

Przy pierwszej inicjalizacji widoku strony głównej uruchamiane zostają jeszcze dwa dodatkowe *hooki* *useEffect()* (Listing 8). Ich zadaniem jest pobieranie listy planów treningowych do wyświetlenia w sekcji *Twoje plany treningowe*.

useEffect(() => {

        if (userId) {

            fetchTrainingPlans(

                userId, setIsLoading, handleSetTrainingPlans, showErrorMsg

            )

            setFetchAfterGenerated(false);

        }

    }, [userId, fetchAfterGenerated]);

useEffect(() => {

        if(userId && fetchAfterDelete) {

            fetchTrainingPlans(

                userId, setIsLoading, handleSetTrainingPlans, showErrorMsg

            );

        }

    }, [fetchAfterDelete]);

Listing 8: Hooki pobierające plany treningowe

Uruchamiane są w momencie pierwszej inicjalizacji oraz za każdym razem, gdy któraś ze zmiennych (*userId, fetchAfterGenerated, fetchAfterDelete*) zostanie zmieniona. By zapobiec próbie pobrania planów, nawet gdy użytkownik nie jest zalogowany, w środku funkcji anonimowej znajduje się warunek sprawdzający, czy użytkownik jest zalogowany i tylko w tym przypadku pobierane są plany treningowe. Pobieranie planów jest zadaniem metody *fetchTrainingPlans()* (Listing 9)

Funkcja ta definiuje na początku kontroler mający na celu zaprzestanie wysyłania i odbierania jakichkolwiek danych w momencie, gdy użytkownik przejdzie na inną stronę.

Metoda *fetch()* jako parametry otrzymuje url oraz obiekt ustawiający specyfikacje żądania:

* *method* – oznacza jaki typ zapytania REST będzie wysyłany,
* *headers* – *content-type* wskazuje jaki typ danych będzie zwracany ze źródła,
* *signal* – jako wartość otrzymuje sygnał kontrolera zdefiniowanego na początku funkcji.

W tym momencie kontroler serwera wystawiający api na podanym url zaczyna swoją pracę i zwraca odpowiednie plany treningowe.

const fetchTrainingPlans = (id, setIsLoading, handleSetTrainingPlans, showErrorMsg) => {

    const controller = new AbortController();

    setIsLoading(true);

    fetch("http://localhost:8080/api/trainingPlan/" + id, {

        method: 'GET',

        headers: {

            'Content-Type': 'application/json'

        },

        signal: controller.signal,

        })

        .then(resp => {

            if (resp.ok) {

                return resp.json();

            }

            throw resp;

        })

        .then(data => {

            if(data){

                handleSetTrainingPlans(data.trainingPlanTableData, showErrorMsg);

            }

        })

        .catch(error => {

            showErrorMsg();

            throw error;

        })

        .finally(() => {

            setIsLoading(false)

        });

        return () => {

            controller.abort();

        }

}

Listing 9: Ciało funkcji pobierającej plany treningowe

Gdy serwer zwróci dane wykonuje się pierwsza metoda *then()*, która sprawdza, czy odpowiedź jest prawidłowa i jeśli tak, wykonuje się druga metoda o tej samej nazwie, w której możemy operować na otrzymanym z serwera obiekcie. W tym przypadku wykonuje się funkcja *handleSetTrainingPlans()*, która mapuje otrzymany obiekt i zapisuje te obiekty w zmiennej *userTrainingPlans*. Po wykonaniu tej operacji wykonuje się funkcja *finally()*.

Można zauważyć, że od funkcji *fetch()* odchodzi jeszcze jedna metoda *catch*(), której zadaniem jest obsługa wyjątków zwróconych z serwera.

W międzyczasie funkcja *MainPageContainer()* zwraca kod JSX, w którym znajdują się odwołania *UserTrainingPlans* oraz *UserDataToGeneratePlan*. Odnoszą się one do funkcji znajdujących się w plikach *UserTrainingPlans.js* oraz *UserDataToGeneratePlan.js*.

