UNIWERSYTET GDAŃSKI

WYDZIAŁ MATEMATYKI, FIZYKI I INFORMATYKI

# Piotr Jadczak

*Kierunek: Informatyka – profil ogólnoakademicki*

*Specjalność:*

*Numer albumu: 266389*

# Tytuł pracy

**Architektura współczesnych aplikacji webowych na przykładzie aplikacji do finansów osobistych.**

Praca licencjacka napisana

pod kierunkiem dr Hanny Furmańczyk

(nazwisko promotora)

# Gdańsk 2022

Streszczenie pracy

Zakres pracy obejmuję projekt i implementację aplikacji webowej PersonalFinanceApp, której zadaniem jest pomoc w zarządzaniu finansami osobistymi. Program oferuje takie funkcjonalności jak: rejestracje i logowanie użytkowników, budowanie portfela oszczędnościowego, inwestycyjnego na podstawie wybranych akcji z polskiej Giełdy Papierów Wartościowych oraz najpopularniejszych walut obcych, dostęp do ich aktualnego kursu w złotówkach, archiwizacje i kategoryzacje wydatków, definiowanie celu oszczędnościowego oraz śledzenie postępu jego realizacji na podstawie posiadanych oszczędności. Przewodnim celem pracy, jest zaprezentowanie, na podstawie tej implementacji, praktycznego zastosowania architektury klient-serwer. Zastosowanie tego wzorca skutkuje podziałem rozwiązania na dwa niezależne od siebie programy: stronę kliencką oraz stronę serwerową. Rolą strony serwerowej jest implementacja wszystkich wcześniej wymienionych funkcjonalności aplikacji. Obejmuje to między innymi: dostarczenie logiki biznesowej aplikacji) sposobu przechowywania i udostępniania danych programu i jej użytkowników oraz wystawienie interfejsu API, który umożliwi komunikację z serwerem. Zadaniem strony klienckiej jest dostarczenie użytkownikowi graficznego interfejsu, z pomocą którego będzie mógł w wygodny dla siebie sposób korzystać z funkcjonalności oferowanych przez serwer aplikacji.

Implementacja aplikacji klienckiej została oparta na React.js, bibliotece języka JavaScript. Opiera się ona na podziale programu na komponenty, czyli funkcję języka JavaScript XML zwracające fragmenty interfejsu takie jak menu, formularze, sekcje strony, z których zbudowana jest strona internetowa aplikacji. Dodatkowo klient aplikacji używa języka CSS3 do stylizacji interfejsu użytkownika.

Serwer aplikacji został napisany w frameworku Spring Boot. Do przechowywania danych programu używa bazy danych SQL HSQLDB. Logika biznesowa aplikacji została napisana w języku Java 11. Program korzysta również z wielu bibliotek oferowanych przez Spring, z których najważniejsze to: Spring Data JPA – umożliwia mapowanie obiektowo-relacyjne, Spring Security – restrykcja dostępu do zasobów aplikacji oraz autentykacja użytkowników, Spring Web – dostarczenie REST API. Do pobierania i zarządzania bibliotekami, budowania projektu i kompresji do pliku JAR zostało użyte narzędzie Gradle.

Spis treści

1. Opis problemu

1.1 Porównanie dostępnych rozwiązań

1.2 Możliwość zastosowania praktycznego

1.3 Aspekty badawcze pracy

2. Przypadki użycia aplikacji

2.1 Wymagania niefunkcjonalne i funkcjonalne

2.2 Dokumentacja techniczna aplikacji

2.3 Interfejs użytkownika

3. Implementacja

3.1 Architektura rozwiązania

3.2 Wzorce projektowe

3.3 Web scraping

3.4 RESTful API

3.5 Autentykacja użytkowników

4. Testowanie manualne

4.1 Testowanie strony serwerowej

4.2 Testowanie strony klienckiej

5. Bibliografia

Wstęp

Współczesne aplikacje webowe powinny odpowiadać potrzebom użytkowników, którzy łączą się z nimi za pomocą szerokiej gammy urządzeń jak: komputery stacjonarne, laptopy, smartfony czy nawet telewizory. Twórcy aplikacji, mając to na względzie, muszą przygotować odpowiednie interfejsy dla każdego typu urządzenia uwzględniając jego charakterystykę. Dodatkowo, serwisy zazwyczaj nie składają się z jednego programu, a łączą w sobie wiele oddzielnych aplikacji komunikujących się ze sobą. Połączenie tych dwóch problemów wraz ze stale zmieniającymi się wymaganiami dotyczącymi funkcjonalności aplikacji powoduje, że decyzja o oparciu aplikacji na jednym programie, odpowiadającym jednocześnie za interfejs użytkownika i logikę biznesową serwisu, staje się niepraktycznym rozwiązaniem. Z pomocą przychodzi tutaj wzorzec klient-serwer, za pomocą którego aplikacja może realizować te same funkcjonalności niezależnie, z jakiego typu urządzenia łączy się z nią użytkownik. Wzorzec ten upraszcza implementację wymagań technicznych poprzez uniezależnienie od siebie warstwy prezentacyjnej od logicznej aplikacji. Ułatwia również podział aplikacji serwerowej na programy zajmujące się określoną grupą funkcjonalności oraz korzystanie z zewnętrznych dostawców serwerów poprzez niezależny od nich interfejs kliencki. Nic więc dziwnego, że spowodowało to przejście dostawców aplikacji na tą architekturę i popularyzacje frameworków jeżyka JavaScript jako narzędzi tworzenia wydajnych interfejsów użytkownika.

1. Opis problemu

W ostatnich latach (2020 – 2022) na fali paniki na rynkach finansowych, gwałtownych wahań giełd na całym świecie związanej z pandemią COVID-19, poluzowaniem polityki pieniężnej i wieloma innymi wydarzeniami, temat akcji, papierów wartościowych, kryptowalut przebił się szerzej do opinii publicznej. Chęć wzbogacenia się przyciągnęła miliony nowych osób na giełdy i rynki finansowe, a jako że współczesny handel papierami wartościowymi oparty jest na aplikacjach internetowych i mobilnych, spowodowało to gwałtowny wzrost użytkowników tych aplikacji, lub też pojawienie się zupełnie nowych rozwiązań. Niestety mnogość rozwiązań dostępnych na rynku posiada też wady.

Inwestując w różne instrumenty finansowe jak akcje, waluty, kryptowaluty, użytkownicy najczęściej wybierają maklera, który dane instrumenty oferuje, a także u którego inwestowanie jest najwygodniejsze i najtańsze (najniższa prowizja). Taka sytuacja powoduje, że osoba inwestująca w szeroki wachlarz instrumentów finansowych, posiada zazwyczaj kilku kont maklerskich, w których są te środki ulokowane. Śledzenie inwestycji oraz uzyskanie informacji o łącznym zysku bądź stracie, gdy wycena aktywów jest płynna, staje się bardzo utrudnione. Często prowadzi to do tego, że osoby inwestujące, podsumowanie swoich inwestycji trzymają w zwykłym arkuszu kalkulacyjnym. Ma to wiele wad jak: brak kontroli wprowadzania danych, brak aktualnych kursów instrumentów finansowych, mało przejrzysty interfejs oraz niska wygoda korzystania.

* 1. Porównanie dostępnych rozwiązań

Na rynku istnieje wiele aplikacji pozwalających na inwestowanie, czy szeroko pojęte zarządzanie finansami osobistymi. Rozwiązanie te można podzielić na dwie grupy: maklerów oraz dostawców wirtualnych portfelów inwestycyjnych. Aplikacje maklerskie dają użytkownikowi możliwość przechowywania, nabywania oraz sprzedaży oferowanych przez nich instrumentów finansowych. W Polsce jednymi z najpopularniejszych dostawców tych aplikacji są XTB, MBank oraz Bank ING. Wysoką popularnością cieszy się również aplikacja Revolut posiadająca atrakcyjne kursy wymiany walut. Wirtualne portfele inwestycyjna pozwalają stworzyć między innymi aplikacje Investing.com i Yahoo Finance – dla rynków zagranicznych, aplikacja GPW – dla polskiego rynku akcji. Pozwalają one na dodawanie wielu instrumentów finansowych oraz śledzenie ich aktualnych kursów. Żadna z aplikacji nie posiada jednak pełnego pakietu instrumentów mogących być atrakcyjnymi dla polskiego klienta (w aplikacjach zagranicznych - brak wielu polskich akcji, w aplikacji GPW - brak kursów obcych walut).