Do funkcji *UserTrainingPlans()* jako parametry przekazywane są plany treningowe, flaga mówiąca o tym, czy w danym momencie są pobierane dane planów treningowych oraz funkcja usuwająca plan z listy. Metoda ta służy do wyświetlania tabeli z danymi dotyczącymi planów. Obsługuje też usuwanie planu treningowego z bazy danych. Wysyła żądanie podobne do tego z Listing 9, różnicą jest to, że metoda ustawiona jest teraz na *DELETE* oraz uderza w inny *endpoint*. Po otrzymaniu prawidłowej odpowiedzi z serwera wywoływana jest z funkcji *UserTrainingPlans()* metoda *deletePlanById()* znajdująca się w *MainPageContainer*.

## Problemy implementacyjne

Naturalnym jest napotykanie problemów podczas implementacji aplikacji lub jakiegoś rozwiązania. Nie inaczej było w przypadku tej strony internetowej. Napotkano wiele mniejszych i kilka większych problemów:

* wysyłanie obiektów typu enum między *backendem* a bazą danych,
* błąd *cross-origin resource sharing* przy wysyłaniu żądań z *frontendu* do *backendu*,
* prosta implementacja pobierania ćwiczeń do generowania planu treningowego, które spełniać będą wszystkie kryteria.

### Błąd CORS przy wysyłaniu żądań

W momencie, gdy wysłane zostało żądanie z aplikacji internetowej, zwracany był błąd, który sprawiał, że nie można było współdzielić zasobów z serwerem. Rozwiązaniem było dodanie nowej klasy *WebConfig* z dwoma adnotacjami: *@Configuration* oraz *@EnableWebMvc*. Klasa ta dodatkowo została rozszerzona o klasę abstrakcyjną *WebMvcConfigurerAdapter*. Dzięki temu można było nadpisać funkcję *addCorsMappings()*, w której do rejestru dodano źródło <http://localhost:3000> - port zajęty przez aplikację.

Dodatkowo do każdego kontrolera serwera dodano adnotację *@CrossOrigin* przedstawioną w punkcie 5.4.

### Implementacja pobierania ćwiczeń do generowania planu treningowego

Encja ćwiczeń w bazie danych ma pola trudność, typ oraz 4 pola definiujące, czy osoba ćwicząca jest otyła, w podeszłym wieku, młoda lub czy jest kobietą. Istnieje 12 kombinacji parametrów osoby. By uniknąć pisania funkcji zawierającej w sobie 12 skomplikowanych wyrażeń *if*-*else* zadeklarowano zmienną *flag*. Jej wartość zwracana jest przez funkcję *getFlagValue()* zdefiniowaną w klasie *TrainingPlanUtils*.

Rozwiązanie to miało być szybką „implementacją” 4-bitowej flagi, gdzie wartość „bitu” jest ustawiana, jeśli odpowiadający mu atrybut ćwiczenia jest równy *true*.

Ustawienie flagi umożliwia prosty sposób znalezienia odpowiedniej funkcji z *ExerciseRepository*. Funkcja *getExercisesBasedOnFlag()* zbudowana jest z wyrażenia *switch*, które w zależności od wartości flagi wybiera prawidłową metodę do wywołania, która zwróci listę ćwiczeń spełniających odpowiednie kryteria.

### Komunikacja z użyciem obiektów enum

Tworzenie typów *enum* w bazie danych oraz po stronie *backendu* stworzył duży problem z mapowaniem obiektów podczas komunikacji *serwer-baza*\_*danych*.

Rozwiązanie tego problemu przedstawił użytkownik Vlad Mihalcea [14], który jest autorem biblioteki *hybernate-types* [15].

W aplikacji na początku zaimplementowane były cztery klasy *enum* odpowiadające tym z bazy danych. Należało jednak stworzyć kolejne cztery klasy *enum* rozszerzone o konstruktor i dwie metody:

* *toString()* – z adnotacją *@JsonValue*,
* *fromValue()* – z adnotacją *@JsonCreator*.

Następnie należało dodać klasę *PostgreSQLEnumType*, która rozszerzała klasę *org.hibernate.type.EnumType*. Znajduje się w niej jedna funkcja *nullSafeSet()*. Teraz, gdy klasa odpowiadająca encji bazy danych używa zmiennych o typie *enum*,należy dodać nad taką klasą adnotację *@TypeDef*, w której ustawiamy pola *name* na *psql\_enum* oraz *typeClass* na klasę *PostgreSQLEnumType.class*. Pola *enum* takiej klasy powinny być dodatkowo oznaczone adnotacją *@Type,* gdzie wartość *type* jest równa wartości podanej w polu *name*.

Po wykonaniu tych operacji komunikacja i mapowanie *enumów* znajdujących się na serwerze i bazie danych powinna przebiegać pomyślnie.

# Testowanie aplikacji

Nieważne jak dobrze napisany jest kod, jeśli nie jest przetestowany, to nie jest on wiarygodny. Aplikacja została przetestowana na dwa sposoby. Pierwszy z nich to testy manualne po stronie *frontendu*, a drugie to testy *backendu* przy użyciu narzędzia *JUnit*

## Przeprowadzone testy

Testy *backendu* obejmują działanie klas *Mediator* obsługujących główną logikę aplikacji. Sprawdzały one poprawność działania aplikacji podczas przekazywania do niej danych prawidłowych oraz danych, które powodowały wyrzucanie wyjątku. Główna logika tych testów była podzielona na 3 części:

* *given* – w której deklarowane są zmienne testowe oraz określane jest zachowanie metod zamockowanych klas lub interfejsów,
* *when* – w tej części testu wywoływana jest metoda do testowania,
* *then* – w której znajdują się sprawdzenia, czy zwrócone dane mają wartości zgodne z oczekiwaniami.

Przykładem testu może być logowanie się użytkownika po stronie serwera. W sekcji *given* definiowane są zmienne, a następnie przy użyciu funkcji *when()* pochodzącej z narzędzia *Mockito* definiowane jest zachowanie funkcji *findGymUserByLogin()*, która implementowana jest przez repozytorium klasy *GymUser.*

@DisplayName(**"Logged In - Correct Data"**)  
@Test  
**public void** LogInCorrectly() {  
 *// given* Integer id = 1;  
 String login = **"login"**;  
 String password = **"password"**;  
 GymUser user = **new** GymUser();  
 user.setId(id);  
 user.setLogin(login);  
 user.setPassword(password);  
 Optional<GymUser> maybeUser = Optional.*of*(user);  
 *when*(**repository**.findGymUserByLogin(login)).thenReturn(maybeUser);  
 LoginUserReq req = **new** LoginUserReq();  
 req.setLogin(login);  
 req.setPassword(password);  
 *// when* LoginUserResp resp = **mediator**.loginUser(req);  
 *// then  
 assertEquals*(id, resp.getUserId());  
 *assertTrue*(resp.getIsUserLoggedIn());

}

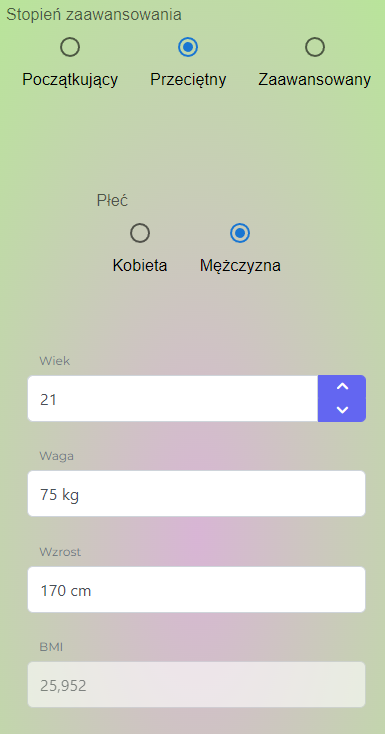
Listing 10: Test Logowania się użytkownika

W przypadku tego testu zwracany obiekt będzie zawierał dane, które należy sprawdzić przy pomocy funkcji *assertEquals()* oraz *assertTrue()*, które w przypadku niezgodności oznaczą test jako wykonany z niepowodzeniem.