* 1. Możliwość zastosowania praktycznego

Implementacja aplikacji PersonalFinanceApp jest próbą odpowiedzi na wyżej zaadresowane problemy. Aplikacja pozwala na wgląd w swoje inwestycje poprzez dodawanie posiadanych przez użytkownika walut obcych jak i akcji (wybranych ze względu na możliwości demo) oraz śledzenia ich aktualnego kursu w złotówkach. Dodatkowo aplikacja posiada panel wydatków, gdzie użytkownik może wprowadzać swoje wydatki, nadawać im kategorie czy też filtrować je po dacie ich zaistnienia. Program posiada, również panel, w którym użytkownik może zdefiniować swój cel oszczędnościowy, a postęp w jego realizacji będzie obliczany na podstawie całości jego inwestycji.

Aplikacja posiada również tablicę podsumowującą widoczną po zalogowaniu, gdzie użytkownik błyskawicznie dowie się najważniejszych informacji jak: suma wszystkich oszczędności, suma wydatków w obecnym miesiącu, czy też postęp w realizacji celu oszczędnościowego.

* 1. Aspekty badawcze pracy

Głównym aspektem badawczym pracy jest prezentacja implementacji rozwiązania klient-serwerpoprzez podział rozwiązania na stronę kliencką i serwerową, które są niezależnymi od siebie programami. Analiza oraz prezentacja na przykładzie rozwiązania w jaki sposób te aplikacje się ze sobą komunikują. Pośrednimi aspektami badawczymi tej pracy, wynikającymi z formuły implementacji, jest zastosowanie RESTfulowego API, który jest interfejsem strony serwerowej oraz web scrapingu(skrobanie stron internetowych) jako alternatywa dla dedykowanych API do pozyskiwanie wartościowych informacji z sieci.

2. Przypadki użycia

Diagram przypadków użycia jest graficzną reprezentacją sposobów użycia programu. Celem diagramu jest ukazanie funkcjonalności systemu informatycznego oraz jego interakcji z aktorami[[1]](#footnote-1). W tym przypadku aktorami określa się osoby (użytkowników) lub też otoczenie systemu wchodzące z nim w interakcje, natomiast przypadkami użycia definiuje się zbiór scenariuszy opisujących w jaki sposób program reaguje na działania aktorów.

Poniższe diagramy przypadków użycia zostały sformułowane przy użyciu języka UML (zunifikowany język modelowania), który służy do formalnego opisu działania systemu. Na diagramach aktorem jest użytkownik aplikacji reprezentowany przez symbol ludzika, elipsy wraz z wewnętrznym opisem reprezentują przypadki użycia, a strzałki pomiędzy nimi opisują relację zachodzące między nimi: *<<include>>* uwzględnia jeden przypadek użycia w sekwencji zachowania innego, *<<extends>>* przypadek użycia rozszerza działanie innego.

Diagramy ukazują funkcjonalności aplikacji do finansów osobistych i są one podzielone na panele, z których są dostępne. Diagramy przypadków użycia przedstawiają:

* (Rysunek 2.1) logowanie i rejestracje, formularze umożliwiające gościowi odwiedzającemu stronę internetową aplikacji zalogowanie się bądź też założenia konta w serwisie

Diagram

Description automatically generated

Rysunek 2.1 - Diagram przypadków użycia (logowanie i rejestracja)

* Diagram

  Description automatically generated(Rysunek 2.2) dodawanie, edycje i usuwanie swoich oszczędności w postaci akcji spółek, walut obcych oraz gotówki przez użytkownika

Rysunek 2.2 Diagram przypadków użycia (panel oszczędności)

* (Rysunek 2.3) dodawanie i usuwanie wydatków użytkownika, filtrowanie wydatków po dacie ich wystąpienia

Diagram

Description automatically generated

Rysunek 2.3 Diagram przypadków użycia (panel wydatków)

* (Rysunek. 2.4) definiowanie i usuwanie celów oszczędnościowych, czyli sumy pieniędzy, którą użytkownik chce uzbierać

Diagram

Description automatically generated

Rysunek 2.4 Diagram przypadków użycia (panel celów oszczędnościowych)

2.1 Wymagania niefunkcjonalne i funkcjonalne

Wymaganiami niefunkcjonalnymi określa się ograniczenia, przy których dany system ma działać. Warunki te nie dotyczą bezpośrednio funkcjonalności oferowanych przez program, ale wpływają na formę ich implementacji. Te ograniczenia to między innymi:

* użyteczność – wymagania dotyczące sposobu użycia aplikacji
* zgodność – otoczenie systemowe oraz języki i technologie, w oparciu, o które program ma działać
* utrzymanie i zarządzanie – warunki dotyczące rozwoju systemu w przyszłości, skalowania aplikacji, dodawania nowych funkcjonalności oraz zarządzania kodem programu oraz jego działanie,

Wymagania niefunkcjonalne realizowane przez aplikację PersonalFinanceApp to:

Użyteczność

NF01 - prosty i czytelny interfejs użytkownika, podział funkcjonalności aplikacji na dedykowane panele grupujące tematycznie funkcjonalności

NF02 - intuicyjny interfejs oraz menu, poprzez które użytkownik przełącza się pomiędzy panelami

Zgodność

NF03 - aplikację można uruchomić lokalnie, do uruchomienia aplikacji użytkownik potrzebuje: Javy 11, powłoki bash, pakietu Node serve, przeglądarki internetowej z parserem JavaScript

Utrzymanie i zarządzanie

NF04 – organizacja strony serwerowej oraz klienckiej w pakiety oraz foldery według wzorca projektowego MVC (model-view-controller). Ułatwi to poruszanie się wewnątrz kodu aplikacji oraz łatwą rozbudowę, detekcję błędów oraz utrzymanie aplikacji

NF05 - podział aplikacji na stronę serwerową (*backend)* i stronę kliencką (*frontend)*, które są niezależne od siebie. Pozwala to na szybkie i łatwe zmiany w logice biznesowej jak i wyglądzie aplikacji bez ich wzajemnego wpływu na siebie

Wymagania funkcjonalne są to funkcje, które pełni system lub cześć systemu. Do wymagań funkcjonalnych zalicza się określenie zależności między danymi wejściowymi a danymi wyjściowymi programu, przetwarzanie dostarczanych danych, szczegółowy sposób działania elementów aplikacji, techniczny sposób realizacji wymagań niefunkcjonalnych[[2]](#footnote-2). Na podstawie wymagań funkcjonalnych formułuje się architekturę aplikacji.

Wymagania funkcjonalne realizowane przez aplikację PersonalFinanceApp to:

Bezpieczeństwo aplikacji

F01 – użytkownik może założyć konto w serwisie podając unikalny login i adres email w obrębie aplikacji oraz co najmniej 8 znakowe hasło

F02 – aby uzyskać dostęp do danych użytkownika należy zalogować się jego loginem i hasłem

F03 – z poziomu aplikacji użytkownik ma dostęp tylko do swoich danych i tylko on może je edytować i usuwać (nie dotyczy kursów akcji i walut dostępnych dla wszystkich użytkowników)

F04 - hasła użytkowników w bazie danych są haszowane klasą biblioteczną Bcrypt Password Encoder, która zapewnią niepowtarzalność klucza szyfrującego i trudny do złamania hasz

F05 – dostęp do funkcjonalności aplikacji można uzyskać tylko poprzez założenie konta w serwisie a następnie zalogowanie się na nie

Walidacja danych

F06 – dane pod kątem ich poprawności są weryfikowane dwukrotnie po stronie klienckiej aplikacji oraz stronie serwerowej

F07 – użytkownik podając nieprawidłowe dane w formularzach dostępnych w aplikacji zostanie powiadomiony, które dane są niepoprawne

F08 – dane aplikacji są przechowywane w bazie danych, która kontroluje przestrzeganie ich więzów integralności

F09 – wstawianie i wyciąganie danych odbywa się wewnątrz transakcji SQL zapewniających synchronizację danych i niepodzielność przeprowadzanych operacji na bazie danych

F10 – kursy akcji i walut są aktualne, aktualizacja danych odbywa się co 15 minut na podstawie aktualnych kursów instrumentów finansowych dostępnych w przeglądarce Google

Komunikacja aplikacji

F10 – strona kliencka aplikacji komunikuje się ze stroną serwerową w sposób bezstanowy za pomocą RESTfulowego API udostępnionego przez serwer

F11 – serwer aplikacji łączy się z bazą danych za pomocą konektora JDBC

Interfejs aplikacji

F12 – użytkownik łączy się z aplikacją lokalnie za pomocą przeglądarki internetowej