Strona internetowa była przetestowana w sposób manualny. Na każdej podstronie i sekcji wprowadzane były dane, które mogły spowodować popsucie działania aplikacji. Testy wypadły dobrze, aplikacja działała prawidłowo. Funkcjonalności zostały zaimplementowane poprawnie, a wiele pól jest objęte walidacją danych, co ogranicza ryzyko błędów. Komunikaty wyświetlają się użytkownikowi podczas korzystania z aplikacji. Podczas poprawnie wykonanej operacji lub operacji przerwanej przez błąd, użytkownik otrzyma odpowiedni komunikat.

## Wyniki testów manualnych

Najważniejszą funkcjonalnością aplikacji jest generowanie planów. W systemie istnieją trzy rodzaje ćwiczeń – siłowe, kardio oraz mobilności. Dla treningu siłowego generowane zostają cztery tygodnie treningowe, natomiast dla pozostałych - jeden. Przeprowadzono test dla przeciętnego użytkownika wybierającego trening siłowy z czterema dniami treningowymi w tygodniu. Sekcja *Wygeneruj swój plan* została przedstawiona na Rysunek 15.



Rysunek 15: Sekcja "Wygeneruj swój plan"

Po wciśnięciu przycisku *Wygeneruj plan!* należy oczekiwać pojawienia się w sekcji *Twoje plany treningowe* czterech planów treningowych, gdzie każdy z tych planów będzie zawierał cztery dni treningowe z ćwiczeniami, których dominująca część jest przeznaczona do poprawy siły.

Wynikiem operacji są plany treningowe przedstawione na Rysunek 16. Dominującym typem każdego z planów zgodnie z oczekiwaniami jest typ *Siła*. Daty dni treningowych nie nachodzą na siebie – żaden z nich nie trwa dłużej niż 7 dni, a to oznacza, że wygenerowane plany spełniają wymagania funkcjonalne podane w punkcie 2.3.

Przerwa pomiędzy czterema dniami treningowymi w planie wynosi jeden dzień. Dzień treningowy składa się z 6 ćwiczeń z zaplanowaną liczbą powtórzeń i serii dla każdego z nich.

Obraz zawierający stół

Opis wygenerowany automatycznie

Rysunek 16: Wygenerowane plany treningowe.

Ręczne tworzenie planów treningowych jest drugą najważniejszą funkcjonalnością aplikacji. Użytkownik samodzielnie wybierając ćwiczenia jest w stanie tworzyć dni treningowe. Z dni składać się będzie plan treningowy. Podczas wybierania daty dnia treningowego ważna jest jej unikalność. W przypadku powtórzenia dat użytkownik otrzyma komunikat informujący go o tym, że dni w planie muszą się różnić.

Wynik tworzenia planu możemy zobaczyć na stronie głównej aplikacji. Oprócz planów, które są aktualnie przypisane do użytkownika, pojawi się tam plan, który użytkownik ręcznie stworzył.

# Podsumowanie

Implementacja aplikacji internetowej pozwoliła na poznanie nowych technologii oraz rozwinięcie umiejętności pisania dobrego, czytelnego kodu oraz jego testowania. Doświadczenie zdobyte podczas studiów pozwoliło na zaprojektowanie prostej, ale wystarczającej do potrzeb aplikacji, bazy danych, której tabele i relacje można w przyszłości użyć w innych projektach o podobnej tematyce.

## Wnioski

Celem pracy było stworzenie aplikacji internetowej, która generuje plany treningowe dla użytkownika w zależności od jego preferencji. Efekt końcowy pracy pozwala na nieco więcej niż to. Aplikacja pozwala na generowanie planu, ale i ręczne stworzenie planu poprzez samodzielne wybieranie ćwiczeń. Plany można przeglądać, edytować i usuwać. Ćwiczenia znajdujące się w bazie danych są zrozumiale opisane. Zawierają film instruktażowy pokazujący jak prawidłowo wykonywać ćwiczenie, którym można posłużyć się do jego nauki. Gdy osoba korzystająca z aplikacji nie jest zalogowana, nie zostanie wyświetlona zawartość strony. Użytkownik może logować się i rejestrować w aplikacji. Dane użytkownika tj. waga, wzrost i wiek są aktualizowane za każdym razem, gdy ten wygeneruje swój plan treningowy, Użytkownik ma możliwość zmiany swojego loginu,   
e-mailu i hasła. Po stronie serwera i strony internetowej przed wykonaniem jakiejkolwiek operacji sprawdzana jest najpierw spójność danych. Na *backendzie* zdefiniowano klasy rozszerzające klasę *RuntimeException* w celu wyświetlenia dokładniejszych informacji dotyczących błędów pojawiających się w trakcie działania aplikacji. Baza danych postawiona jest za pomocą narzędzia *Docker*, które jest jednym z najpopularniejszych narzędzi do konteneryzacji aplikacji w małych, lekkich maszynach wirtualnych. Narzędzie *Flyway* pozwala na uzupełnianie bazy danymi za każdym razem, gdy uruchomimy serwer.

## Koncepcja rozwoju aplikacji

Aplikacja ma wiele ciekawych sposobów, w jakie można ją rozwinąć. Istnieje bowiem wiele funkcjonalności, które można dodać do implementacji. Mogą to być:

* wprowadzenie bezpieczeństwa przy pomocy narzędzi takich jak *Keycloak* lub *SpringSecurity*,
* podział roli użytkowników,
* nowe role mogą skutkować powstaniem użytkownika, który jest trenerem personalnym,
* implementowanie czatu między użytkownikami tj. trener personalny i osoba, która zaczyna ćwiczyć na siłowni,
* dodanie możliwości dodawania, edycji i usuwania ćwiczeń z poziomu aplikacji dla roli *Admin*,
* rozszerzenie aplikacji o układanie diety, która jest ważną częścią treningów,
* dodanie funkcjonalności oceniania wygenerowanych planów i ćwiczeń,
* implementacja generowania planu treningowego używając nauczania maszynowego,
* poprawienie ogólnego wyglądu aplikacji.

Najciekawszym z punktów wydaje się być generowanie planu używając nauczania maszynowego. Wadą tego rozwiązania byłby fakt, że duża liczba użytkowników musiałaby generować i oceniać swoje plany, by model zaczął generować je poprawniej i były dopasowane do użytkownika.

Jeśli aplikacja miałaby kiedykolwiek zostać wdrożona i przekazana do publicznego użytku, wymaganym byłoby zaimplementowanie systemów bezpieczeństwa tj. zabezpieczenie poufnych danych użytkowników i zablokowanie dostępu do tych podstron, do których użytkownik o danej roli nie ma dostępu.

# Bibliografia

1. NAWROCKI, J., & OLEK, Ł. OPISYWANIE PROCESÓW BIZNESOWYCH Z WYKORZYSTANIEM PRZYPADKÓW UŻYCIA.
2. ZOFIA KRUCZKIEWICZ, SPECYFIKACJA WYMAGAŃ FUNKCJONALNYCH ZA POMOCĄ DIAGRAMU PRZYPADKÓW UŻYCIA 2016
3. JOY, B., STEELE, G., GOSLING, J., & BRACHA, G. (2000). THE JAVA LANGUAGE SPECIFICATION.
4. FLANAGAN, D. (2020). JAVA-SCRIPT: THE DEFINITIVE GUIDE.
5. BANKS, A., & PORCELLO, E. (2017). LEARNING REACT: FUNCTIONAL WEB DEVELOPMENT WITH REACT AND REDUX. " O'REILLY MEDIA, INC."
6. WALLS, C. (2015). SPRING BOOT IN ACTION. SIMON AND SCHUSTER.
7. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP, POSTGRESQL 15.1 DOCUMENTATION
8. RED GATE SOFTWARE LTD, FLYWAY DOCUMENTATION 2022
9. ANDERSON, C. (2015). DOCKER [SOFTWARE ENGINEERING]. IEEE SOFTWARE, 32(3), 102-C3.
10. GULATI, S., SHARMA, R., (2017) JAVA UNIT TESTING WITH JUNIT 5: TEST DRIVEN DEVELOPMENT WITH JUNIT 5
11. KACZANOWSKI, T. (2012). PRACTICAL UNIT TESTING WITH TESTNG AND MOCKITO. TOMASZ KACZANOWSKI.
12. https://www.samouczekprogramisty.pl/testy-jednostkowe-z-uzyciem-mock-i-stub/ (data dostępu: 03.12.2022 r.)
13. https://pl.reactjs.org/docs/hooks-overview.html (data dostępu: 04.12.2022 r.)
14. https://stackoverflow.com/questions/27804069/hibernate-mapping-between-postgresql-enum-and-java-enum (data dostępu: 04.12.2022 r.)
15. https://github.com/vladmihalcea/hibernate-types (data dostępu: 04.12.2022 r.)
16. SZCZYGIELSKA, A. (2020). AKTYWNOŚĆ FIZYCZNA POLAKÓW W CZASIE PRACY ZDALNEJ-PREZENTACJA WYNIKÓW BADAŃ EMPIRYCZNYCH. BEZPIECZEŃSTWO PRACY: NAUKA I PRAKTYKA, 22-26.
17. https://studiosoftware.pl/blog/planowanie-projektu-informatycznego-6-krokow-do-sukcesu/ (data dostępu: 06.12.2022 r.)
18. https://impicode.pl/blog/jakie-czynniki-wplywaja-na-wybor-technologii-do-projektu-it/ (data dostępu: 06.12.2022 r.)
19. https://pl.wikipedia.org/wiki/Plain\_Old\_Java\_Object   
    (data dostępu: 07.12.2022 r.)