F13 – użytkownik łączy się z aplikacją lokalnie za pomocą udostępnionego API

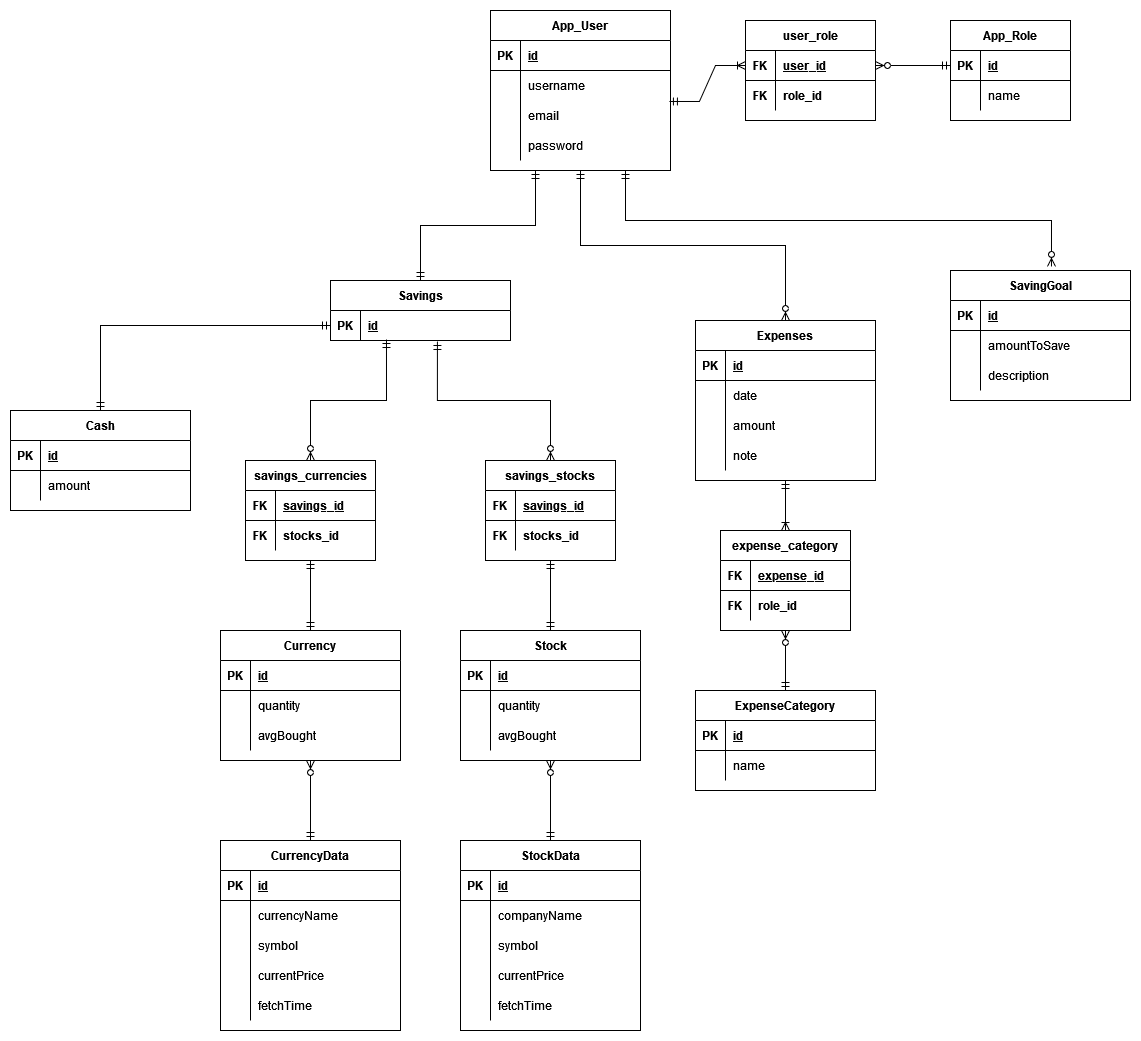
2.2 Dokumentacja techniczna

Diagram modelu bazy danych

Diagram modelu bazy danych jest graficzną reprezentacją encji oraz relacji między encjami występujących w bazie danych. Przedstawia sposób organizacji danych w bazie, pola występujące w encjach oraz relacje zachodzące między tabelami.

Diagram bazy danych jest pomocny przy tworzeniu zapytań do bazy danych, tworzenia widoków, które łączą wiele tabel oraz określania stopnia zależności między danymi zawartymi w bazie.

Rysunek 2.5 przedstawia diagram modelu bazy danych aplikacji PersonalFinanceApp. Baza ta została utworzona automatycznie za pomocą mapowania obiektowo-relacyjnego zaimplementowanego po stronie serwerowej aplikacji.



Rysunek 2.5 Diagram bazy danych aplikacji PersonalFinanceApp

Diagram klas

Diagram klas jest graficzną reprezentacją struktury systemu w modelach obiektowych[[3]](#footnote-3). Przedstawia klasy występujące w systemie wraz z ich opisem (pola, metody oraz zależności między klasami takie jak dziedziczenie, zawieranie się, implementowanie interfejsów). Diagram klas służy do ukazania szczegółów implementacji klas programu w sformalizowany sposób.

Diagram klas strony serwerowej aplikacji PersonalFinanceApp został wygenerowany za pomocą zintegrowanego środowiska programistycznego IntelliJ i zapisany w pliku class\_diagram.svg. Diagram przedstawia wszystkie klasy aplikacji wraz z ich polami, metodami oraz relacjami występującymi między klasami.

Dokumentacja WebAPI

API jest to interfejs programowania aplikacji, co w uproszczeniu oznacza zbiór funkcji oferowanych przez aplikację[[4]](#footnote-4). Poprzez te funkcję inna aplikacja, użytkownik lub programista ma możliwość komunikacji z aplikacją udostępniającą te API. WebAPI jest szczególnym rodzajem API, bo do komunikacji, jak sama nazwa wskazuje, używa protokołów sieci web takich jak protokół http.

Strona serwerowa aplikacji PersonalFinanceApp udostępnia RESTfulowe API. za pomocą którego strona kliencka aplikacji się z nią komunikuje. Dokumentacja API została wykonana za pomocą programu Postman, który jest jednym z najpopularniejszych platform służącym do budowania, testowania i dokumentacji API. Narzędzie te posiada ponad dwadzieścia milionów użytkowników oraz największy publiczny zbiór API[[5]](#footnote-5).

Dokumentacja jest przechowywana w postaci kolekcji żądań http (Postman collection), które obsługuje aplikacja. Kolekcja ta znajduje się w pliku api-doc.postman\_collection i przechowuje dane w formacie JSON. Z powodu tego, że format JSON jest mało czytelny dla człowieka, kolekcja ta została sparsowana za pomocą narzędzia postman\_doc\_gen[[6]](#footnote-6) i znajduje się w pliku api-doc.html. Kolekcję te można również zaimportować oraz wyświetlić w programie Postman wybierając w panelu narzędzia Collection >> import >> api-doc.postman\_collection.

Diagram komponentów React

Komponentem w bibliotece React nazywa się funkcję lub też klasę napisaną w języku JSX, która jest reprezentacją części interfejsu użytkownika. Komponenty pozwalają podzielić stronę kliencką aplikacji na niezależne od siebie, mogące być wielokrotnie używane moduły, za pomocą których można budować strony internetowe.

Strona klienckiej aplikacji PersonalFinanceApp została oparta na architekturze komponentów React, gdzie każdy element aplikacji jak formularz, menu, sekcja na stronie jest oddzielnym komponentem. Ich dokumentacja został wygenerowana przy pomocy środowiska IntelliJ i zapisana w pliku components\_diagram.svg.

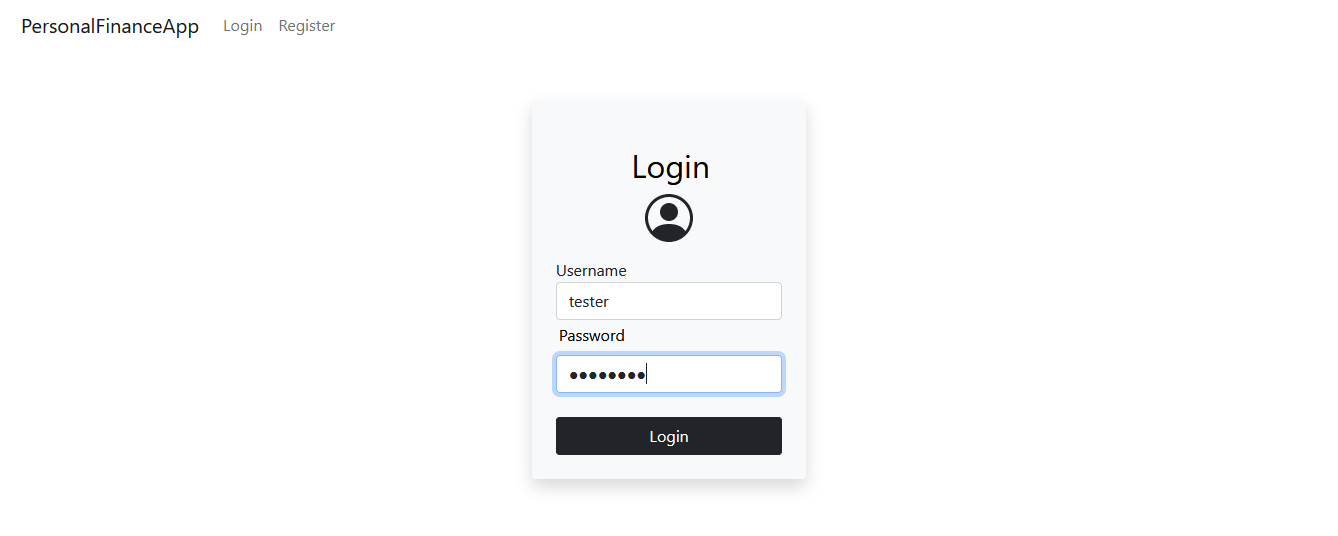
2.3 Interfejs użytkownika

Interfejsem użytkownika określa się obszar, w którym użytkownik wchodzi w interakcje z programem. Użytkownik do komunikacji z systemem musi posiadać narzędzie wejścia i wyjścia, jako że nie jest w stanie bezpośrednio komunikować się z maszyną. W aplikacjach webowych tym narzędziem staje się strona internetowa lub zestaw stron, z którymi użytkownik łączy się za pomocą przeglądarki internetowej.

Dla twórcy, wydawcy aplikacji ważne jest, żeby interfejs użytkownika był czytelny, intuicyjny oraz prosty w obsłudze. Powoduje to, że w firmach zajmujących się komercyjnie wytwarzaniem oprogramowanie znajdują się zespoły analityków oraz programistów, którzy badają użyteczność, intuicyjność oraz niezawodność interfejsu użytkownika.

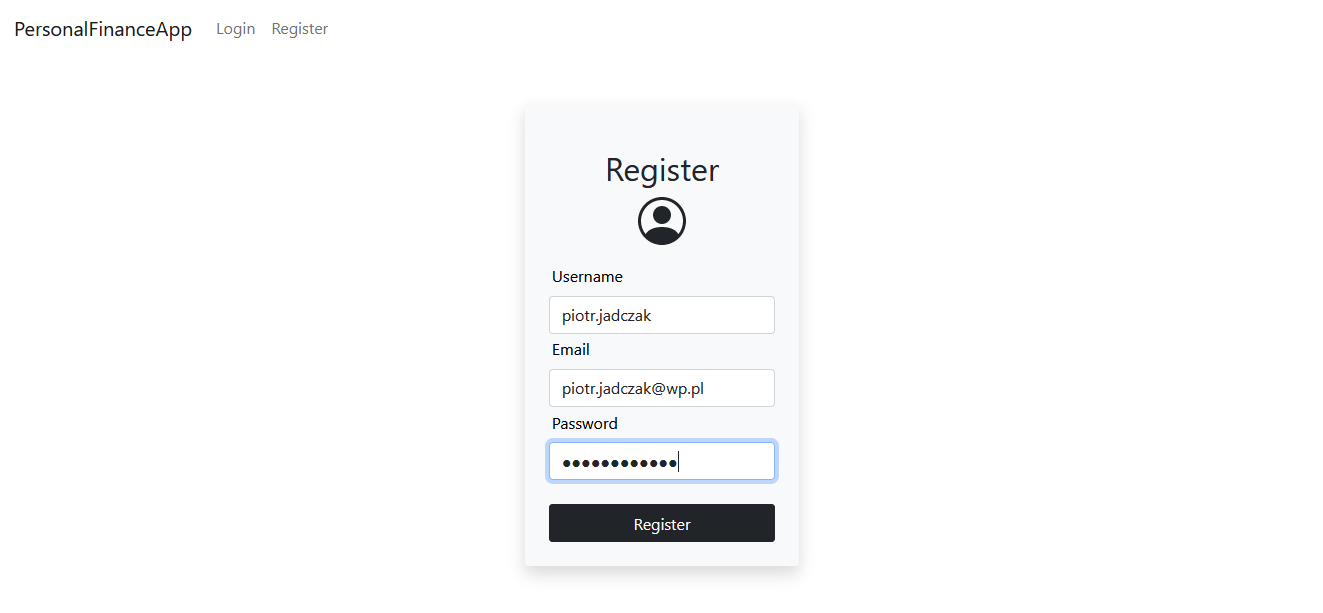
Aplikacja PersonalFinanceApp również stara się realizować te cele oraz wymagania. Robi to poprzez grupowanie funkcjonalności aplikacji w tematyczne panele, a dostęp do nich użytkownik ma za pomocą zakładek w menu na górze ekranu. Stylizacja stron HTML została wykonana przy użyciu jeżyka CSS3.

Interfejs użytkownika aplikacji PersonalFinanceApp składa się z paneli:



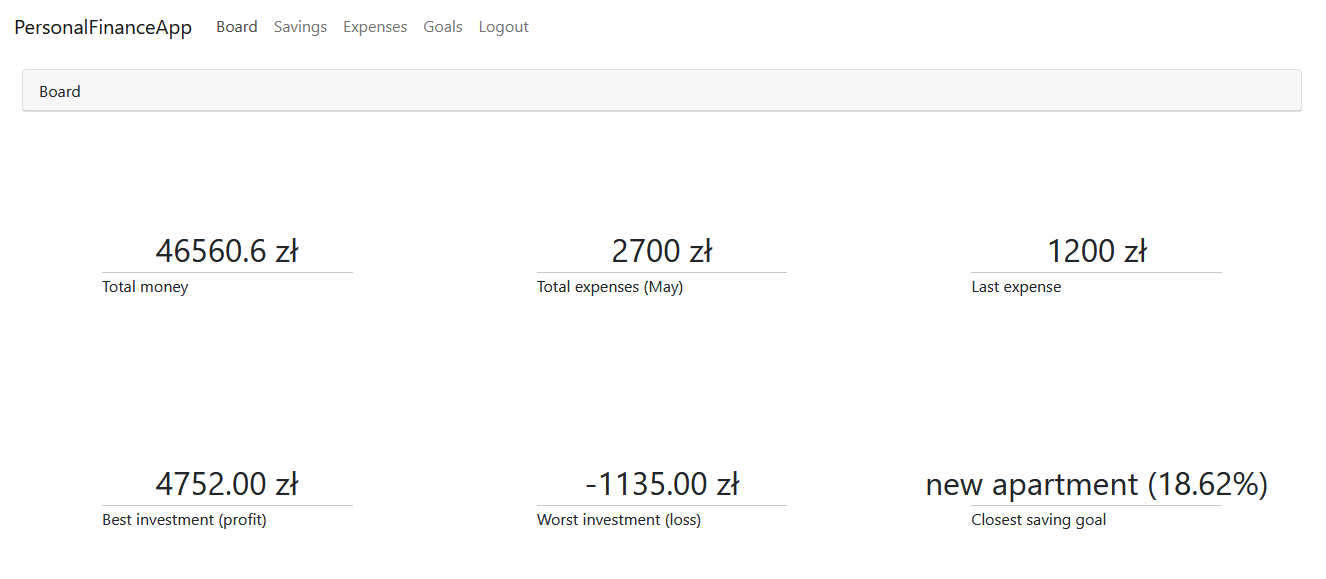
Rysunek 2.6 Panel logowania

Rysunek 2.6 formularz logowania, pozwala na zalogowanie się do aplikacji poprzez podanie prawidłowego loginu i hasła dla wybranego użytkownika