# Spis rysunków

[Rysunek 1: Popularność siłowni w Polsce na przestrzeni lat 2017-2022 5](#_Toc121343141)

[Rysunek 2. Strona główna fabrykasily.pl 7](#_Toc121343142)

[Rysunek 3: Koszty pakietów treningowych na stronie budujmase.pl 7](#_Toc121343143)

[Rysunek 4: Diagram związków encji 10](#_Toc121343144)

[Rysunek 5: Diagram przypadków użycia 11](#_Toc121343145)

[Rysunek 6: Ekran menu głównego 12](#_Toc121343146)

[Rysunek 7: Ekran listy dni treningowych 13](#_Toc121343147)

[Rysunek 8: Ekran edycji dnia treningowego 14](#_Toc121343148)

[Rysunek 9: Ekran listy ćwiczeń z danego dnia treningowego 14](#_Toc121343149)

[Rysunek 10: Ekran tworzenia planu treningowego 15](#_Toc121343150)

[Rysunek 11: Ekran listy wszystkich ćwiczeń 16](#_Toc121343151)

[Rysunek 12: Ekran szczegółów ćwiczenia 17](#_Toc121343152)

[Rysunek 17: Porównanie zainteresowania Angularem i ReactemJS 19](#_Toc121343153)

[Rysunek 14: Struktura pakietów na przykładzie TrainingPlan 25](#_Toc121343154)

[Rysunek 15: Sekcja "Wygeneruj swój plan" 39](#_Toc121343155)

[Rysunek 16: Wygenerowane plany treningowe. 40](#_Toc121343156)

# Spis listingów

[Listing 1: Skrypt do generowania tabeli Exercise 27](#_Toc121342989)

[Listing 2: Klasa Exercise 28](#_Toc121342990)

[Listing 3: Repozytorium Exercise 30](#_Toc121342991)

[Listing 4: Klasa ExerciseController 30](#_Toc121342992)

[Listing 5: Klasa ExerciseMediator 31](#_Toc121342993)

[Listing 6: Translator klasy Exercise 31](#_Toc121342994)

[Listing 7: Funkcja sprawdzająca, czy użytkownik jest zalogowany 34](#_Toc121342995)

[Listing 8: Hooki pobierające plany treningowe 34](#_Toc121342996)

[Listing 9: Ciało funkcji pobierającej plany treningowe 35](#_Toc121342997)

[Listing 10: Test Logowania się użytkownika 38](#_Toc121342998)

# Dodatki

Załącznik 1: Scenariusze przypadków użycia.

1. POJO – „termin używany do określenia obiektów będącymi zwyczajnymi obiektami Java, nie zaś obiektami specjalnymi.” [19] [↑](#footnote-ref-1)
2. *dto* – (z ang. „Data Transfer Object”) obiekt transferu danych [↑](#footnote-ref-2)
3. „Hooki są to funkcje, które pozwalają „zahaczyć się” w mechanizmy stanu i cyklu życia Reacta, z wewnątrz komponentów funkcyjnych.” [13] [↑](#footnote-ref-3)