Rysunek 2.7 Panel rejestracji

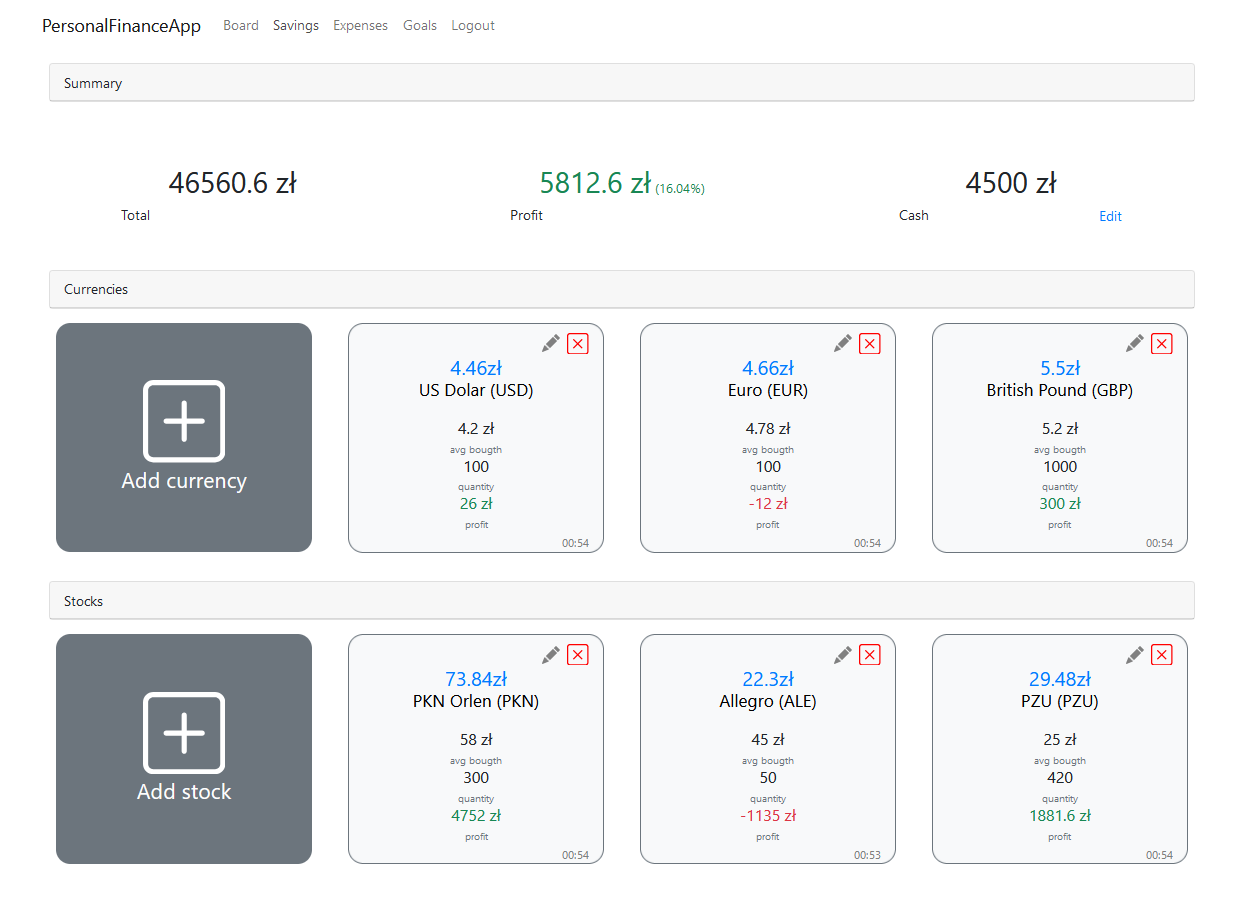
Rysunek 2.7 formularz rejestracji, pozwala na założenie konta gościowi serwisu poprzez podanie unikalne dla aplikacji loginu, adresu email, oraz co najmniej ośmioznakowego hasła



Rysunek 2.8 Panel tablicy podsumowania

Rysunek 2.8 tablica widoczna bezpośrednio po zalogowaniu w aplikacji. Zawiera 6 pozycji podsumowujących przechowywane dane użytkownika:

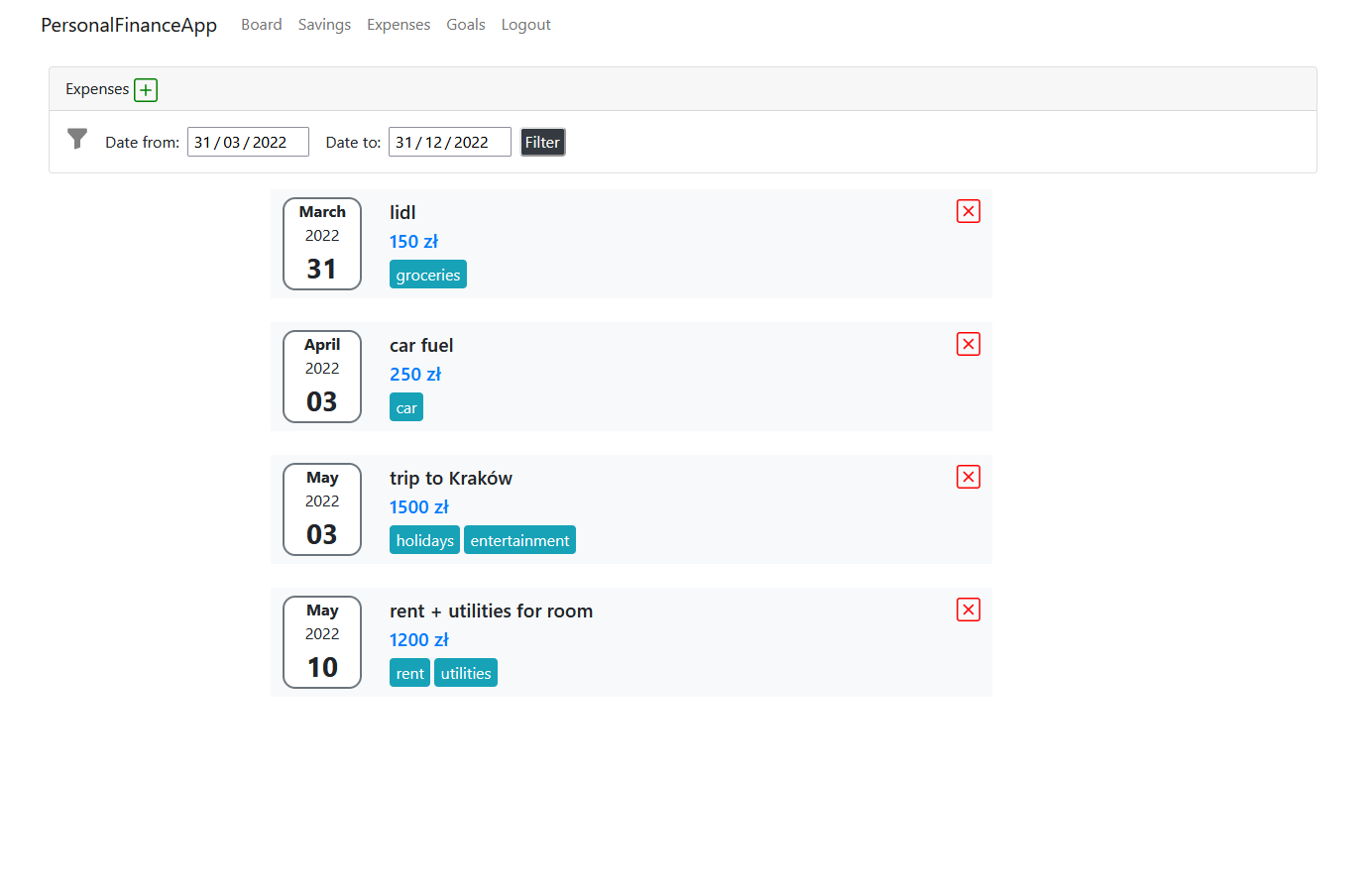
* Total money - posiadane przez użytkownika oszczędności (gotówka i inwestycje)
* Total expenses - suma wydatków w obecnym miesiącu kalendarzowym
* Best investment (profit) – największy zysk nominalnie z pojedynczej inwestycji (brak zysku wyświetla komunikat: „No profit yet”)
* Worst investment (loss) - największy strata nominalnie z pojedynczej inwestycji (brak straty wyświetla komunikat: „No loss yet”)
* Closest saving goal – procentowy postęp i nazwa celu oszczędnościowego, który jest najbliżej osiągnięcia 100% (brak celu wyświetla komunikat: „No goals yet”)



Rysunek 2.9 Panel oszczędności

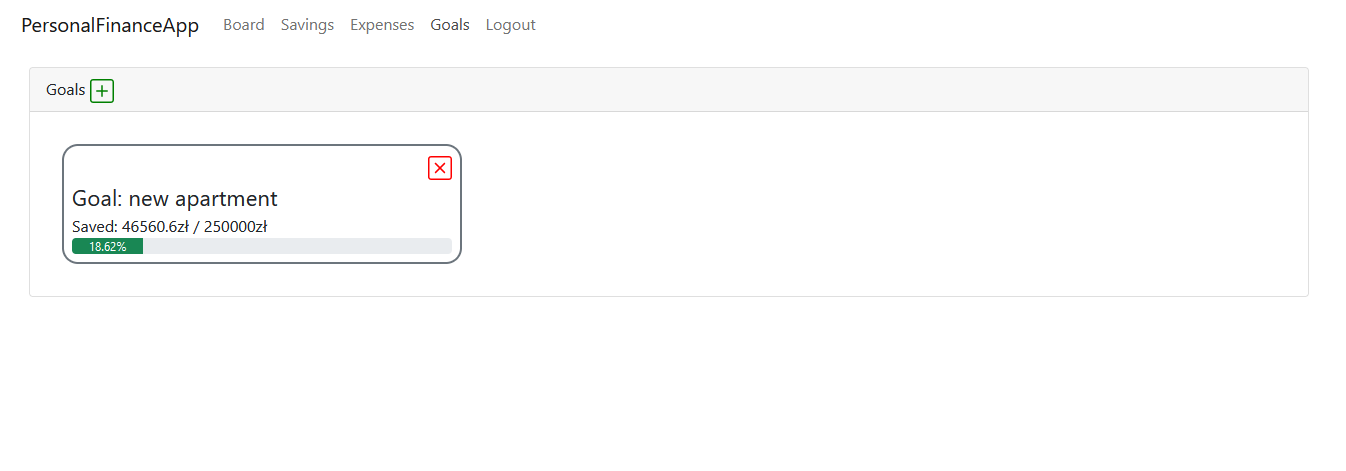
Rysunek 2.9 panel oszczędności, który składa się z sekcji:

* Total – pole wyświetlające sumę wszystkich oszczędności: waluty, akcje, gotówka
* Profit – pole wyświetlające nominalny jak i procentowy zysk, bądź stratę z inwestycji
* Cash – pole wyświetlające gotówkę użytkownika, poprzez przycisk Edit, użytkownik może edytować tą wartość
* Currencies – sekcja umożliwiająca dodawanie, edycję oraz usuwanie walut obcych, każda pozycja posiada ilość posiadanej waluty, średni kurs jej zakupu, obecny kurs, zysk lub stratę z pozycji, czas aktualizacji kursu
* Stocks – sekcja umożliwiająca dodawanie, edycję oraz usuwanie akcji, każda pozycja posiada ilość posiadanych akcji, średni kurs ich zakupu, obecny kurs, zysk lub stratę z pozycji, czas aktualizacji kursu



Rysunek 2.10 Panel wydatków

Rysunek 2.10 panel umożliwia dodawanie i usuwanie wydatków, filtrowanie ich po datach, wydatki składają się z daty ich wystąpienia, kwoty, opisu oraz ich kategorii



Rysunek 2.11 Panel celów oszczędnościowych

(Rysunek 2.11) panel celów oszczędnościowych, umożliwia dodawanie oraz usuwanie celów oszczędnościowych, cel oszczędnościowy składa się z kwoty do zebrania, opisu celu jak i paska postępu realizacji celu

3. Implementacja

Strona serwerowa

Implementacja strony serwerowej aplikacji została wykonana przy użyciu frameworka Spring Boot w wersji 2.6.4. Spring jest to otwarto źródłowa platforma, która ułatwia budowę aplikacji opartych na języku Java[[7]](#footnote-7). Najważniejszym elementem frameworka jest dostarczenie wsparcia infrastrukturalnego na poziomie aplikacji. Realizowane jest to poprzez dostarczenie szerokiego zestawu narzędzi, które zapewniają techniczną obsługę wymagań programu jak między innymi: obsługa bazy danych, zabezpieczenie aplikacji, komunikacja webowa. Pozwala to zespołom programistycznym, pracującym w tym frameworku, skoncentrować się na implementacji procesów biznesowych, bez zbędnych powiązań ze środowiskiem wdrożeniowym aplikacji. Spring Boot, jak sama nazwa wskazuje (boot pl. Uruchomić), jest rozszerzeniem frameworka Spring, które skraca i upraszcza konfigurację aplikacji oraz dostarcza kontener aplikacji umożliwiający jej uruchomienie jednym kliknięciem.

W przypadku strony serwerowej aplikacji PersonalFinanceApp całość konfiguracji zawiera się w zaledwie dwóch plikach. Pierwszym z nich jest plik build.gradle. Zawiera on dane dotyczące bibliotek, wersji Spring, Javy, które są wykorzystywanie podczas budowania aplikacji przez narzędzie Gradle. Budowanie polega na kompilacji klas języka Java aplikacji do postaci kodu bajtowego, a następnie umieszczenie ich, wraz z wykorzystywanymi bibliotekami i plikami, w jednym archiwum jar. Zaletą takiego rozwiązania jest to, że całość aplikacji znajduje się w jednym archiwum, co ułatwia umieszczenie jej na serwerze i pomaga w zarządzaniu kolejnymi wersjami aplikacji. Drugim plikiem konfiguracyjnym jest plik application.properties. Zawiera on ustawienia połączenia z bazą danych, ustawienia mapowania obiektowo-relacyjnego oraz zmienne aplikacji używane do ustawień autoryzacji. Ponadto, aplikacja do obsługi żądań http wykorzystuje serwer Apache Tomcat, który jest domyślne wbudowany w ten framework.

Aplikacja PersonalFinanceApp do implementacji oferowanych funkcjonalności oraz aspektów technicznych jej działania używa wielu bibliotek. Za dostarczenie mapowania obiektowo-relacyjnego odpowiada biblioteka Spring Data JPA umożliwiająca tworzenie relacji między klasami oraz operacje na bazie danych SQL przy użyciu interfejsu Repository. Połączenie z bazą danych za pomocą konektora JDBC umożliwia biblioteka HSQLDB. Walidacje danych wejścia i wyjścia aplikacji zapewnia kolekcja Spring Validation. Biblioteki Spring Security oraz JJWT dostarczają możliwość implementacji bezstanowej autentykacji użytkowników API aplikacji za pomocą tokenów JSON Web Token. JSOUP dostarcza klasy oraz metody pozwalające na skrobanie stron internetowych wykorzystane w aplikacji PersonalFinanceApp do pobierania aktualnych kursów walut oraz akcji. Biblioteka Spring Web zapewnia kontrolery REST wykorzystane w udostępnianym przez serwer WebAPI.

Strona kliencka

Aplikacja kliencka została oparta na React.js, bibliotece języka JavaScript służącej do implementacji interfejsów użytkownika. React pozwala na tworzenie interaktywnych widoków poprzez składanie ze sobą małych, izolowanych od siebie elementów. Te elementy nazywane są komponentami React. Komponenty React używają języka JSX, rozszerzenia języka JavaScript pozwalającego na umieszczanie w funkcjach języka kodu HTML. Izolacja komponentów oznacza, że zawierają one wewnątrz logikę oraz funkcję, które zarządzają jej stanem, a są niewidoczne dla reszty aplikacji. Takie podejście upraszcza tworzenie i zarządzanie złożonymi interfejsami użytkownika. Elementem wyróżniającym bibliotekę React.js jest wykorzystanie wirtualnego DOM (pl. Obiektowy Model Dokumentu). DOM oznacza sposób reprezentacji dokumentów XML i HTML za pomocą struktury drzewa, w którym każdy węzeł reprezentuje obiekt, który jest częścią dokumentu. Wykorzystanie wirtualnego DOM przez Reacta, polega na przechowywaniu stanu aplikacji w pamięci. Dzięki takiemu podejściu, React jest w trakcie wykryć różnicę między wirtualnym, a rzeczywistym DOM i precyzyjnie zaktualizować model. Pozwala to na szybsze i płynniejsze aktualizacje widoków, oraz optymalizacje wydajności aplikacji.

Interfejs aplikacji PersonalFinanceApp został wykonany zgodnie ze wzorcem Single Page Application. Oznacza to, że strona kliencka aplikacji posiada tylko jeden plik HTML, a jego aktualizacja jest wykonywana przy użyciu komponentów React. Charakterystyczną cechą aplikacji SPA jest to, że nie przeładowuje ona całej strony internetowej, a jedynie jej fragmenty. Ponadto w implementacji zostały użyte biblioteki języka React wspomagające tworzenie interfejsu użytkownika. Stylowanie strony zostało wykonane przy użyciu biblioteki Bootstrap, które zawiera gotowe pakiety stylów, które dołącza się do elementów HTML, za pomocą predefiniowanych klas. Do wykonania formularzy na stronie wykorzystano biblioteki React Modal oraz React Select do utworzenia wyskakujących okien, które zawierają formularze dostępne w aplikacji. Przełączanie się pomiędzy widokami na stronie za pomocą górnego Menu umożliwia biblioteka React Router. Do implementacji komunikacji ze stroną serwerową aplikacji użyto bibliotekę axios. Pozwala ona na wysyłanie asynchronicznych, czyli równoczesnych w czasie, żądań http na wskazane adresy, a następnie ich odpowiednie obsłużenie. Wszystkie widoki interfejsu dostępne w aplikacji PersonalFinanceApp zostały ukazane w rozdziale 2.9.

3.1 Architektura aplikacji

Architektura aplikacji PersonalFinanceApp została oparta na modelu klient-serwer. Polega on na podziale zadań i obowiązków pomiędzy dostawców zasobów oraz usług, których w modelu określa się serwerami, a klientów zajmujących się zgłaszaniem do serwerów żądań usług[[8]](#footnote-8). Klienci zwykle nie udostępniają swoich zasobów, a jedynie pozyskują je z zawartości i usług serwerów. Do komunikacji serwer-klient wymieniają wiadomości według wzorca żądanie-odpowiedź. Klient wysyła żądanie, na które serwer udziela mu odpowiedzi. W aplikacjach webowych ta wymiana informacji jest dokonywana przy użyciu protokołów sieci web, najczęściej protokołu http. Serwer w tym modelu wystawia API, które odbiera żądania, a następnie udziela odpowiedzi stając się warstwą dostępu do usług serwera.

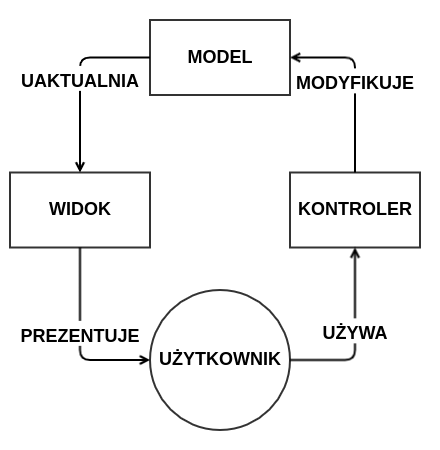
Aplikacja PersonalFinanceApp realizuje model klient-serwer poprzez podział rozwiązania na dwie niezależne aplikacje: część kliencką, wykonanej przy użyciu biblioteki React oraz część serwerową opartą na frameworku Spring. Do komunikacji między sobą, programy używają protokołu komunikacji http. Wysyłane żądania przez klienta są odbierane przez API wystawione przez serwer aplikacji. Następnie są one obsługiwane według zaimplementowanej logiki biznesowej programu i wysyłane są do klienta odpowiedzi. Strona kliencka aplikacji do finansów osobistych sama nie udostępnia żadnych danych, a wszystkie informacje pozyskuje z serwera. Pełni wyłącznie role interfejsu użytkownika i reagując na jego zachowania wysyła odpowiednie żądania do serwera i wyświetla otrzymane od niego dane.

Warto wspomnieć, że frameworki webowe przystosowane do modelu klient-serwer, często wspierają wielowątkowość aplikacji co zwiększa ich wydajność. Aplikacja PersonalFinanceApp również wykorzystuje tą możliwość. Spring domyślnie replikuje kontrolery REST obsługujące żądania http. Strona kliencka aplikacji również korzysta z wielowątkowości wykonując asynchronicze zapytania do serwera (AJAX).

3.2 Wzorce projektowe

Aplikacje webowe komercyjne jak i otwarto źródłowe (open-source) są zazwyczaj dużymi projektami często składającymi się z tysięcy, bądź nawet dziesiątek tysięcy klas, plików danych oraz plików konfiguracyjnych. Zarządzanie, rozwój oraz utrzymanie tak wielkich projektów wymusza podział plików źródłowych według określonego wzoru na zbiory, pakiety, które w pewien sposób je pogrupują. Jednym ze wzorców służących do takiego podziału kodu programu na części jest wzorzec MVC (model-view-controller).

Zastosowania wzorca MVC polega na podziale aplikacji webowej na części modelu, widoku oraz kontrolera. Część modelu odpowiedzialna jest za implementację logiki biznesowej aplikacji, czyli tego w jaki sposób dane są przetwarzane, jakie są pomiędzy nimi zależności oraz jak mają działać funkcjonalności aplikacji. Rolą widoku we wzorcu MVC jest wizualna prezentacja części modelu aplikacji, czyli danych lub stanu, w ramach wybranego interfejsu. Kontroler ma za zadanie przyjąć dane wejściowe od użytkownika aplikacji, który może być zarówno osobą, ale także zewnętrzną aplikacją, a następnie zareagować na te dane powodując odwołanie do modelu, który następnie odpowiednio uaktualni widok. Schemat podziału aplikacji na te części i pełnione przez nie zadania ukazano na rysunku 3.1.



Rysunek 3.1 Schemat działania modelu MVC [żródło: https://pl.wikipedia.org/wiki/Model-View-Controller]

Strona kliencka aplikacji PersonalFinanceApp jest odpowiedzialna za realizację części widoku, poprzez wyświetlanie interfejsu użytkownika oraz danych dla niego dostępnych w postaci stron internetowych HTML. Zastosowaniem części model wzorca MVC zajmuje się strona serwerowa aplikacji. Realizacja ta odbywa się poprzez wstawianie oraz wydobywanie informacji z bazy danych aplikacji według zdefiniowanej logiki biznesowej. Odpowiadają za to klasy pogrupowane w pakiety języka Java:

* pakiet model – reprezentacja klasowa encji występujących w bazie danych, enkapsulacja odbieranych i wysyłanych danych w obiekty DTO (*data transfer object*)
* pakiet repository – reprezentacja obiektowa bazy danych, odpowiada za komunikację z bazą danych, wstawianie, wyciąganie, usuwanie oraz modyfikację danych
* pakiet service – zawiera serwisy posiadające logikę biznesową aplikacji

Część kontroler wzorca jest realizowana zarówno przez klienta jak i serwer aplikacji. Strona kliencka realizuje tę funkcję poprzez pobieranie danych od użytkownika za pomocą formularzy HTML oraz reagowaniem na jego żądania zmiany widoków, gdy użytkownik chce wyświetlić formularz lub przełączać się pomiędzy panelami. Strona serwerowa również realizuje ten aspekt poprzez udostępnianie API, które zawarte jest w pakiecie controller. Zawiera one wszystkie punkty końcowe, czyli adresy URL pod które można wysyłać żądania. Frontend wykorzystując te API, ma dostęp do funkcjonalności aplikacji i wysyłając odpowiednie żądania uaktualnia model na serwerze oraz używając odpowiedzi udzielonych przez serwer, aktualizuje widok interfejsu użytkownika. Należy zauważyć, że podział aplikacji według wzorca MVC często nie jest restrykcyjny (tak jak w wypadku aplikacji PersonalFinanceApp), a jedynie sugeruje programiście zajmującym się implementacją funkcjonalności programu, gdzie powinien umieścić swój kod w zależności od tego co chce osiągnąć.

Implementacja strony serwerowej zwiera również realizację wzorca projektowego dependency injection (wstrzykiwanie zależności), charakterystycznego dla aplikacji frameworka Spring. Wzorzec wstrzykiwania zależności opiera się na tym, że obiekt aplikacji (Bean) otrzymuje inny obiekt lub obiekty od którego zależy, czyli używa do oferowanych przez siebie funkcjonalności bądź sposobu zachowania. Schemat wstrzykiwanie zależności wykorzystanego w aplikacji PersonalFinanceApp przedstawiono na rysunku 3.2.



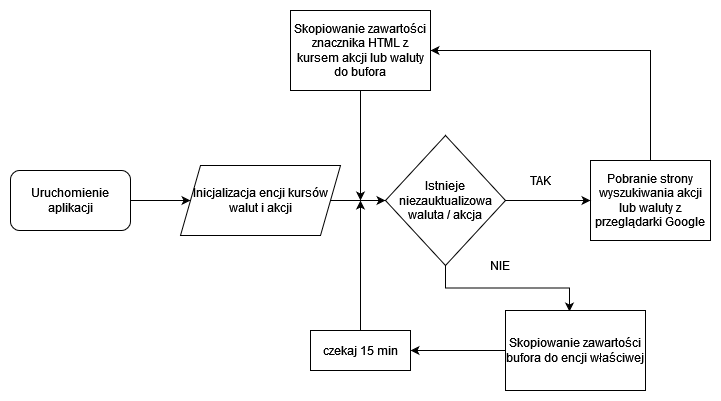
Rysunek 3.2 Schemat wzorca wstrzykiwania zależności wykorzystanego w aplikacji [źródło własne]

Kontrolery aplikacji odbierają żądania http użytkowników. Do udzielenia odpowiedzi wykorzystują serwisy, które oferują zestawy operacji dostępnych w ich zakresie. Kontroler wybiera, według logiki biznesowej, metodę oferowaną przez serwis, a ten używa repozytorium do przeprowadzenia operacji na bazie danych aplikacji. Model ten realizuje również wzorzec separation of concerns (podział obaw), czyli podział klas aplikacji na sekcje według ich zastosowania. W tym przypadku sekcja kontroler jest interfejsem, który umożliwia interakcje aplikacji z użytkownikiem bądź warstwą kliencką. Sekcja serwis zawiera logikę biznesową aplikacji a sekcja repozytorium jest warstwą dostępu do danych, czyli jest odpowiedzialna wyłącznie za przeprowadzanie operacji na bazie danych.

3.3 Web scraping

Aplikacja do finansów osobistych pozyskuje aktualne kursy akcji oraz walut za pomocą web scrapingu. Web scraping (pl. skorbanie stron) to metoda wydobycia danych ze stron internetowych. Czynność ta, może być wykonywana manualnie, ale najczęściej odnosi się do zautomatyzowanego procesu wykonywanego przez programy komputerowe[[9]](#footnote-9). Zasada działania skrobania stron jest bardzo prosta. Program pobiera wybraną stronę internetową, a następnie wybiera z niej dane, takie jak znaczniki HTML, zdjęcia, fragmenty tekstu. Pozyskane dane mogą być bezpośrednio zapisywane przez ten program, lub też dalej przez niego przetwarzane by uzyskać pożądane informacje. Web scraping może być alternatywą dla dedykowanych API udostępniających zestawy danych, gdy takowe API nie istnieją lub są nieosiągalne dla twórcy aplikacji. Niestety, największą wadą web scrapingu, w stosunku do dedykowanych API, jest niska wydajność. Spowodowane jest to tym, że aplikacja skrapująca dane musi za każdym razem pobierać całe strony internetowe, czyli wielkie pliki, które składają się w dużym stopniu z niepotrzebnych aplikacji informacji.

Schemat implementacji pobierania aktualnych kursów wybranych akcji oraz walut obcych dostępnych w aplikacji PersonalFinanceApp znajduje się na rysunku 3.3.



Rysunek 3.3 Schemat implementacji pobierania aktualnych kursów akcji oraz walut [źródło własne]

Podczas uruchamiania serwera aplikacji tworzone są encję, których zadaniem będzie przechowywanie kursów akcji oraz walut. Dane do inicjalizacji tych encji są pobierane z plików currency\_input.csv oraz stock\_input.csv. Następnie wykonywane jest pobranie kursów. Dla każdej akcji oraz waluty pobierana jest strona wyszukiwania z przeglądarki Google zawierająca jej nazwę oraz dopisek kurs. W kolejnym kroku, pobierana jest zawartość znacznika HTML zawierająca aktualną wycenę instrumentu oraz zapisywana w bazie danych. Wybór tego znacznika jest trywialną sprawą, bowiem przeglądarka Google, dla wszystkich oferowanych w aplikacji instrumentów finansowych, zawiera odpowiednio dla akcji i walut tą samą klasę przy znaczniku posiadających ich wycenę. Wyceny aktywów są najpierw zapisywane do encji buforowej. Decyzja ta spowodowana jest tym, że operacja skrapowania jest powolna (dla walut zajmuję kilkanaście, a dla akcji kilkadziesiąt sekund). Buforowanie pozwala na uniknięcie problemu brudnego odczytu, czy też blokowania encji zawierającej kursy instrumentów finansowych przez system transakcji, gdyby dane te zapisywane były bezpośrednio do encji udostępnianej dla użytkownika aplikacji. Po pobraniu wszystkich kursów, dane kopiowane są z bufora do encji właściwej. Następnie uruchamiany jest stoper, który całą procedurę aktualizacji kursów akcji oraz walut będzie wykonywał regularnie co piętnaście minut.

3.4 RESTful API

Komunikacja między klientem, a serwerem wewnątrz aplikacji PersonalFinanceApp odbywa się za pomocą RESTfulowego API. Styl architekturalny REST (Representational state transfer) został zaproponowany przez Roya Thomasa Fieldinga w jego rozprawie doktorskiej „Architectural Styles and the Design of Network-based Software Architectures”. W swojej pracy autor określił sześć warunków, które musi spełniać architektura REST[[10]](#footnote-10):

1. Wykorzystanie architektury klient-serwer

Czym jest wzorzec klient-serwer oraz jak implementuje go aplikacja opisano w rozdziale 3.1.

2.Bezstanowość

Zasada ta polega na tym, że żądania wysyłane do API od klienta muszą zawierać wszystkie informacje potrzebne serwerowi do jego obsługi. Serwer nie zawiera mechanizmu sesji. W aplikacji PersonalFinanceApp żadne zapytanie nie korzysta z informacji z sesji, a bezstanowość najlepiej można dostrzec w autentykacji użytkowników przy pomocy tokenów co zostało opisane w rozdziale 3.5.

3.Wykorzystanie pamięci podręcznej

Serwer aplikacji może określać, kiedy udzielana przez niego odpowiedź powinna być zapisana w pamięci podręcznej, a kiedy nie. Mechanizmu tego aplikacja do finansów osobistych używa zapisując token otrzymany po udanym logowaniu, który następnie jest używany do autoryzacji wszystkich żądań wysyłanych przez klienta do serwera.

4.Jednolity interfejs

Interfejs API między klientem, a serwerem musi zgodny być z czterema zasadami:

* identyfikacja zasobu – zasób na serwerze musi być identyfikowalny, aby serwer mógł się do niego dostać i przeprowadzać na nim operacje. W API aplikacji wszystkie metody odnoszące się do konkretnego obiektu, używają id (kluczy głównych) encji do jednoznacznego ich określenia
* operowanie zasobami poprzez reprezentację – klient otrzymuje od serwera reprezentacje zasobu, która musi zawierać wystarczająco informacji, żeby móc go wykorzystać. Serwer implementowanej aplikacji do tego celu wykorzystuje obiekty DTO (data transfer object), które grupują dane będące następnie używane przez klienta.
* wykorzystanie komunikatów samoopisujących się – każde zapytanie musi zwierać wszystkie informacje potrzebne odbiorcy do jego obsłużenia w tym: typ protokołu, format danych wiadomości, żądana operacja. API aplikacji PersonalFinanceApp w każdym zapytaniu korzysta z protokołu http oraz określa żądano operacje jak GET, POST, PUT, DELETE. Format wysyłanych oraz odbieranych wiadomości to JSON.
* hipermedia jako stan silnia aplikacji – dane wysyłane przez serwer muszą zawierać dodatkowe akcje i zasoby dostępne dla klienta, żeby dotrzeć do uzupełniających informacji, żeby obsłużyć odpowiedź serwera. Aplikacja PersonalFinanceApp nie potrzebuje korzystać z tego mechanizmu do implementacji swoich funkcjonalności.

5.System warstwowy

System składa się z wielu warstw ułożonych hierarchicznie, z których każda dostarcza usługi warstwie nad nią, co w rezultacie sprowadza się do tego, że konsumuje usługi poniżej. Przykładem jest tu warstwa autentykacji implementowanej aplikacji, przez którą przechodzi każde zapytanie do API zanim trafi do logiki. Warstwa ta została opisana w rozdziale 3.5.

6.Kod na żądanie (opcjonalnie)

Udostępnianie klientowi kodu wykonywalnego przez serwer. Dokładny opis tej zasady został pominięty, bo nie jest wykorzystywany przez implementacje aplikacji.

Podsumowując, o API aplikacji PersonalFinanceApp można powiedzieć, że jest RESTful (pl. w pełni REST), ponieważ spełnia wszystkie wymagania architektury REST.

3.5 Autentykacja użytkowników

4. Testowanie manualne

4.1 Testowanie strony serwerowej

4.2 Testowanie strony klienckiej

Zakończenie

5. Bibliografia

1. Diagramy przypadków użycia:

<https://wolski.pro/diagramy-uml/diagram-przypadkw-uzycia/>

1. Wymagania funkcjonalne:

<https://pl.wikipedia.org/wiki/Wymagania_funkcjonalne>

1. Diagram klas:

<https://pl.wikipedia.org/wiki/Diagram_klas>

1. WebAPI:

<https://www.geeksforgeeks.org/what-is-web-api-and-why-we-use-it>

1. Postman:

<https://en.wikipedia.org/wiki/Postman_(software)>

1. Postman doc gen:

<https://github.com/karthiks3000/postman-doc-gen>

1. Spring Boot:

<https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html>

1. Wzorzec klient-serwer: <https://en.wikipedia.org/wiki/Client%E2%80%93server_model>
2. Web scraping:

<https://miroslawmamczur.pl/web-scraping-co-to-i-jakie-sa-dobre-praktyki/>

1. Architektura REST:

kurs Cisco DevNet Associate, moduł 4

1. Źródło: https://wolski.pro/diagramy-uml/diagram-przypadkw-uzycia/ [odczyt 12.06.2022] [↑](#footnote-ref-1)
2. źródło: <https://pl.wikipedia.org/wiki/Wymagania_funkcjonalne> [odczyt 30.05.2022] [↑](#footnote-ref-2)
3. źródło: <https://pl.wikipedia.org/wiki/Diagram_klas> [odczyt 30.05.2022] [↑](#footnote-ref-3)
4. źródło: <https://www.geeksforgeeks.org/what-is-web-api-and-why-we-use-it> [odczyt 23.05.2022] [↑](#footnote-ref-4)
5. źródło: <https://en.wikipedia.org/wiki/Postman_(software)> [odczyt 23.05.2022] [↑](#footnote-ref-5)
6. źródło <https://github.com/karthiks3000/postman-doc-gen> [odczyt 30.05.2022] [↑](#footnote-ref-6)
7. źródło: <https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html> [odczyt 07.06.2022] [↑](#footnote-ref-7)
8. źródło: <https://en.wikipedia.org/wiki/Client%E2%80%93server_model> [odczyt 07.06.2022] [↑](#footnote-ref-8)
9. źródło: <https://miroslawmamczur.pl/web-scraping-co-to-i-jakie-sa-dobre-praktyki/> [odczyt 12.06.2022] [↑](#footnote-ref-9)
10. źródło: kurs Cisco DevNet Associate, moduł 4 [↑](#footnote-ref-10)